Android Hardware Management

Project Report submitted in partial fulfilment of the requirement for the degree of Bachelor of Technology.

in

Computer Science & Engineering

under the Supervision of

Mr. Suman Saha

By

Shashank Parmar (121252)

to

Department of Computer Science & Engineering and Information Technology

Jaypee University of Information Technology Waknaghat, Solan-173234, Himachal Pradesh

Candidate's Declaration

I hereby declare that the work presented in this report entitled **"Android Hardware Management"** in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering/Information Technology** submitted in the department of Computer Science & Engineering and Information Technology**,** Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from August 2015 to May 2016 under the supervision of Mr. Suman Saha (Assistant Professor, Dept. of CSE).

The matter embodied in the report has not been submitted for the award of any other degree or diploma.

Shashank Parmar
121252

Certificate

This is to certify that project report entitled **"Android Hardware Management"**, submitted by Shashank Parmar, 121252 in partial fulfilment for the award of degree of Bachelor of Technology in Information Technology to Jaypee University of Information Technology, Waknaghat; Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date:

Mr. Suman Saha

Assistant Professor

Acknowledgement

I would like to express my gratitude and appreciation to all those who gave me the perfect environment for completion of this report. A special thanks to my final year project supervisor, Mr. Suman Saha, whose stimulating suggestions and encouragement, helped me to get to the thrust of this topic and understand the importance of the project.

I would also like to acknowledge with much appreciation the crucial role of the staff of Computer Laboratory, who provided me with the lab facilities as and when required. Additionally, I appreciate the guidance given by the panels members especially during the previous project presentation who made me realize the various dimensions I was probably missing out and hence, they gave away a room for improvement in the project.

Again a special thanks to my friends who gave me valuable suggestions regarding the project.

Date: 30-5-2016 **Shashank Parmar**

121252

Table of Contents

List of Figures

List of Abbreviations

AOSP - Android Open Source Project

HAL - Hardware Abstraction Layer

OHA - Open Handset Alliance

OEM - Original equipment manufacturer

DRM - Digital rights management

GPS - Global Positioning System

AOT - Ahead-of-Time compiler

ART - Android Runtime

CENS - Centre for Embedded Networked Sensing

OTA - Over-the-air programming

OTG - On The Go

APK - Android application package

Abstract

Android is an open source software stack created for a wide array of devices with different form factors. The primary purposes of Android are to create an open software platform available for carriers, OEMs, and developers to make their innovative ideas a reality and to introduce a successful, real-world product that improves the mobile experience for users.

This system also makes sure there was no central point of failure, where one industry player could restrict or control the innovations of any other. The result is a full, production-quality consumer product with source code open for customization and porting.

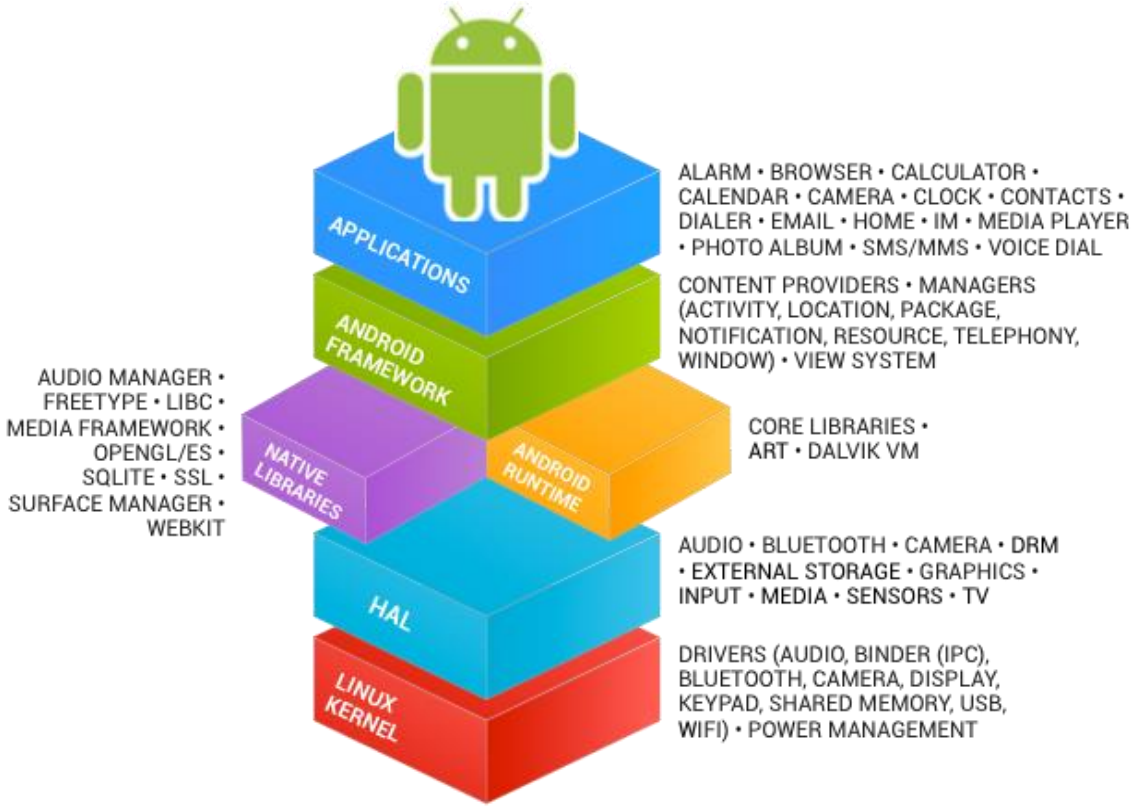

Figure 1. Android stack

Battery life is a perennial user concern. To extend battery life, Android continually adds new features and optimizations to help the platform optimize the off-charger behaviour of applications and devices.

Governance Philosophy

Android was further developed by a group of companies known as the Open Handset Alliance, led by Google. Today, many companies -- both original members of the OHA and others -- have invested heavily in Android. These companies have allocated significant engineering resources to improve Android and bring Android devices to market.

