# POWER CONSUMPTION TESTING FOR IOS

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Yannick Mingashanga Kwete

In Partial Fulfillment
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

October 2013

Fargo, North Dakota

# North Dakota State University
## Graduate School

**Title**

Power Consumption Testing for iOS

**By**

Yannick Mingashanga Kwete

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

**MASTER OF SCIENCE**

SUPERVISORY COMMITTEE:

Ken Magel
Chair

Gursimran Walia

Carlos Hawley

Approved:

2/11/14
Date

Dr. Brian Slator
Department Chair

**ABSTRACT**

The shift from traditional software development for personal computers to mobile applications on iOS comes with new challenges and considerations. Software development teams similar to my work group built expertise in delivering quality products for the Mac OS platform. Although the core development and testing processes largely remain the same on iOS devices (iPhones and iPads), a major concern is around an application's power consumption. The engineering challenge is to build innovative applications that do not drain the battery too quickly.

Power consumption testing on iOS is an important area to validate and optimize to deliver quality applications to mobile customers. As a developing space, there is limited information available on testing for battery. However, given that all application activities consume CPU cycles, developers should strive to optimize CPU usage over file I/O and network operations to reduce an application's power consumption on iOS devices.

**TABLE OF CONTENTS**

## LIST OF TABLES

## LIST OF FIGURES

Figure Page

# CHAPTER 1. INTRODUCTION

Application development on iOS comes with a new set of challenges and considerations not seen on the desktop software development. On iOS, applications are sandboxed, resources are limited and the mobile devices (iPhones and iPads) rely on battery power with a finite capacity per charge. While my work group for many years built expertise developing and testing software for the Mac OS X platform, the transition to mobile application development required evaluating new risks and creating a mitigation test plan to ensure that the team had the right quality guards tailored for iOS applications.

On the management side, there was a similar concern raised on how the applications would fare on iOS from a power consumption (battery usage) standpoint. They understood that battery was a new area that the team had not considered as much in the past and it was important to have a test strategy in place to address this space.

In addition, the resounding question across the engineering group was – what should the team test (look for) and optimize to reduce an application's power consumption on iOS devices? Given the novelty of the iOS platform, information and documentation on testing for battery life is scarce. Apple through its developer's channel provides guidance on engineering applications for iOS; this information is available from its Worldwide Developers Conference (WWDC) video sessions and tools reference documentation. The recommendation focus on three key areas for validating power consumption and battery life (Apple Inc. 2011):

1. Reduce network traffic by managing the number of connections to servers, amount of data transferred using compression and supporting resume-able transmissions. Some optimization approaches include bursting (send and/or receive all at once), caching and using compact data formats.

2. Allow the device to sleep if there is no user interaction with the application. Battery life is rooted on sleep, so a key optimization is to extend the stand by time of the device as much as possible.
3. Avoid unnecessary redraws and adjust to lower frames per seconds when possible (if quality is not impacted) to reduce CPU and GPU activity.

Further investigation on iOS battery testing only yields suggestions for end users on do's and don'ts to maximize battery life. However, this is not helpful for application developers as these recommendations are related to end user controlled settings.

As a development group building low level components that are later integrated into an iOS application, the major concerns revolve on the core operations performed components perform. In my group's case, the key operations are: 1) accessing the network, 2) file I/O (writing to and reading from disk), and 3) processing data (CPU computation). Given this set of operations, what can developers do to optimize for longer battery life while not compromising functionality? Applications that integrate low level components expect them to perform specific tasks on their behalf. Ideally, a component's functionality is encapsulated to hide its implementation details which is a good design decision that leads to flexibility and decoupling (Reddy 2011, 159). This design approach opens up opportunities to optimize for battery efficiency in isolation and the changes will be transparent when integrated into the final application. Since all software activities on iOS consume CPU cycles, could optimizing CPU usage increase an application's energy efficiency?

The investigation in this paper evaluates the power consumption for each of the three operations, 1) network, 2) file I/O and 3) CPU, in isolation to understand their cost and optimization benefits. Other resources that use power such as Bluetooth connectivity, user interface drawing (GPU), Global Positioning System

(GPS) and accelerometer are not considered because they are not within the scope of features (components) developed by my team. Similarly, device settings such as display brightness are not evaluated because they are user controlled settings that cannot be optimized during application development.

The next chapter (chapter two) covers the testing methods for power consumption on iOS while chapter three reviews the test application used to investigate the different test methods, and energy usage. The experimental results are provided in chapter four, and the recommendations are presented in chapter five. Finally, the limitations, future works and extensibility are discussed in chapter six with the conclusion following in chapter seven. Additional materials are included in appendix A and B.

## CHAPTER 2. TEST METHODS FOR BATTERY USAGE

Testing is an important aspect of software development as it ensures the quality bar is met and yields the necessary confidence to deliver the product to the hands of the customers.

In recent years, the technology industry has seen a shift in consumer demand from the traditional PCs to mobile devices. The main reason is that handheld devices are becoming more powerful, for example with a smart phone or tablet today, individuals can browse the web, collaborate via email, instant messenger and video conferencing.

The trend towards mobile devices has underpinned the success of the iPhone and iPad devices. The key to the success has been the iOS platform that has a great developer community with a wide range of applications. Applications are the driving force that has encouraged mobility because consumers understand that they can complete different tasks while on the go given they have downloaded the right apps.

However, the overall quality of the experience in a mobile landscape is a blend of good hardware and software engineering. This means that the hardware should be designed to be portable – lightweight while not compromising on the features users have come to expect: high resolution displays, cameras, extended battery life, faster processors, and multi-touch sensors. On the software side, it is all about design, functionality, performance, usability, and other software quality attributes that also make a great desktop application. Unlike a traditional PC that is always connected to a power source, mobile devices rely on battery power. From a software development standpoint, a key performance metric to monitor and validate is an application's battery life; the core of a great mobile app is good energy consumption. How long can a customer use the app before the battery drains out?

To determine the quality of software, software developers need to have a clear understanding of what the software is supposed to do. Although in some cases

the term 'quality' may be relative and viewed from different perspectives, Nell Dale and David Teague, the writers of C++ Data Structures, describe quality software as one that runs correctly, can be read and understood, evolves easily and is developed efficiently under the constraints of time and money (Dale and Teague 2001, 4).

Quality software must work correctly. 'Works correctly' implies that the software operates and performs its functions as expected. Any unexpected behavior is a software bug. A software bug as defined by Ron Patton, the author of Software Testing, occurs when the software behaves abnormally (does not adhere to the requirement document) (Patton 2001, 15). Table 1 presents a summary of three infamous software bugs as described by Patton (Patton 2001, 10).

Table 1. Infamous software bugs

| **Date** | **Software bug** | **Bug summary** |
|---|---|---|
| **1994-1995** | Disney's Lion King | The Lion King Animated Storybook failed to work on different common PC models. Disney was not aware of the bug until the software failed in the customers' hands. |
| **1994** | Intel Pentium Floating-Point Division Bug | The Intel processor worked fine except with large numbers. The operation: [(4195835 / 3145727) * 3145727 – 4195835] should produce zero but their Pentium processor had a floating-point division bug that produced results other than zero. |
| **1974** | The Y2K (Year 2000) Bug | In 1974, a programmer used 2-digits format to represent the year to save memory space and it worked fine until the year 2000. Billions of dollars were spent to fix the bug to avoid software failures when the year turned to 00 (for 2000). |

Data source: (Patton 2001, 10)

As seen in table 1, quality software needs to work correctly. Erroneous software is the main source of frustration and dissatisfaction for end users. Thus, the success of a software product is based mainly on its quality rather than demand.