The companies that have invested in Android have done so on its merits because we believe an open platform is necessary. Android is intentionally and explicitly an open source -- as opposed to a free software -- effort; a group of organizations with shared needs has pooled resources to collaborate on a single implementation of a shared product. The Android philosophy is pragmatic, first and foremost. The objective is a shared product that each contributor can tailor and customize.

Uncontrolled customization can, of course, lead to incompatible implementations. To prevent this, the Android Open Source Project also maintains the Android Compatibility Program, which spells out what it means to be "Android compatible" and what is required of device builders to achieve that status. Anyone can (and will!) use the Android source code for any purpose, and we welcome all legitimate uses. However, in order to take part in the shared ecosystem of applications we are building around Android, device builders must participate in the Android Compatibility Program.

The Android Open Source Project is led by Google, who maintains and further develops Android. Although Android consists of multiple subprojects, this is strictly a project management technique. We view and manage Android as a single, holistic software product, not a "distribution", specification, or collection of replaceable parts. Our intent is that device builders, port Android to a device; they don't implement a specification or curate a distribution.

Android is an open platform which is becoming very popular operating system. Its open source code is easily handled by the users to get and use new contents and applications on their handsets. Android is based on Linux kernel. Android device are being activated per day and power management of these device is becoming an issue. The one problem is

common that is Low battery life of the device. It is not a common thing in smartphones. Now days, more powerful with power consuming technologies like GPS, 3G and 3GS. The diverse range of wireless interfaces and sensors, and the increasing popularity of power hungry applications that take advantage of these resources can reduce the battery life of mobile handhelds to few hours of operation. The research community and operating system and hardware vendors found interesting optimisations and techniques to extend the battery life of mobile phones.

CHAPTER 1: INTRODUCTION

1.1 Introduction : Android Hardware Management and the HAL(Hardwar Abstraction Layer)

Android gives you the freedom to implement your own device specifications and drivers. The hardware abstraction layer (HAL) provides a standard method for creating software hooks between the Android platform stack and your hardware. The hardware abstraction layer (HAL) defines a standard interface for hardware vendors to implement and allows Android to be agnostic about lower-level driver implementations. The HAL allows you to implement functionality without affecting or modifying the higher level system. HAL implementations are packaged into modules (.so) file and loaded by the Android system at the appropriate time.

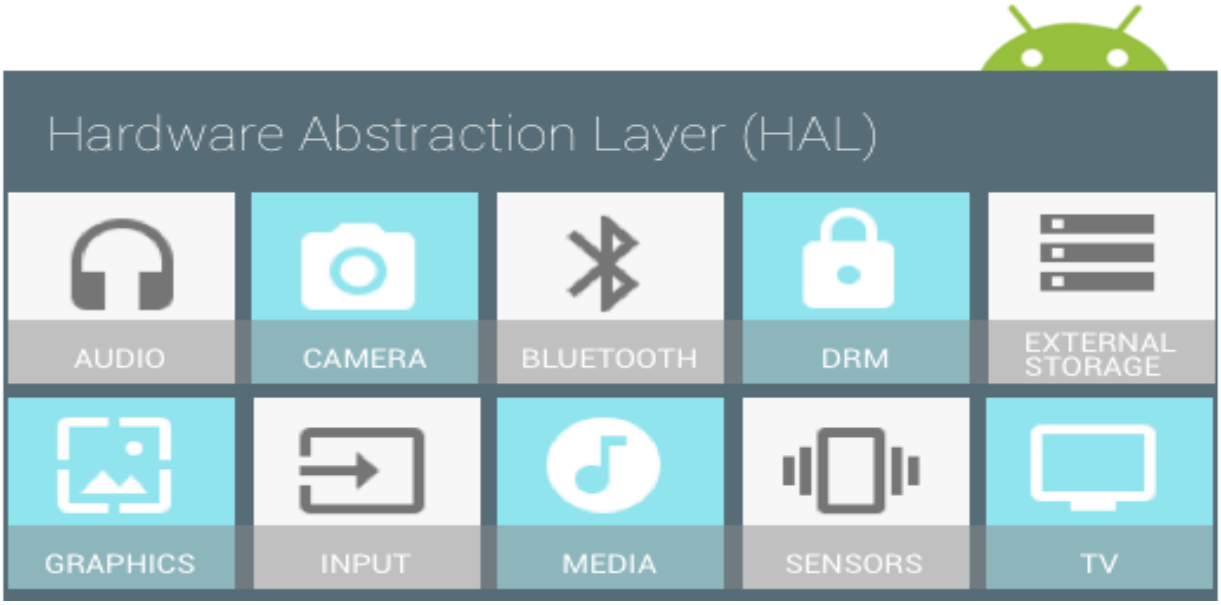

Figure 1 Hardware abstraction layer (HAL) components

You must implement the corresponding HAL (and driver) for the specific hardware your product provides. HAL implementations are typically built into shared library modules (.so files). Android does not mandate a standard interaction between your HAL implementation and your device drivers, so you have free reign to do what is best for your situation. However, to enable the Android system to correctly interact with your hardware, you must abide by the contract defined in each hardware-specific HAL interface.

1.1.1 Standard HAL structure

Each hardware-specific HAL interface has properties that are defined in hardware /libhardware/include/hardware/hardware.h, which guarantee that HALs have a predictable structure. This interface allows the Android system to load the correct versions of your HAL modules in a consistent way. There are two general components that a HAL interface consists of: a module and a device.

A module represents your packaged HAL implementation, which is stored as a shared library (.so file). It contains metadata such as the version, name, and author of the module, which helps Android find and load it correctly. The hardware /libhardware/include/hardware/hardware.h header file defines a struct, hw_module_t that represents a module and contains information such as the module version, author, and name.

In addition, the hw_module_t struct contains a pointer to another struct, hw_module_methods_t, that contains a pointer to an "open" function for the module. This open function is used to initiate communication with the hardware that the HAL is serving as an abstraction for. Each hardware-specific HAL usually extends the generic hw_module_t struct with additional information for that specific piece of hardware. For example in the camera HAL, the camera_module_t struct contains a hw_module_t struct along with other camera-specific function pointers:

```
typedef struct camera_module {
    hw_module_t common;
    int (*get_number_of_cameras)(void);
```

```
    int (*get_camera_info)(int camera_id, struct camera_info *info);
} camera_module_t;
```

When you implement a HAL and create the module struct, you must name it HAL_MODULE_INFO_SYM. For instance, here is an example from the Nexus 9 audio HAL:

```
struct audio_module HAL_MODULE_INFO_SYM = {
    .common = {
        .tag = HARDWARE_MODULE_TAG,
        .module_api_version = AUDIO_MODULE_API_VERSION_0_1,
        .hal_api_version = HARDWARE_HAL_API_VERSION,
        .id = AUDIO_HARDWARE_MODULE_ID,
        .name = "NVIDIA Tegra Audio HAL",
        .author = "The Android Open Source Project",
        .methods = &hal_module_methods,
    },
};
```

A device abstracts the actual hardware of your product. For example, an audio module can contain a primary audio device, a USB audio device, or a Bluetooth A2DP audio device. A device is represented by the hw_device_t struct. Like a module, each type of device defines a more-detailed version of the generic hw_device_t that contains function pointers for specific features of the hardware. For example, the audio_hw_device_t struct type contains function pointers to audio device operations:

```
struct audio_hw_device {
    struct hw_device_t common;
    /**
     * used by audio flinger to enumerate what devices are supported by
     * each audio_hw_device implementation.
     *
```

```
     * Return value is a bitmask of 1 or more values of audio_devices_t
     */
    uint32_t (*get_supported_devices)(const struct audio_hw_device *dev);
  ...
};
typedef struct audio_hw_device audio_hw_device_t;
```

In addition to these standard properties, each hardware-specific HAL interface can define more of its own features and requirements. See the HAL reference documentation as well as the individual instructions for each HAL for more information on how to implement a specific interface.

1.1.2 HAL modules

HAL implementations are built into modules (.so) files and are dynamically linked by Android when appropriate. You can build your modules by creating Android.mk files for each of your HAL implementations and pointing to your source files. In general, your shared libraries must be named in a certain format, so that they can be found and loaded properly. The naming scheme varies slightly from module to module, but they follow the general pattern of:

`<module_type>.<device_name>`.

1.2 Need of battery management

Android mobile devices provide a balance of good hardware performance with a large user application market. According to Nielsen, a global marketing research company, Android is currently the most popular smartphone among users. As the demands on smartphone usage increase, so do complaints about battery life. Many users would be satisfied if their Android smartphones simply lasted a full day on a single charge. Many users must carry their chargers

with them and recharge several times a day under normal usage. Popular online suggestions to extend battery lifetime include manually managing hardware components such as GPS, 3G, Wi-Fi, and Bluetooth and turning them off when they are not in use. However, this approach is wrought with inefficiency and frustration as users struggle to remember when to have different components on/off or they forget to enable a device they need.

Leaving your devices plugged in at 100 percent is also harmful for battery life. Battery University says overcharging is not good for the battery: "Avoiding full charge has benefits, and some manufacturers set the charge threshold lower on purpose to prolong battery life... Li-ion cannot absorb overcharge, and when fully charged the charge current must be cut off. A continuous trickle charge would cause plating of metallic lithium, and this could compromise safety." Also for the developers to test their application they have to plug in their device to the pc repeatedly this exhausts the device battery life cycles and shortens the battery life.

1.3 Synchronous Charging

We wish to modify this concept a bit to apply it to mobiles charging from laptops so that when the mobile is plugged into the laptop both reach the end of their discharge cycles simultaneously.

Synchronous charging is a rather new concept, when we want multiple devices to charge in sync so that they exhaust their battery or get fully charged simultaneously.

This will help us in situations where we have both devices as our dependencies.

Many times we end up in a situation in which we need both our devices like when traveling and we need to use Internet with our mobiles as modems.

Through synchronous charging we will only charge the mobiles from the laptops battery when our estimated projection of the laptops battery life exceeds that of the mobile.

The processing and calculation of the status of batteries of both the devices will be done using an application in the android device and the operating system used will be Ubuntu.

We will use Ubuntu since it is open-source Operating System and we will have a shell script available to us so that we can read the laptop's battery status.

1.4 Adaptive charging

Other battery management techniques in market adaptive fast charging (or rapid charging, quick charging, turbo charging) quick charging allows you to dump a lot of power into your battery by using higher-than-normal voltage until it reaches what's called "saturation" - usually around 60-80% charge depending on how the phone's power management is configured.

At that point, the phone's power controller scales back the amount of power it's receiving and your phone will begin to charge more and more slowly as it approaches 100%. This is where the "adaptive" language comes - quick charging allows your phone to intelligently scale the amount of power it takes from the charger based on the current charge state of the battery.

So, how do you know if your phone supports quick charging? You'll need to do some research on the web. You're going to have to consult Qualcomm's Quick Charge website or your device manufacturer if you want to know for sure without actually trying a quick charger. There are lists of phones with this technology out there - Qualcomm has a very good one that you can find it shouldn't be difficult information to find. Just remember that Quick Charge 2.0, quick charging, fast charging, adaptive fast charging, and turbo charging - they're all usually going to be referring to the same thing.

CHAPTER 2: LITERATURE REVIEW

2.1 Title: Android Smartphone: Battery saving service

Saving power of Smartphone's battery becomes an important because of appearance of applications and technologies that consume more power such as GPS and Wi-Fi. This paper reports on development and evaluation of an Android service to save power, it tries to utilizes Smartphone idle times to stop technologies such as Wi-Fi and Bluetooth and application that may consume battery power; this service reads settings, configured by user, and stops/resumes technologies and application accordingly when events such as Screen goes off/on, Wi-Fi signal becomes weak or not exist, and when user goes to non-covered area. The idea behind this service is to increase battery life by stopping power-consuming while user does not use application and technologies. An empirical study has been conducted to evaluate the effectiveness of this service on Android Smartphone. The result shows there is a slight increase in battery life.