And as software development shifts to mobile platforms such as iOS, battery life becomes a key measure for software quality. Along with this wave comes a new class of software bugs related to battery life. Hervé Guihot says it best in his book "Pro Android Apps Performance Optimization" that "Users typically won't notice if your application preserves battery life. However, they most likely will notice if it does not." (Guihot 2012, 206). This is also reflected on the web where a simple search returns a large collection of results related to customer complaints on batteries draining too fast, including the following two examples:

1. *"My iphone 4 battery often drains too quickly, and I wonder if it has something to do with my Mail App…"* posted by user on the Apple support communities (Apple Support Communities 2010).

2. *"…A possible culprit is the operating system's new multitasking feature that allows certain apps and programs to be run simultaneously, such as a music player and an Internet browser…"* reported by technewsdaily (Technewsdaily 2010).

There are many similar reports on blogs and articles online that highlight the fact that the iOS platform has been dogged by battery consumption issues (bugs).

As a result, there is a new category for mobile application testing to measure how quickly the battery drains (Myers, Sandler and Badgett 2011, 221). To effectively test power consumption, we need to develop a clear understanding of the methods available to accurately measure battery life for an application. There are several ways to measure battery consumption on the iPhone and iPad. Apple provides two of these methods; another is to use an external tool such as the power

monitor; or finally an exhaustive method (measuring how long it takes for the battery to drain from a full charge to empty). The energy usage cannot be measured while the device (iPad or iPhone) is connected to the Mac or PC since the battery is being charged when it is connected to the USB port.

The remainder of this chapter reviews the benefits and limitations of each of the following testing methods:

1. Apple's options to power consumption measurement:
    a. The Energy Diagnostic Instruments
    b. iOS Software Development Kit (SDK)
2. Power monitor by MonSoon Solutions Inc.
3. Exhaustive method to get actual battery life

### *Apple's Options to Power Consumption Measurement*

Energy Diagnostic Instruments

Instruments is an application packaged in the Xcode development toolset to provide performance debugging and monitoring capabilities on the desktop as well as iOS. One of the instruments included is the Energy Diagnostic instrument which tracks energy usage over time that is visualized in a timeline format (Anderson 2009, 437).

There are a few steps needed to enable energy measurement on iOS using the energy diagnostics instrument as described by Apple's documentation (Apple Instruments Documentation 2012). To gather power consumption measurements, the "power logging" setting under the device's developer section needs to be turned on while the device is connected to the Mac with Instruments running as pictured in figure 1. After this step, the device can be disconnected to perform the test scenario.

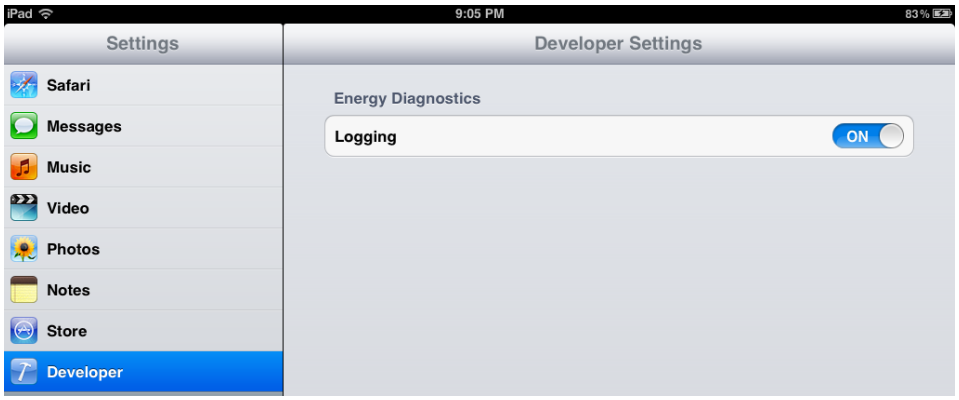


Figure 1. An illustration of turning on power logging

When the test is done, the device is connected back to the computer to collect the energy usage logs through the "Import Energy Diagnostics from device" option in Instruments. Once the logs are loaded, the energy usage for difference resources can be viewed in Instruments. Following are the advantages and disadvantages of Instruments.

Advantages of this approach:

- Instruments is provided by Apple as part of its development tools and continues to improve over time.
- The energy usage logs can be saved as trace files that can be viewed in Instruments at a later time.
- No custom code is required to capture the energy usage logs.
- It is good for debugging as applications can be profiled using Xcode.
- Instruments provides useful information about the device state, and individual components' power states.

Drawbacks of this approach:

- Measuring the energy consumption is a manual process. The iOS device must be disconnected from the Mac, this means that JavaScript UIAutomation that relies

on the device being connected to the Mac cannot be used to simulate user actions for battery testing.

- The accuracy is +/- 5%, and the sampling rate is no quicker than every second.
- The energy usage logs are only viewable in Instruments and do not export to other data formats such as comma-separated values (CSV) files.

iOS SDK

Another way to measure battery usage is through APIs included in the iOS SDK since iOS 3.0. It provides a set of APIs to retrieve the percentage of power remaining on the device. The battery level ranges from 0.0 to 1.0. For example, 0.70 means approximately 70% of the battery is left. The code snippet to illustrate how to obtain the remaining battery level with the API is as follows:

```
float batteryLevel = [[UIDevice currentDevice] batteryLevel];
```

Advantages of this approach:

- It is easy to incorporate in code in one line of Objective-C. To measure energy usage, the battery level can be captured before and after running a particular scenario under test.
- The coding simplicity provides a good option for automations to create regression suite to monitor battery usage.

Drawbacks of this approach:

- For automation, the test code has to be compiled alongside the product source code. Removing test code out of the product source code towards the end of the development cycle adds risk and may require additional testing time to validate the changes.

- The accuracy level reported by the API is within +/- 5%. For example, if the actual battery level is 77%, the API reports 75%, while 78% is reported as 80%.

### *The Power Monitor*

The power monitor is a blend of a hardware and software tool that together supply power and the ability to measure the energy usage of any device that uses a single lithium (Li) battery (Monsoon Solutions Inc. 2011). The power monitor, with mockup battery on the iPhone 4, can be used to provide power and capture energy usage. The energy usage measurements include current (I), voltage (V), and power (W).

The Power Monitor Setup

To use the power monitor with an iOS device such as the iPhone 4, the device needs to be modified to bypass the battery to rely on the power monitor as the energy source.

With a modified test device in place, the rest of the steps are simple – 1) connect the power monitor to the power outlet, 2) connect the power monitor to a Windows machine via USB and 3) install and run the power monitor software, PowerTool, to control the behavior of the hardware such as start and stop the power supply to the iOS device. Figures 2 through 4 show the modified iPhone setup for use with the power monitor. In addition, the Vacuum Base PanaVise Combo model 381 is used as a stand to hold the iPhone 4 during test execution to help keep the phone steady for accurate measurement (PanaVise Products, Inc. n.d.). The iPhone has an accelerometer that is sensitivity to gravity so moving the phone may drain additional power due to this feature.