2.2 Title: Adaptive Battery Management on Smartphones

Overview

Modern mobile phones are not single-purpose devices anymore. But rather, they are multi-functional programmable computers. Users run a plethora of applications in addition to voice calling on their smartphones. Significant diversity in usage habits combined with the diversity in hardware platforms makes battery life of smartphones unpredictable. If batteries lasted long enough, unpredictability would not have been a serious concern. However, the linear battery improvements are no match for the rate of new features and applications on smartphones. As a result, the average battery life time of smartphones gets shorter on each new generation of smartphones in the market.

In addition, many pervasive computing applications, such as those developed at CENS, have components that continuously run in the background. Such applications are particularly power consuming. Unlike traditional fully interactive applications, users do not have direct control over the resource consumption of background tasks.

Therefore, many users are usually startled by their smartphone's short battery life when running CENS applications. In this work we introduce a new system to give users control over their phone's battery life. We do so by CENS applications, which run in the background and consume significant power, adaptive to usage and context.

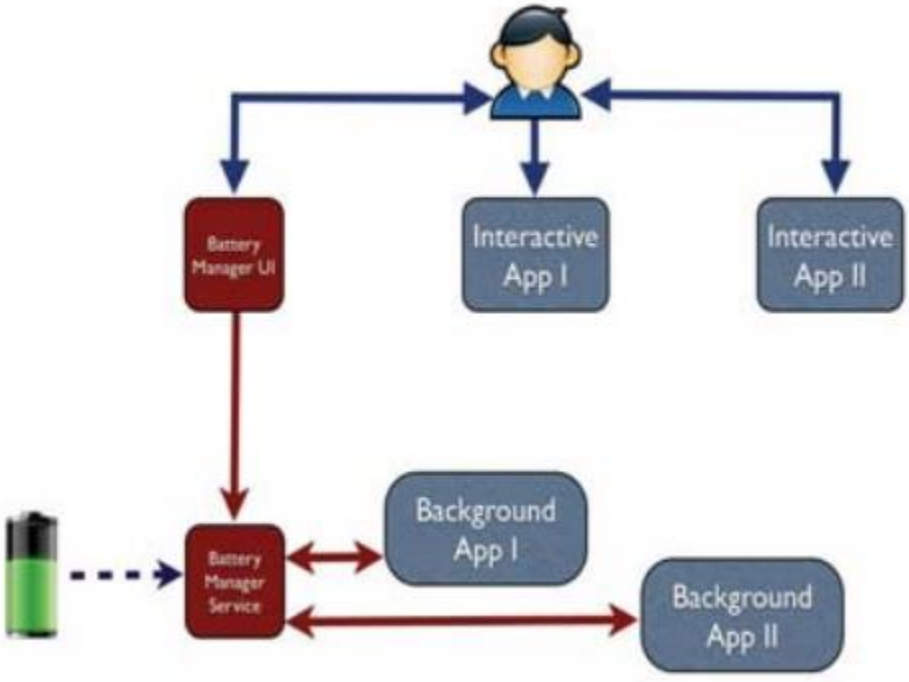

Figure 2 Architecture for adaptive battery management on smartphones

Approach

Four approaches to managing power consumption to minimize user surprises can be considered:-

• Most systems leave battery management entirely to applications. Most commercial mobile platforms such as Android and Symbian follow this approach.

• The opposite of the previous approach is managing energy as a primary system resource. Applications can consume only what is allocated to them by the OS. ECO System and Cinder are re- search operating systems that follow this approach.

• A third approach to guaranteeing a reasonable battery life is limiting the amount of work that applications can do without the user's direct control. For example, the iPhone OS does not allow arbitrary background jobs by limiting multitasking.

• Our approach that we propose in this work combines the best of the first and second approaches.

Managing energy consumption at the highest layer, i.e. applications, is more effective. The application can make better choices to trade off accuracy and performance with energy. On the other hand, applications lack a global view of the system. Specially, effective battery management requires knowledge of other applications workload, something that applications do not expose to others for a good reason.

2.3 Title: Energy Management Techniques in Modern Mobile Handsets

The state of the art of lithium-ion batteries clearly indicates that energy efficiency must be achieved both at the hardware and software level

Introduction

Today's mobile phones are equipped with a wide range of sensing, computational, storage and communication resources that bootstrapped the birth of rich mobile applications such as location aware services and mobile social networks. However, those applications can potentially reduce the battery life of mobile handsets to few hours of operation. Unfortunately, battery technologies have not experienced the same evolution as the rest of hardware components in mobile handsets. Most mobile phones are powered by lithium-ion batteries that can provide many times the energy of other types of batteries in the same fraction of space. However, the state of the art in battery technology shows that the only alternative left at the moment to extend the battery life of mobile phones is reducing the power consumption at the hardware level and designing more energy efficient applications and operating systems.

As we have mentioned in the introduction, power-efficiency in mobile systems can be achieved at different levels. Hence, the survey is structured following taxonomy of the papers under study based on the type of optimisation they are proposing. This classification is as follows:

•

Energy aware operating systems

The main question about energy efficiency in mobile devices is who should be responsible for energy management? Applications or operating system?

Probably the right answer is both. At the operating system level, the main idea is to reduce energy consumption by unifying resource and energy management and by leveraging collaboration between applications and operating system.

In fact, a key part of energy-efficient resources and energy management is having a good understanding of how resources are demanded by users and applications in the system. This section describes some attempts towards energy aware mobile operating systems, energy-efficient resource management and resource profilers.

•

Energy measurements and power models

Understanding how energy is being consumed by the hardware components is essential in order to design energy-aware systems. This section describes some:

•

Users' interaction with applications and computing resources

Battery lifetime has become one of the top usability issues of mobile systems. Hence, improving battery lifetime is highly related to a better understanding of how users interact with their battery and their resources. Any energy-aware system must be able to know when, where and how the user drains the battery and when there will be future charging opportunities. This section comprises different papers trying to understand battery charging cycles and users' resource demands.