Figure 2. The back of the iPhone 4 with a replacement cover

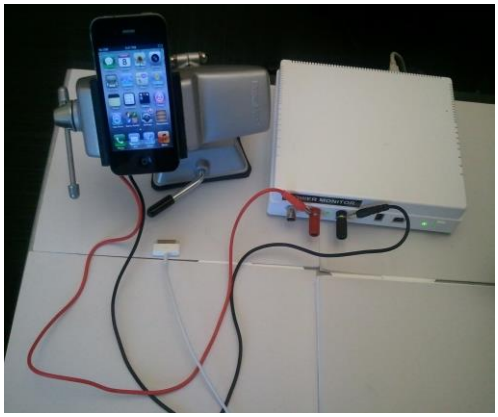

Figure 3. The iPhone is on with power supplied by the power monitor

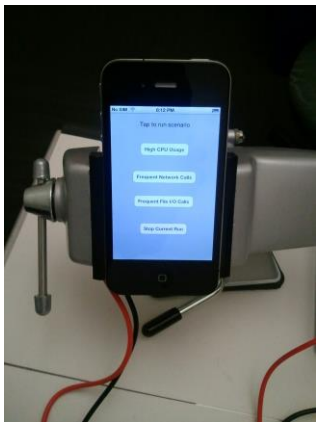

Figure 4. The test app running on the iPhone powered by the power monitor

The PowerTool software is easy to use. With the power monitor hardware switched on, clicking the 'Vout Enable' button allows power to flow through the connected circuitry to the iPhone 4.

The power monitor software is feature rich and provides start and stop a capture of energy usage, and save results to a file that can be exported to different formats such as comma-separated values (CSV). There are also options to copy the graph, statistics and snapshots of the result window. The statistics include run duration, number of samples taken, total energy consumed, averages for power, current and voltage used. And the most important of the statistics in the capture is projected battery life. A screenshot of the software in action is presented in figure 5.

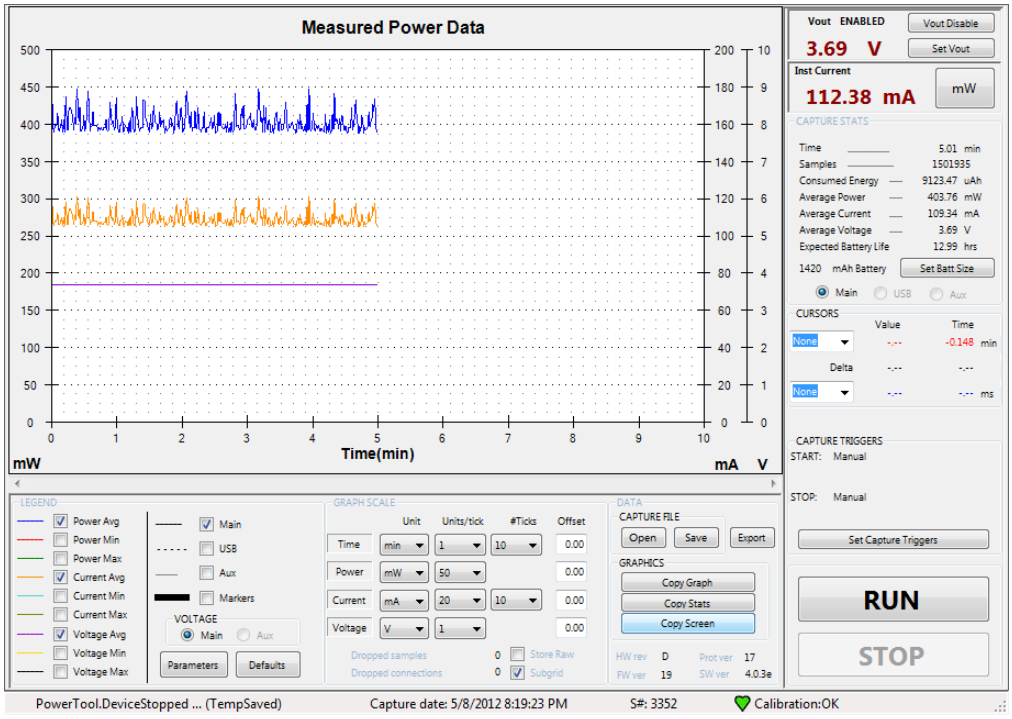


Figure 5. The PowerTool showing a capture of the test app in idle state

Advantages of this approach:

- The power monitor collects energy usage information including the measurements for the current, voltage and power used during a period execution.

- It has the capability to export the data to CSV formats that can be read by other applications such as Excel.

- It captures consumed energy, average voltage (volts); average power (watts); average current (mA); and expected battery life for a test scenario.

It has the capability to set voltage range and energy range of a particular battery. This is useful when testing against phones with different energy makeup.

Drawbacks of this approach:

- Measuring energy usage with the power monitor is a manual process and not readily automatable without additional investments in the infrastructure.
- It requires a Windows based machine while iOS development is Mac based.
- It is complex and costly to get a mockup battery working in an iPhone 4.

### *The Exhaustive Method to Get Actual Battery Life*

The exhaustive method is similar to stress testing. This approach measures the time it takes to completely drain a fully charged device that is running the application under test. To simulate continuous usage, the display is left on and the auto-lock feature is turned off. The resulting duration is then compared to the baseline idle scenario where no user application is running.

Advantages of this approach:

- Measurements represent actual and perceived battery life for the application under test.
- It provides stress testing coverage while measuring battery life.
- The exhaustive method does not require the use of custom tools.

Drawbacks of this approach:

- It takes a long time to obtain results. For each test execution, the battery needs to be drained from full charge (100%) to empty (0%).

- The test execution duration is unpredictable, so it is difficult to plan and schedule battery testing with this approach because the rate of power drain from 100% to 0% greatly depends on the actual test.
- Accuracy is lost after a certain period of testing as the battery on the device loses capacity over time. Results from a new device may differ from those obtained using an old device that has undergone many recharge cycles.

## CHAPTER 3. THE TEST APP

The test app, shown in figure 6, was created to investigate the impact of CPU usage, network and file I/O on battery life. The simple user interface design was chosen to minimize resource usage and its impact to power consumption during idle periods. We wanted the idle state to closely resemble the idle energy usage in the iOS home screen with no user application running. Each of the buttons, when pressed, triggers a pre-defined test scenario.

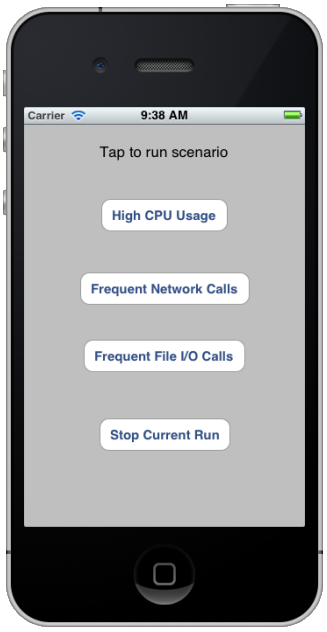


Figure 6. The test app user interface

To validate the effectiveness of the design, we compared the test app's energy usage on idle against the baseline usage from the iOS home screen. The results in table 2 show that the interface design for the test app does not consume more energy when idle than the baseline (iOS home screen).

Table 2. The energy usage of the test app (in uAh) compared to the home screen for the iPhone 4 running iOS 5.0

| Application state | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average |
|---|---|---|---|---|---|---|
| Home Screen - Idle | 15.02 | 14.76 | 15.43 | 15.69 | 14.69 | 15.12 |
| Test App - Idle | 11.65 | 14.82 | 14.78 | 14.92 | 14.65 | 14.16 |

With a baseline understanding of the test app's idle state, we can focus the measurement on the scenarios that trigger different level of energy usage by stressing the CPU, network and file I/O individually as follows:

### *Investigation on High CPU Usage*

For the high CPU usage scenario, the goal was to simulate a CPU intensive job to validate its impact to battery life. I investigated three recursive mathematical methods that rely heavily on the CPU for the computation:

Fibonacci Series

Fibonacci series is 0, 1, 2, 3, 5, 8, ... where each subsequent number is the sum of the two preceding terms. It is solved recursively as (Lipschutz and Lipson 2007, 54):

- if (n == 0 || n == 1) { return n; }
- if (n > 1) { return Fibonacci(n – 2) + Fibonacci(n – 1); }

Factorial Function

The factorial is the product of positive integers starting from 1 to n, denoted as n!, where 0! = 1. For example, the factorial of the number 5 is 120 (or 5 * 4 * 3 * 2 * 1). This is solved recursively as follows (Lipschutz and Lipson 2007, 53):

- if (n == 0) { return 1; }
- if (n > 0) { return n * factorial(n – 1); }

Ackermann Function

The Ackermann function takes two non-negative integer arguments, 0, 1, 2...n and it is recursively defined as (Lipschutz and Lipson 2007, 54):

- if(m == 0) { return n + 1; }
- if(m != 0 && n = 0) { return Ackermann(m – 1, 1); }
- if(m != 0 && n != 0) { return Ackermann(m – 1, Ackermann(m, n – 1)); }

### *Function Selection for High CPU Usage*

After evaluating possible implementations for the three functions, the factorial calculation was chosen because of the simplicity of the algorithm; we want a CPU intensive job and the recursive-ness of the factorial operations spike the CPU consistently without a complex implementation. For the high CPU test, the application spawns off 100 threads to compute multiple factorials simultaneously. I did not choose the Ackermann function for the high CPU implementation because of its complexity and non-trivial verification requirements. The validation of a complex solution might have required additional time to debug and fix, thus taking time away from battery testing. Similarly, the Fibonacci function was considered but the factorial function was preferred as it is easy to validate by checking the output against scientific calculators that readily include this function.

### *Frequent Network Operations*

To exercise frequent network operations, the test application pings an external server in a continuous loop at one second intervals. The ping operations are over the 3G or Wi-Fi radio, depending on the connection settings for the device during the test execution.

### *Frequent File I/O Calls*

The file I/O scenario will perform read and write operations every millisecond in a continuous loop until the stop button is pressed.

### *Stop Current Run*

Tapping this button stops the currently running operation. Each operation is run repeatedly in a continuous loop to simulate a period of activity, so this will trigger the exit flag to end the run.

The source code for the test application is provided in the appendix section. The next chapter discusses the experiments and test results.

## CHAPTER 4. EXPERIMENTS AND TEST RESULTS

This section covers the experimental results from testing a set of applications to examine their power consumption using the test methods discussed in chapter 2. The experiments focus on using the Energy Diagnostics instrument, the power monitor and the exhaustive approach. The iOS SDK method is not covered because it requires modifying an application's source code and this is not possible for third party applications that are available in the App store.

To conduct the experiment, a consistent test environment was created by using the same hardware and software settings. The devices used were an iPhone 4 and an iPad 2 both running iOS version 5.0.0. In addition, the display brightness level was set a medium (50%) with auto brightness turned off. Finally, all running applications and unused resources such as Bluetooth and locations services were turned off to ensure that only the application under test was consuming energy.

### *Results from Energy Diagnostics Instrument*

The high CPU usage test was used to illustrate the Energy Diagnostics Instrument and the results from the run are provided in figures 7 to 14. The energy usage level is a fraction over 20 at a particular time interval as seen on the timeline for the device. For example, 0/20 indicates the device is connected to a power source while 14/20 represents 70% of energy usage.

There is no network activity reported in figure 9 as the test app was running the high CPU usage scenario which does not perform network operations. The display brightness provides information on the brightness level, the brighter the screen the more energy consumed. Figure 10 shows that the test device's display brightness level was 50%.

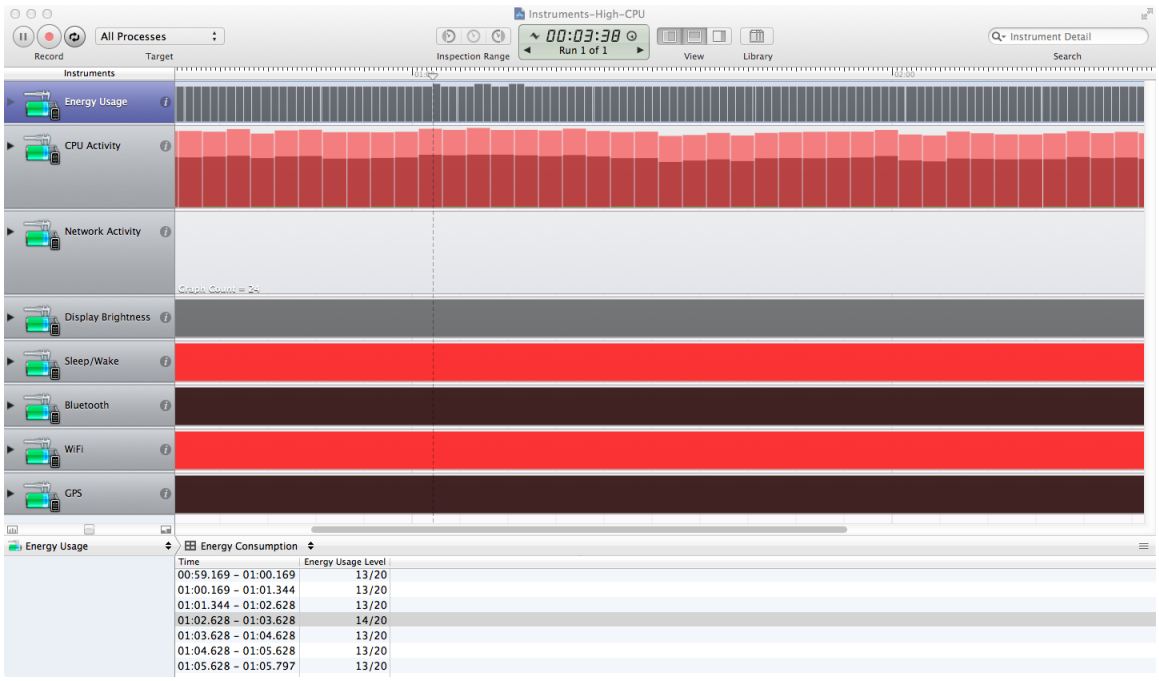


Figure 7. The energy diagnostics trace with the focus set on energy usage

CPU Activity

CPU Activity

| Time | Total Activity | Foreground App Activity | Audio Processing | Graphics |
|---|---|---|---|---|
| 00:51.922 – 00:54.850 | 83.2% | 128.4% | 0% | 0.1% |
| 00:54.850 – 00:57.850 | 84.2% | 128.1% | 0% | 1.1% |
| 00:57.850 – 01:00.848 | 85.1% | 129% | 0% | 1.1% |
| 01:00.848 – 01:03.847 | 89.7% | 134.1% | 0% | 0.2% |
| 01:03.847 – 01:06.850 | 88.7% | 132.2% | 0% | 1.1% |
| 01:06.850 – 01:09.847 | 89.8% | 135.5% | 0% | 0.1% |
| 01:09.847 – 01:12.848 | 89.8% | 132.1% | 0% | 1.1% |