•

Wireless interfaces and protocol optimisations

Wireless interfaces are major power consumers on mobile systems. There are multiple ways of making wireless interfaces more efficient at every layer of the protocol stack (also cross-layer optimisations) by taking advantage of the different power states. However, they usually require application, operating system and network infrastructure cooperation. As we have

already mentioned in the introduction, discussing new wireless interfaces and link layer optimizations are not within the scope of this survey.

•

Sensors optimisations

Location-aware applications became one of the most popular services in mobile systems. A mobile device has sensors such as GPS, network-based positioning systems and accelerometer for location with different resolutions and power demands. As a result, there is a trade-off between energy-consumption and accuracy. This section discusses solutions to minimise the energy consumption of continuous sensing at the software level.

•

Computation off-loading

Cloud computing is opening new possibilities to mobile systems in many ways. Computation off-loading has been shown to be effective for extending the computational power and battery life of resource-restricted devices since the late 90s. In fact, even modern mobile operating systems rely more and more on online services running in the cloud.

Remote execution allows migrating computation from battery-powered mobile devices to wall-powered, higher performance machines hosted somewhere on the Internet.

However, there are factors such as network state that can clearly affect its performance. This section covers the most relevant works about computation off-loading in mobile devices from an energy perspective.

The need of energy-awareness in mobile OS

The concept of an energy-aware operating system has been proposed in the late 90s with energy-aware operating systems for laptops like Odyssey and ECO System. In 2000, Ellis pointed up that energy should be considered as a first-class resource in addition to the traditional OS perspective of maximizing performance. Although this topic has been almost abandoned during the mid-2000, it has regained researchers' attention recently due to the energy limitations of current smartphones in which power-hungry applications (or even malware) can reduce the battery life of the handset to few hours of operation. This was the motivation behind mobile energy-aware operating systems for mobile handsets such as Cinder and ErdOS.

There are two opposite propositions about how and by whom energy-aware policies in mobile devices should be performed. On the one hand, some authors suggest that applications must adapt dynamically to energy limitations as in Chameleon [19] but this approach lacks of a central entity responsible for monitor all the resources consumption caused by other applications. On the other hand, other researchers suggest that resources and energy management should be entirely done at the operating system. However, this solution can present scalability problems. Both Odyssey and ECO System present an intermediate solution. They follow a hybrid approach in which both applications and operating system collaborate to reduce the power consumption in a mobile phone. Ideally, the operating system must know applications' resource demands and the available energy resources until the next charging opportunity to reduce the power consumption while maximising user experience. However, new programming models, schedulers, energy measurement tools, resource profilers and power-based APIs must be developed in order to support software-level energy management.

Conclusion

Mobile handsets are still power-hungry devices despite the tremendous efforts done by hardware manufacturers and operating system vendors in the last years. Modern mobile platforms such as Androids are built as modifications of general-purpose operating systems which do not consider energy-efficiency as a key performance goal. In fact, modern handsets incorporate power-hungry hardware resources such as touchscreen displays and location sensors, and they support Internet data services so they are always connected to the network. All these resources bootstrapped a rich ecosystem of mobile applications but their design is clearly driven by usability factors rather than energy efficiency. Since the mid-90s, researchers have been emphasizing the need of considering energy as a fundamental system resource in mobile devices. In this survey, we covered the most relevant articles about energy-efficient resource management in mobile systems that can be implemented in current mobile handsets.

We classified the papers in six categories based on the type of optimisation they propose: operating system and efficient resource management, energy measurements and power models, users' interaction with mobile resources, wireless interfaces and sensors management, and finally, we talked about the new opportunities that process and system migration to the cloud can offer. As far as we know, this is the first survey about mobile green computing in the last decade and we strongly believe that some of the improvements highlighted in this survey will be part of future mobile OS design.

Managing mobile resources from an energy-efficient perspective without diminishing the user experience is clearly one of the most challenging problems in mobile computing now a days. Power management considerations often require certain actions to be deferred, avoided or slowed down to prolong battery life. It can even require changing dynamically the power states of the hardware components and applications behaviour depending on the available resources. However, these techniques can impact the user experience with the handsets.

Moreover, limitations such as the lack of energy-aware support from hardware components make this problem even harder to solve. Hardware manufacturers do not offer enough information about the energy consumption in runtime to the operating system and applications. Many power-hungry resources are embedded in the same chipset as in modern

ARM-based chips and the system do not have enough visibility about the power consumption and the power modes of the different resources available in the device. Consequently, most of the works rely on energy measurements obtained with external multimeters or with inaccurate power models obtained from linear regression techniques.

On the other hand, mobile operating systems must take advantage of all the possibilities they have to save energy. As we can currently see in modern platforms and applications, the dependency on cloud services is becoming more necessary for different purposes such as storage and computation offloading. However, we strongly believe that collaborative mechanisms for sharing resources opportunistically with co-located devices using low-power local wireless connectivity can have two immediate benefits. Firstly, devices can save important amounts of energy and secondly, they can improve the user experience and quality of service by enabling access to remote resources that might not be available locally. Nevertheless, the characteristics of current local wireless interfaces such as Bluetooth make supporting this feature difficult. Mobile computation should not be limited exclusively to the local device and, as a result, resources management should be distributed and collaborative within groups of collocated devices. This approach will need to face new trust schemes, access control policies, security mechanisms, privacy and possibly incentive schemes while trying to minimise the negative impact of users' mobility.

CHAPTER 3: System Development

3.1 Methodology Overview

- **Make a contribution to AOSP**
- **Make an android application**
- **Make a desktop application**

3.1.1 Contribution to AOSP

- The original plan was to contribute to AOSP using a .mk file.
- This file then transforms into a .so file which is an importable module.

3.1.2 Developing an Android Application

- Develop an android app to test if our theory of effective battery management is viable.
- Develop an application to check battery statistics by reading an internal file and broadcasting it on an app.
- Develop the main application which stops the USB charging by modifying an internal file.
- Develop another application which manages the Bluetooth, Wi-Fi , GPS and data connection when the battery level decreases below a particular point or as required by the user.

3.1.3 Developing a Desktop Application

- Develop a desktop app instead to cut the power source from the usb port.
- This gives us new opportunities as well as creates new problems.