Figure 8. A view of the CPU activity instrument

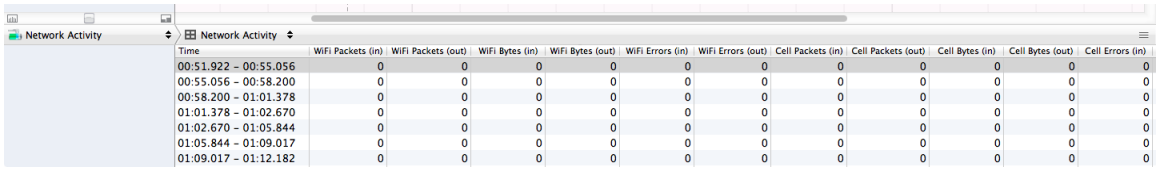


Figure 9. A view of the network activity instrument

Display Brightness

Brightness

| Time | Brightness Level |
|---|---|
| 00:00 – 03:39.141 | 50% |
| 03:39.141 | 50% |

Figure 10. A view of the display brightness instrument

The Sleep/Wake reports the state of the device (whether the screen is on or off). During the test run, the display did not go sleep and the state is reported as 'running' which is illustrated in figure 11.

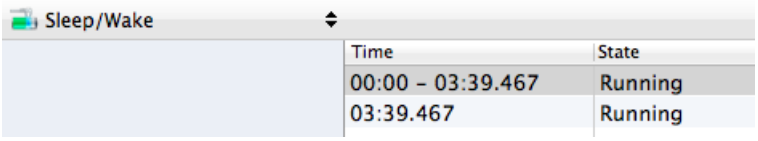


Figure 11. A view of the sleep/wake instrument

The Bluetooth instrument reports the power state (on or off) of the radio. As pictured in figure 12, Bluetooth on the device was turned off.

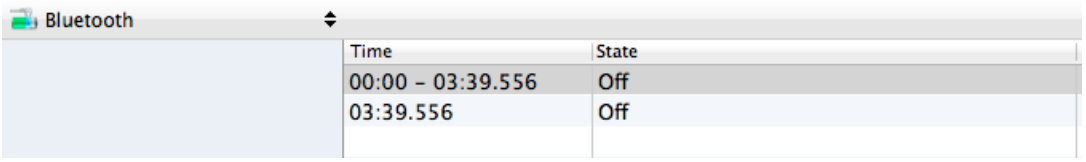


Figure 12. A view of the Bluetooth instrument

The Wi-Fi instrument reports the power state (on or off) of the radio. At the time of the capture, the Wi-Fi was turned on as seen in figure 13.

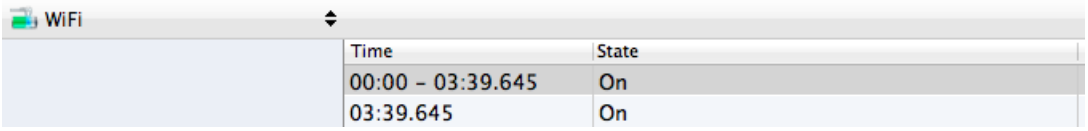


Figure 13. A view of the Wi-Fi instrument

Finally, the GPS instrument in figure 14 reports the power state (on or off) of the service. At the time of the capture, the GPS was turned off.

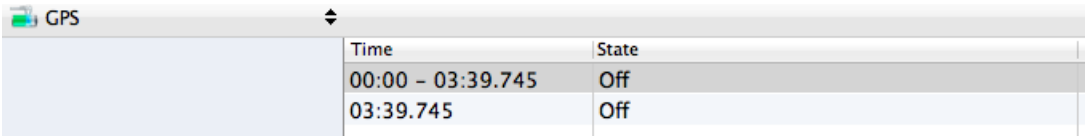


Figure 14. A view of the GPS instrument

### *Results from the Power Monitor*

Several tests were conducted with the power monitor using the test app and a select set of free popular apps from the App store. Two tests were performed with each app to compare the results between its idle state and active state. The exception is the iOS home screen which is used as the benchmark and only its idle state is captured. The apps and scenarios tested are provided in table 3 and their results are included in tables 4 through 6.

Table 3. The set of apps and scenarios tested

| Application | Idle state | Active state |
|---|---|---|
| Facebook | Measured while the app was on the sign-in screen before the user is authenticated. | Measured while loading/browsing a user's profile that was not previously accessed or cached. |
| Flashlight | Measured while the flashlight was off | Measured while the flashlight was turned on |
| Flixster | Measured while the app was on box office tab with everything loaded. | Measured while loading/browsing movie details that were not previously accessed or cached. |
| Skype | Measured while the app was on the sign-in screen before the user is authenticated. | Measured while a Skype video chat was in session. |
| Test App | Measured while the app was not running any test scenarios. | Measured while a test scenario was running each of the following three scenarios.<br>1) High CPU - 100 threads repeatedly compute the factorial of 64, spiking the CPU consistently.<br>2) Frequent Network Calls - Pinging external server in an infinite loop keeping the network radio on high power state.<br>3) Frequent File I/O – Repeatedly perform read/write to a local file. |

### *Key Observations*

The results from table 4 demonstrate the impact that applications running on the iPhone 4 have to its battery life. It is not a surprise that the Flashlight app when active is the most power hungry application from the set of third party applications

tested. Keeping the flash light up drains the battery quickly and users almost readily expect it to. The unexpected result was the low estimated battery life for the Flashlight app while idle with the light turned off. One observation is that the app does not stop doing work while there is no user interaction with the device. Instead, the app is continuously refreshing the view with new online advertisements which result in network transactions that keep the radio on full power state, continuous CPU usage processing the web service request and response, and redrawing keeps the GPU active.

Table 4. The estimated battery life (hrs) – iPhone 4 running iOS 5.0

| Application state | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average |
|---|---|---|---|---|---|---|
| Test App – Idle (Wi-Fi) | 22.18 | 19.66 | 17.92 | 17.04 | 18.06 | 18.97 |
| Skype – Idle (Wi-Fi) | 16.70 | 17.64 | 21.69 | 16.51 | 16.86 | 17.88 |
| Home Screen – Idle (Wi-Fi) | 17.15 | 16.79 | 16.91 | 17.28 | 17.75 | 17.18 |
| Flixster – Idle (Wi-Fi) | 16.31 | 12.57 | 14.97 | 16.67 | 16.28 | 15.36 |
| Test App - File IO (no network) | 10.40 | 36.08 | 9.44 | 9.29 | 9.74 | 14.99 |
| Facebook – Idle (Wi-Fi) | 13.96 | 15.06 | 15.07 | 11.88 | 17.76 | 14.75 |
| SkyBurger – Idle (Wi-Fi) | 13.76 | 13.15 | 13.11 | 14.06 | 13.59 | 13.53 |
| SkyBurger – Active (Wi-Fi) | 10.73 | 8.84 | 9.56 | 10.09 | 10.17 | 9.88 |
| Test App - Network Calls (Wi-Fi) | 9.27 | 9.58 | 8.97 | 9.03 | 9.01 | 9.17 |
| Facebook – Active (Wi-Fi) | 14.42 | 5.72 | 10.06 | 7.72 | 6.25 | 8.83 |
| Flixster – Active (Wi-Fi) | 6.70 | 7.31 | 11.01 | 8.96 | 8.32 | 8.46 |
| Test App - Network Calls (3G) | 7.61 | 7.58 | 7.61 | 7.59 | 7.59 | 7.60 |
| Test App - High CPU (no network) | 7.31 | 6.07 | 7.48 | 6.20 | 8.34 | 7.08 |
| Flashlight – Idle (Wi-Fi) | 18.74 | 5.25 | 4.69 | 7.89 | 5.08 | 5.64 |
| Skype – Active (Wi-Fi) | 4.64 | 4.66 | 4.57 | 4.49 | 5.50 | 4.77 |
| Flashlight – Active (Wi-Fi) | 3.22 | 3.43 | 3.47 | 3.72 | 3.91 | 3.55 |
| Flashlight – Active (3G) | 2.15 | 2.18 | 2.16 | 2.15 | 2.13 | 2.15 |

The next scenario with the lowest estimated battery life is the Skype video chat at 4.77 hours. In this scenario, the load generated by the live multimedia stream impacts battery through the continuous usage of CPU, view redraws, and network activity. Furthermore, Skype uses voice over IP (VoIP) which relies on the multitasking feature in iOS which can drain the battery quickly (Vo 2011, 261). Compared to its idle state (with an estimated battery life of 17.88 hours), we can assert the impact of doing work to battery life and the benefits yielded by allowing the application to idle whenever possible to extend battery life. However, without having access to the source code, we can only come to a black box conclusion about what the background work the Skype app may be doing that is contributing to its high energy consumption.

Table 5 shows the estimated projected battery life for each of the test app scenarios. Reviewing the results from the table, we can conclude that CPU usage is a major factor in energy usage ahead of 3G. The variation in the CPU test is due to the way the scenario is implemented – to compute the factorial of 64 on multiple threads, the calculations starts a recursive chain that eventually unwinds to return the results. As each thread completes and returns (exits), the load on the CPU temporarily decreases until new threads are spanned and the computation loop begins a new iteration. Depending on the timing of the sampling to get the estimated battery life, the values could be from the peak of the operations or the end/start of new a loop. This assessment also explains the degree of consistency in results for the 3G scenario as the implementation of the network operation is a web service call that occurs every second which keeps the radio at its full power state. The radio power on the device stays on full power for up to 10 seconds after each network transaction (Apple Inc. 2011). This is one reason Apple recommends bursting network transactions, that means to send and/or receive all at once with big

transactions, and longer silent periods in between to avoid multiple network calls too often (Apple Inc. 2011).

Table 5. The test app estimated battery life (hrs) – iPhone 4 running iOS 5.0

| Application state | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average |
|---|---|---|---|---|---|---|
| Idle (network off) | 22.18 | 19.66 | 17.92 | 17.04 | 18.06 | 18.97 |
| File IO (network off) | 14.83 | 14.49 | 14.65 | 14.71 | 14.71 | 14.65 |
| Network Calls (Wi-Fi) | 9.27 | 9.58 | 8.97 | 9.03 | 9.01 | 9.17 |
| Network Calls (3G) | 7.61 | 7.58 | 7.61 | 7.59 | 7.59 | 7.60 |
| High CPU (network off) | 7.31 | 6.07 | 7.48 | 6.20 | 8.34 | 7.08 |

The results from the iPad 2, shown in table 6, are similar to those obtained in the iPhone 4 testing. The exhaustive approach was used to capture the battery life of each of the test app scenarios. Each execution started with a fully charged iPad and the run completed when the battery was completely drained. The tests were limited to the test app because we cannot automate third party applications to run scenarios repeatedly due to the sandbox restriction for applications on iOS.

Table 6. The actual battery life on iPad 2 running iOS 5.0

| Test app scenario | Test description | Battery life (hrs) |
|---|---|---|
| Idle-display On | Idle benchmark app in foreground, display never sleeps. (Minimal occasional activity to log current timestamp) | 11.50 |
| High Wi-Fi Usage | Pinging external server in an infinite loop with 1 second sleeps between iterations | 10.67 |
| High 3G Usage | Pinging external server in an infinite loop with 1 second sleeps between iterations | 10.17 |
| High CPU Usage | 100 threads repeatedly compute the factorial of 64 (i.e. 64!) | 7.00 |

## CHAPTER 5. RECOMMENDATIONS

Based on the investigation and experimental results, we can conclude that applications that heavily use the network and the CPU consume more power. However, connectivity is important for mobile applications as they can rely on the server horse power to do the bulk of the processing work and for storage since these mobile devices come with limited memory space. In a way, network cost is necessary to support some of the functionality provided by mobile applications. As a result, based on the results from the tests, the recommendation is to optimize CPU usage and focus on extending idle to maximize battery life.

Idle is the period the application is not processing any user input (user interactions). On the desktop, applications typically take advantage of the idle (inactive) period to perform housekeeping tasks in the background. However, this design is not ideal for mobile as it keeps the system busy and continues to drain power while the user is not interacting with the application.

To optimize and test for idleness, focus on CPU usage to monitor activity. The benefit of monitoring CPU usage is that it is a catchall bucket. Lower CPU usage also lowers battery consumption (Sadun 2008, 29). If there is any activity, network, file I/O or UI redraws, it will be reflected in the CPU usage either as spikes or consistent non-zero usage value. As a result, my recommendation to the team was to validate CPU with a target guidance of 0% on idle and 1% in the worst case.

To help the team achieve this goal, I created an overlay instrument that displayed the CPU usage in real time over the UI in a non-intrusive way for debug (internal) builds of the application. This implementation was instrumental in catching bugs; the team and the dogfood community did not have to use extra tools, instead they simply relied on the real time CPU numbers to determine whether the application adhered to idle or not, and in most cases, we got reports of CPU spikes and constant high CPU usage that we would have otherwise missed.

In terms of tools for a development team, the power monitor is a good all-around tool for measuring power consumption for apps under development as well as comparing results against existing third party applications from the App store. Instruments on the other hand, is a good addition particularly for debugging your own application for which you have access to application source code.

## CHAPTER 6. LIMITATION, FUTURE WORKS AND EXTENSIBILITY

The major limiting factor for power consumption testing is that it is a manual process and not readily automatable. This is an area for future investment to implement test hooks that may be accessible for automation purposes. This can be done through custom automation libraries built internally, or from third parties. Future versions of the iOS SDK may provide automation support for battery testing but in the meantime, this is largely a manual process.

In addition, combinational testing can help us answer questions about the impact of real world scenarios where an application may be using CPU, file I/O and network over 3G or Wi-Fi simultaneously. Another consideration worth investigating is whether differences in the CPU usage level affects how much battery power is drained. Potentially, this could lead to optimizations related to managing the level of load on the CPU to within an optimal range.

Furthermore, investigating additional areas related to GPS, Bluetooth, GPU and accelerometer for their impact to battery usage may be beneficial for developers that rely on these resources in their applications. The GPU is particularly interesting for UI intensive applications such as games or data visualization apps while the GPS has its benefits with location services that bring a new dimension to mobile applications.

For extendibility, the idle optimization and testing strategy was successful for power management verification and in the process, the performance bugs related to the high CPU usage were also identified and corrected.