Limitations for these above methods

- The first method is the best but there is resource gap and other uncertainties
- There are two fronts to tackle this…
    - The first being kivy, but the documentation on using pygenius is unclear
    - In case of the second method which would be a straight forward android app we lack a rooted device with an older kernel(2.6.32 for Linux).

Fact: Even though Ubuntu and Android are for different hardware specifications and use different machine languages. There are still marked similarities between the two, for example file system and hierarchy.

Therefore problems on both platforms tend to mirror each other.

In case of the third solution we are having problems finding a way to turn off power for ports as it is variable for different machines and kernel versions.

3.2 Project Design

The above mentioned problems played a major role in shaping the design of our current application.

- We made a skeleton android app that uses root privileges to change values in specific position in the android system.
- We made the gui on a desktop app since it facilitates faster programming and grants flexibility.
- We connect these two in a way that allows us to turn off usb charging while still maintaining a possibility of data transfer.

3.3 Implementation

Desktop application:

- Technologies used were python, kiwi, sub-process and executor are the main packages.
- In this desktop app we retrieved pc battery data and bind or unbind the usb port power.

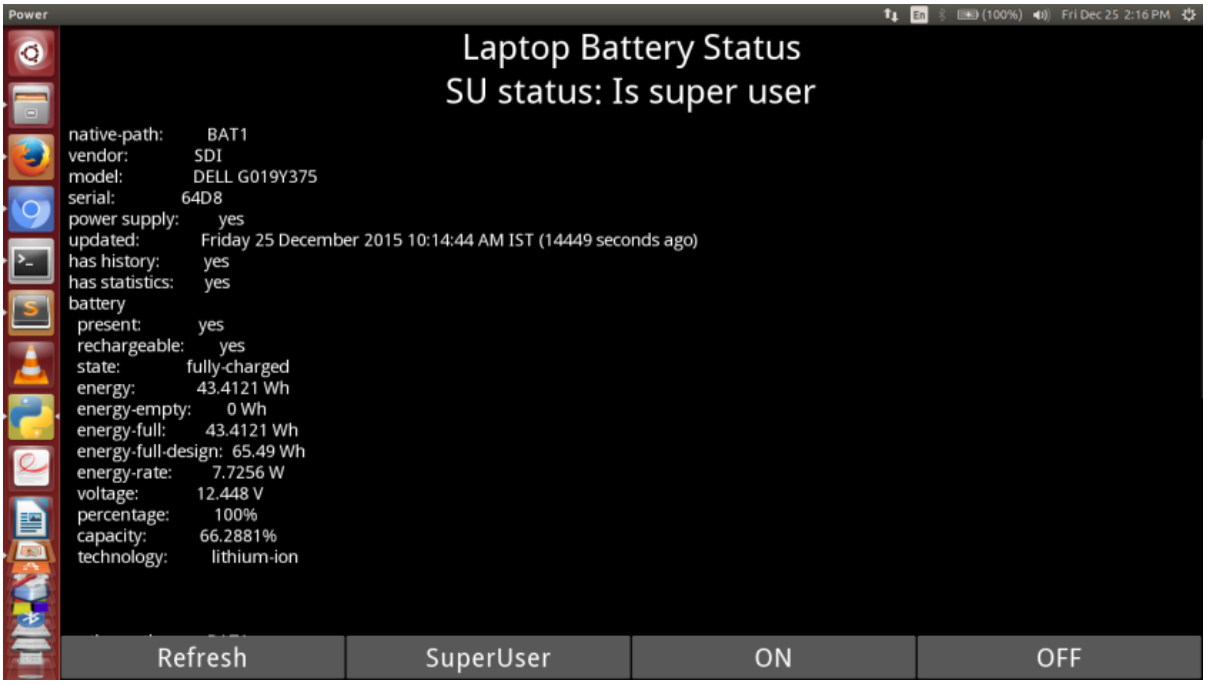

Figure 3 The desktop application

Android application:

- Technologies used were java, eclipse helios and android sdk.
- In this application we retrieved phone's battery data.
- Rooted the phone using towel root and super user. This gave access to the android file system. In the following folder /sys /class /power_supply /battery/ and make changes to two directories battery and usb.
- Create another application that manages the power consuming components even when the android device is lying idle.