The idle CPU optimization approach is readily extensible to other mobile platforms such as Android, Symbian, Windows phone and Blackberry. In fact, in my workgroup teams developing for Android and Windows Phone did leverage the idle test approach in addition to the tools and test methods discussed in chapter two with the exception of Instruments which is platform specific to Apple. The power monitor,

once setup, is practical to use and provides projected battery life for applications under test. The projected battery life metric is helpful when communicating with management as it is a number that they can relate to and customers understand. In addition, the test methods are OS version agnostic, for example in the case of the iPhone, when future iOS upgrades are released the team can continue to seamlessly use the same verification methods.

## CHAPTER 7. CONCLUSION

During the verification and validation process, the software is put through its paces. The goal is to strive to break the software and find as many bugs as possible. The thought of discovering "as many bugs as possible" rather than all the bugs in the software highlights the challenge of testing software. Phillip G. Armour, in his article, The Unconscious Art of Software Testing, describes testing as the search for the unknown that one does not know of (Armour 2005). A bug does not exist until it is discovered through extensive testing.

Despite the difficulties faced in testing, finding bugs is not merely a coincidence or an accident. Software testers have to devise "heuristic strategies" to find the unknown (the errors in the software including power consumption issues on mobile devices) (Armour 2005).

To emphasize the role and importance of software verification and validation in quality software development, table 7 reviews the infamous software bugs presented earlier in table 1, Infamous Software Bugs. Software testing as highlighted in table 7 is a critical step in software development and cannot be overlooked, and on the iOS platform that includes a new aspect related to power consumption.

This project paper provided the validation methods available for battery testing and optimization recommendations for iOS applications. By focusing on idle CPU optimization, developers can build applications that are energy efficient and have the right test strategy to monitor power consumption on iOS. This approach aligns with Apple developers' recommendation that battery life is rooted on sleep – the key optimization is to extend the stand by time of the device (Apple Inc. 2011). Following this approach to power consumption on iOS can help avert the next infamous bugs being due to an oversight in battery testing.

Table 7. A tester's view on the infamous bugs

| **Software bug** | **Testers' views** |
|---|---|
| Disney's Lion King, 1994-95 | The Lion King Animated Storybook failed because Disney did not test the software on different PC models except the ones that its programmers used in development. Had Disney extensively used the verification and validation stage, the bug would have been discovered and fixed early and fixed before being sold to customers. |
| Intel Pentium Floating-Point Division Bug, 1994 | Intel had discovered the bug in its Pentium chip but decided not to fix it because the probability of the error was remote. One of the goals of a software tester is to ensure that bugs get fixed; failing to meet this goal did cost Intel money and its reputation was dented because the bug occurred more frequently than Intel had anticipated. |
| The Y2K (Year 2000) Bug, 1974 | In the Y2K bug, the verification and validation process was omitted altogether. The programmer made a decision to use 2-digits format to represent the year to save memory space. He briefly thought about the year 2000 but maintained the 2-digit format because he felt that his software would be useless by the turn of the century. Had verification and validation been used, the software testers would have tried to answer the "what if the year was 2000?" question and perhaps found a fix early. |

Data Source: (Patton 2001)

## REFERENCES


Anderson, Fritz. 2009. *Xcode 3 Unleashed.* Sams.

Apple Inc. 2011. "Apple's World Wide Developers Conference." *iOS Performance Presentation.*

2012. *Apple Instruments Documentation.* https://developer.apple.com/library/ios/#documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Built-InInstruments/Built-InInstruments.html#//apple_ref/doc/uid/TP40004652-CH6-SW63.

2010. *Apple Support Communities.* https://discussions.apple.com/thread/2607376?start=0&tstart=0.

Armour, Phillip G. 2005. "The Unconscious Art of Software Testing." (Communications of the ACM 48) 15-18.

Dale, Nell, and David Teague. 2001. *C++ Plus Data Structures.* Massachusetts: Jones and Barlett Publisher.

Guihot, Hervé. 2012. *Pro Android Apps Performance Optimization.* Apress.

Lipschutz, Seymour, and Marc Lipson. 2007. *Discrete Mathematics.* Schaum's outlines.

Monsoon Solutions Inc. 2011. *The Power Monitor.* http://www.msoon.com/LabEquipment/PowerMonitor/.

Myers, Glenford J., Corey Sandler , and Tom Badgett. 2011. *The Art of Software Testing.* 3rd. Wiley.

PanaVise Products, Inc. n.d. *PanaVise Combo model 381.* Accessed 2011. http://www.panavise.com/index.html?pageID=1&page=full&--eqskudatarq=6.

Patton, Ron. 2001. *Software Testing.* Sams Publishing.

Reddy, Martin. 2011. *API design for C++.* Massachusetts: Morgan Kaufmann.

Sadun, Erica. 2008. *Taking Your iPhone to the Max.* Apress.

2010. *Technewsdaily.* http://www.technewsdaily.com/696-iphone-battery-draining-fast-with-ios4-power-vampire.html.

Vo, Khang. 2011. *Pro iOS Apps Performance Optimization.* Apress.

## APPENDIX A

```
//  BatterySampleApp – code snippets
//  Created by Yannick Kwete on 4/18/12.
//  Copyright (c) 2012 __MyCompanyName__. All rights reserved.
#pragma mark --- Test Action Section ---
static bool stopRun = false;
/* This function computes the factorial of the given number
* n! = n * (n - 1)!
* Note: 0! = 1, 1! = 1, 2! = 2 * 1 = 2, 3! = 3 * 2 *1 = 6, etc...
* So:
* a) if n = 0, then n! = 1
* b) if n > 0, then n! = n * (n - 1)!
*/
unsigned long long factorial(unsigned long long number)
{
static unsigned long long counter = 0; // for debugging
if (number == 0)
{
return 1;
}
else if (number > 0)
{
unsigned long long result = number * factorial(number - 1);
counter++;
NSLog(@"\nCounter = %llu", counter);
NSLog(@"\nCurrent Result = %llu", result);
```

```
return result;
}
else
{
printf("\nError - Cannot compute factorial for negative integer!");
return -1;      // error code
}
}
void doSomeWork()
{
// perform some calculations
while (!stopRun)
{
printf("\nComputing the factorial 64 (i.e. 64!)");
unsigned long long result = factorial(64); //spike the CPU
NSLog(@"\nResult = %llu", result);
}
}
void runFactorialTest()
{
printf("Thread running");
const int NUM_THREADS = 100;
pthread_t threads[NUM_THREADS];
int status = 0;
float startBatteryLevel = [[UIDevice currentDevice] batteryLevel];
NSFileHandle *runHistoryFile;
NSData *buffer;
```

```
time_t rawtime;
NSString *filePath = [NSTemporaryDirectory()
stringByAppendingPathComponent:@"runHistory.txt"];
// create a new file
NSFileManager *fm = [NSFileManager defaultManager];
if([fm createFileAtPath:filePath contents:nil attributes:nil] == NO)
NSLog(@"Couldn't create a file on disk");
for (int i = 0; i < NUM_THREADS; i++)
{
status = pthread_create(&threads[i], NULL, (void*) &doSomeWork, NULL);
if (status)
{
NSLog(@"Thread creation failed");
}
}
// Run until power runs out
while (!stopRun)
{
/* Update history file with current timestamp so that we can track
* when the battery runs out.
*/
runHistoryFile = [NSFileHandle fileHandleForWritingAtPath:filePath];
if(runHistoryFile == nil)
{
NSLog(@"Open of runHistoryFile for writing failed");
}
[runHistoryFile seekToEndOfFile];
```

```
time ( &rawtime );
const char* currTime = ctime (&rawtime);
buffer = [NSData dataWithBytes: currTime length: strlen(currTime)];
[runHistoryFile writeData:buffer];
[runHistoryFile closeFile];
[buffer release];
[runHistoryFile release];
usleep(60000000); // sleep for one minute before updating runHistory
}
// if stop the run, wait for threads to complete
for (int i = 0; i < NUM_THREADS; i++)
{
pthread_join(threads[i], NULL);
}
[filePath release];
[fm release];
float endBatteryLevel = [[UIDevice currentDevice] batteryLevel];
NSLog(@"Battery level at the beginning was %f and now the current level is %f",
startBatteryLevel, endBatteryLevel);
}
void runFileIOTest()
{
printf("Running File I/O test thread...");
NSFileHandle *runHistoryFile;
NSData *buffer;
time_t rawtime;
```

```
NSString *filePath = [NSTemporaryDirectory()
stringByAppendingPathComponent:@"fileIORunHistory.txt"];
// create a new file
NSFileManager *fm = [NSFileManager defaultManager];
if([fm createFileAtPath:filePath contents:nil attributes:nil] == NO)
NSLog(@"Couldn't create a file on disk");
// Run until power runs out
while (!stopRun)
{
/* Update history file with current timestamp so that we can track
* when the battery runs out.
*/
runHistoryFile = [NSFileHandle fileHandleForWritingAtPath:filePath];
if(runHistoryFile == nil)
{
NSLog(@"Open of runHistoryFile for writing failed");
}
[runHistoryFile seekToEndOfFile];
time ( &rawtime );
const char* currTime = ctime (&rawtime);
buffer = [NSData dataWithBytes: currTime length: strlen(currTime)];
[runHistoryFile writeData:buffer];
[runHistoryFile closeFile];
[buffer release];
[runHistoryFile release];
usleep(1000); // sleep for one milli-sec before updating runHistory
}
```

```
[filePath release];
[fm release];
}
- (IBAction)stopRun:(id)sender
{
NSLog(@"Stopping test run...");
stopRun = true;
}
- (IBAction)runHighCPU:(id)sender
{
NSLog(@"Running High CPU test...");
stopRun = false;
pthread_t testThread;
int result = pthread_create(&testThread, NULL, (void*) &runFactorialTest, NULL);
if (result)
{
NSLog(@"Thread creation failed");
}
}
- (IBAction)runHighFileIO:(id)sender
{
NSLog(@"Running File I/O test...");
stopRun = false;
pthread_t testThread;
int result = pthread_create(&testThread, NULL, (void*) &runFileIOTest, NULL);
if (result)
{
```

```
NSLog(@"Thread creation failed");
}
}
- (void)pingServer
{
NSFileHandle /*inFile,*/ *outFile;
NSData *buffer;
NSURLConnection *connection;
NSString *filePath = [NSTemporaryDirectory()
stringByAppendingPathComponent:@"runNetworkTest.txt"];
// create a new file
NSFileManager *fm = [NSFileManager defaultManager];
if([fm createFileAtPath:filePath contents:nil attributes:nil] == NO)
NSLog(@"Couldn't create a file on disk");
while (stopRun)
{
NSURL *url = [NSURL URLWithString:@"http://www.cplusplus.com/foo"]; // simple
text page returned
// Put that URL into an NSURLRequest
NSURLRequest *req = [NSURLRequest requestWithURL:url];
// Create a connection that will exchange this request for data from the URL
connection = [[NSURLConnection alloc] initWithRequest:req delegate:self
startImmediately:YES];
outFile = [NSFileHandle fileHandleForWritingAtPath:filePath];
if(outFile == nil)
{
NSLog(@"Open of fileB for writing failed");
```

```
}
[outFile seekToEndOfFile];
time_t rawtime;
time ( &rawtime );
printf ( "The current local time is: %s", ctime (&rawtime) );
const char* currTime = ctime (&rawtime);
buffer = [NSData dataWithBytes: currTime length: strlen(currTime)];
[outFile writeData:buffer];
[outFile closeFile];
// space out the iterations by one second
usleep(1000000);
}
}
- (IBAction)runWiFi:(id)sender
{
NSLog(@"Running Network test...");
[NSThread detachNewThreadSelector:@selector(pingServer) toTarget:self
withObject:nil];
}
#pragma mark --- End ---
```

## APPENDIX B

In this section I compiled and documented the set of Apple's guidelines and optimization tips for building power efficient applications after reviewing numerous presentations and videos from the World Wide Developer Conference (WWDC) 2011 and searching through the Apple developer pages.

### *What to Test/Optimize*

There are four key areas for validating power consumption and battery life (Apple Inc. 2011):

1. Reduce network traffic
2. Bursting
3. Sleep/Wake
4. Dynamic frame rates

<u>Reduce Network Traffic</u>

- Better for the customer's data plans
- Efficient energy usage (i.e. battery life)
- Points to keep in mind or think about:
- Number of Network connections to the servers
- Which connections our application is making?
- How much data is transmitted to and from the server connections?
- How is the health and status of these connections? Any retransmission (i.e. retries to complete a transaction and how often)? Stability of the connection affects battery life.
- Average round trip times to complete a transaction
- When is our application using the network?

How do we measure network traffic?

- As part of iOS 5.0 update, we can analyze the points above using the Network Connections Instrument:
- Measure data volume
- Works for TCP/IP and UDP/IP
- Captures the performance metrics for network traffic
- *Remember, if you take care of your network usage, you indirectly improve battery life.*

Optimization suggestions

- Caching content
- Avoid redundant downloads
- Cached data is persistent in iOS 5 – cached data is available even after restarting your application. This feature is on by default.
- Effective use of caching can reduce network traffic tremendously
- Compressing content
- Compress data whenever you can (if using plain text or xml, may be zipping on the server for transmission, etc.)
- Try to use compact data format
- Reduce bigger images
- Using 'Resumable' transfers
- Connection loss is more likely on mobile devices
- Avoid restarting, instead resume the transfer
- Support 'resumable' transfers
- Smarter downloads
- Understand how customers use your app
- Support logging app usage
- Send feedback feature to gather more stats

Bursting

- Send and/or receive all at once (i.e. big transaction, longer silent period, then big transaction…)
- Avoid multiple network calls too often
- Radio power on the device stays on full power for up to 10 seconds after each transaction. (*Exceptions – real time streaming with real time interaction for example.)*

How to measure bursting

1. Use the Energy Diagnostics template in Instruments, it provides statistics on:
2. Energy
3. Samples every second
4. CPU
5. Power states
6. Network activity

Sleep/Wake

- Battery life is rooted on sleep – the key optimization is to extend the stand by time of the device
- Let the device sleep if there is no user interaction on the app
- Avoid unnecessarily waking the device from sleep
- Be carefully with push notifications

How to measure sleep/wake

1. Use the Energy Diagnostics template in Instruments
2. The sleep/wake instrument
3. Dark area means sleep
4. Light area means awake
5. Pay attention to periodic wakes

6. You want to be asleep as long as you can

Dynamic Frame Rates

- Not all scenarios/animations need to be 60 fps
- Adjust to lower fps when possible (if quality isn't affected)
- Avoid redraws
- Reduce CPU and GPU activity

How to measure dynamic frame rates

1. Use the Energy Diagnostics template in Instruments
2. Use the Core Animation template in Instruments

Watch for:

1. High foreground app and graphics activity
2. Energy usage instrument