Figure 4 The battery status application

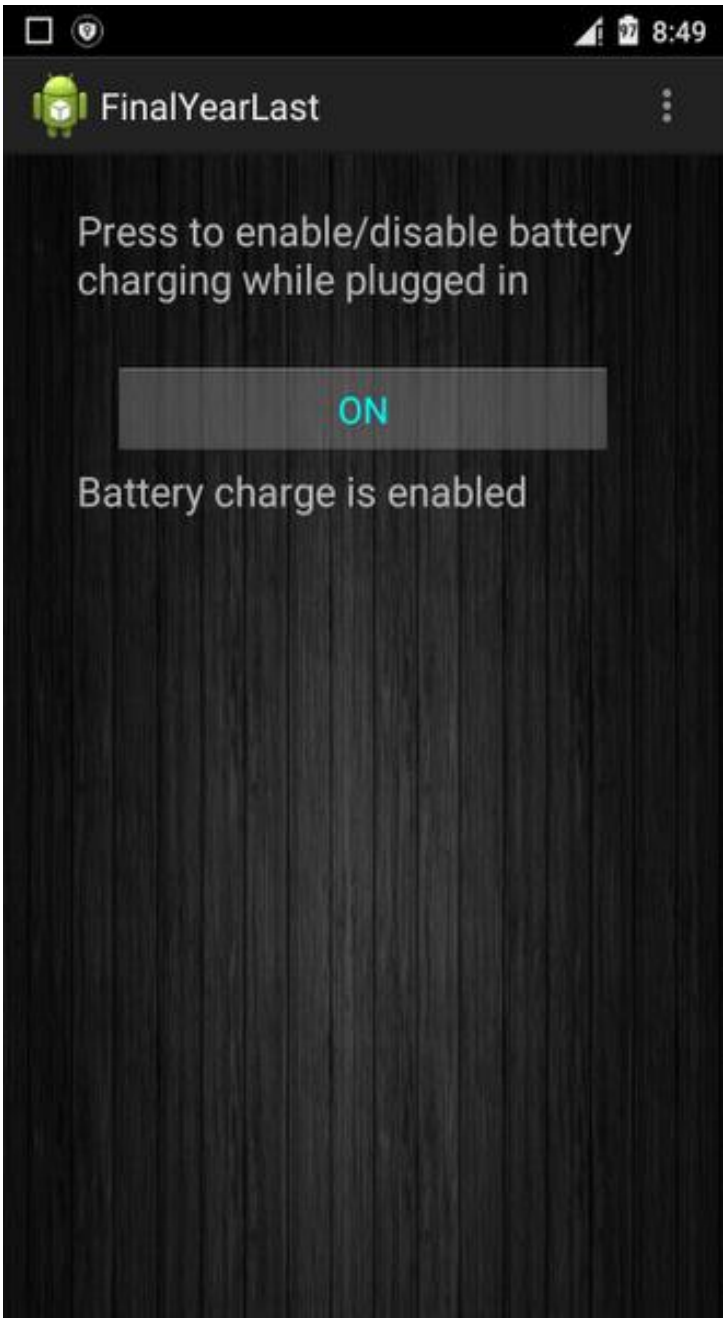

Figure 5 The battery charge control app

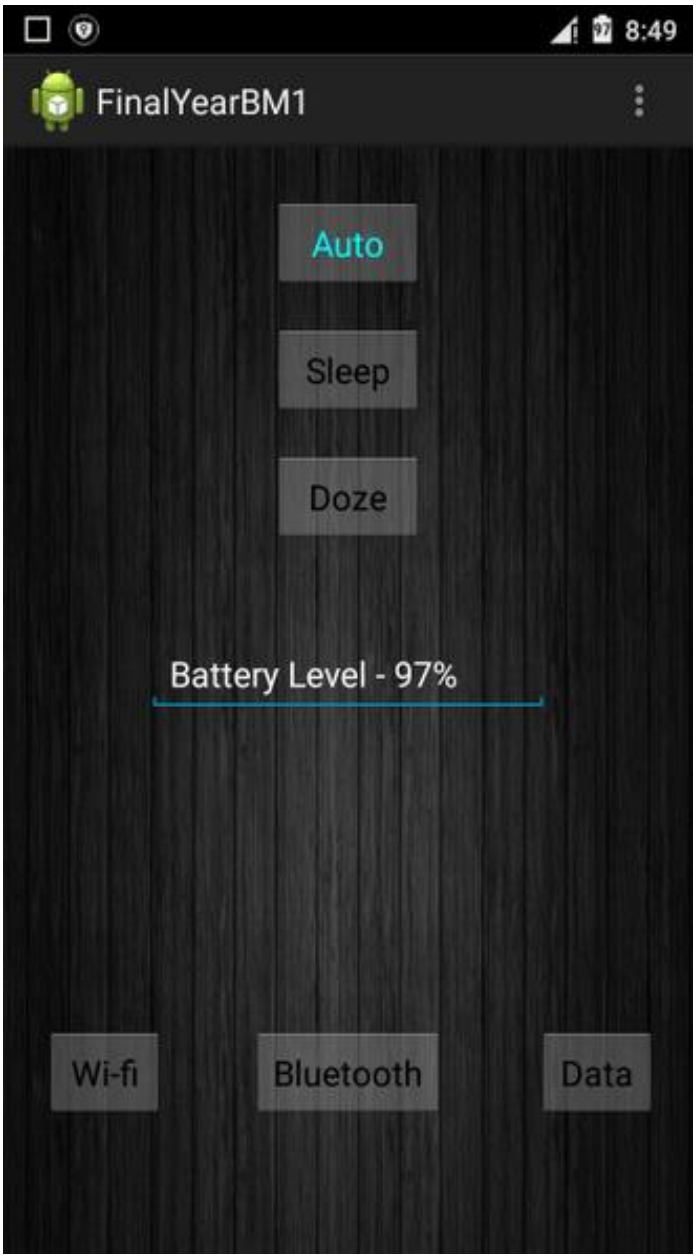

Figure 6 The battery management app

3.4 Explanation and Implementation

Implementation: Linux side

On Linux Terminal:

achyut@achyut-Inspiron-3537:~$ upower -e (--enumerate)

bash: syntax error near unexpected token `('

achyut@achyut-Inspiron-3537:~$ upower -e

/org/freedesktop/UPower/devices/line_power_ACAD

/org/freedesktop/UPower/devices/battery_BAT1

achyut@achyut-Inspiron-3537:~$ upower -i /org/freedesktop/UPower/devices/battery_BAT1

Native-path: BAT1

Vendor: SDI

Model: DELL G019Y375

Serial: 64D8

Power supply: yes

Updated: Friday 18 December 2015 10:56:57 PM IST (1921 seconds ago)

Has history: yes

Has statistics: yes

Battery present: yes

Rechargeable: yes

State: fully-charged

Energy: 43.4121 Wh

Energy-empty: 0 Wh

Energy-full: 43.4121 Wh

Energy-full-design: 65.49 Wh

Energy-rate: 0.0111 W

Voltage: 12.618 V

Percentage: 100%

Capacity: 66.2881%

Technology: lithium-ion

3.5 Explanation: An exploded view

So the important part is how does it all connect?

Think of the internal structure of Android as having only two parts. Those two parts of Android would be:

Interfaces which consists of:

Accessories

Audio

Bluetooth

Camera

DRM

Graphics

Input

Media

Sensors

Storage

TV

Core Technologies which consist of:

ART and Dalvik

Configuration

Data Usage

Debugging

Device Administration

HAL File Reference

OTA Updates

Power

Testing Infrastructure

Within the core technologies we will be concentrating on power:

Power Management

Component Power

Device Power

Power Values

Battery Use

And further within power on power management and device power.

The Interface is the one that contains HAL, and as explained in the introduction its only an interface that defines structures that helps control the hardware, but there is only so much you can do to improve an interface, so what do we mean when we say that we want to "improve the HAL layer of android"?

We mean that we want to improve on the functions interacting with the HAL on the android side and those would be the core technologies, since the implementation of HAL and the Hardware itself are dependent on the manufacturer. But this is only our perspective of how we understand things; somebody else may tell you a different view while meaning the same thing the entire time.

3.5.1 Optimizing Battery Life

Dependencies and prerequisites

- Experience with Intents and Intent Filters

For your app to be a good citizen, it should seek to limit its impact on the battery life of its host device. After this class you will be able to build apps that modify their functionality and behaviour based on the state of the host device.

By taking steps such as disabling background service updates when you lose connectivity, or reducing the rate of such updates when the battery level is low, you can ensure that the impact of your app on battery life is minimized, without compromising the user experience.

Lessons

Optimizing for Doze and App Standby

Learn how to test and optimize your app for the power-management features introduced in Android 6.0 Marshmallow.

Monitoring the Battery Level and Charging State

Learn how to alter your app's update rate by determining, and monitoring, the current battery level and changes in charging state.

Determining and Monitoring the Docking State and Type

Optimal refresh rates can vary based on how the host device is being used. Learn how to determine, and monitor, the docking state and type of dock being used to affect your app's behaviour.

Determining and Monitoring the Connectivity Status

Without Internet connectivity you can't update your app from an online source. Learn how to check the connectivity status to alter your background update rate. You'll also learn to check for Wi-Fi or mobile connectivity before beginning high-bandwidth operations.

Manipulating Broadcast Receivers On Demand

Broadcast receivers that you've declared in the manifest can be toggled at runtime to disable those that aren't necessary due to the current device state. Learn to improve efficiency by toggling and cascading state change receivers and delay actions until the device is in a specific state.

3.5.2 Optimizing for Doze and App Standby

In this part

1. Understanding Doze
    1. Doze restrictions
    2. Adapting your app to Doze
2. Understanding App Standby
3. Using GCM to Interact with Your App
4. Support for Other Use Cases
5. Testing with Doze and App Standby

1. Testing your app with Doze
2. Testing your app with App Standby

6. Example Use Cases for Whitelisting

Starting from Android 6.0 (API level 23), Android introduces two power-saving features that extend battery life for users by managing how apps behave when a device is not connected to a power source. *Doze* reduces battery consumption by deferring background CPU and network activity for apps when the device is unused for long periods of time. *App Standby* defers background network activity for apps with which the user has not recently interacted.

Doze and App Standby manage the behaviour of all apps running on Android 6.0 or higher, regardless whether they are specifically targeting API level 23. To ensure the best experience for users, test your app in Doze and App Standby modes and make any necessary adjustments to your code. The sections below provide details.

Understanding Doze

If a user leaves a device unplugged and stationary for a period of time, with the screen off, the device enters Doze mode. In Doze mode, the system attempts to conserve battery by restricting apps' access to network and CPU-intensive services. It also prevents apps from accessing the network and defers their jobs, syncs, and standard alarms.

Periodically, the system exits Doze for a brief time to let apps complete their deferred activities. During this *maintenance window*, the system runs all pending syncs, jobs, and alarms, and lets apps access the network.

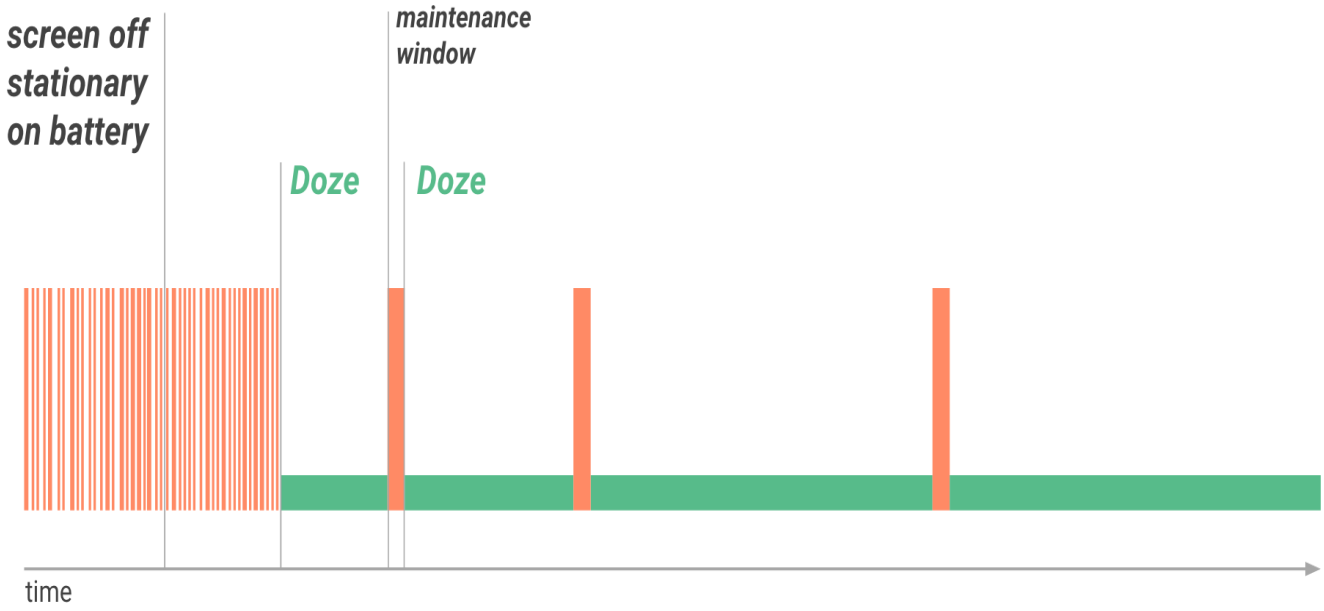

Figure 7 Doze provides a recurring maintenance window for apps to use the network and handle pending activities.

At the conclusion of each maintenance window, the system again enters Doze, suspending network access and deferring jobs, syncs, and alarms. Over time, the system schedules maintenance windows less and less frequently, helping to reduce battery consumption in cases of longer-term inactivity when the device is not connected to a charger.

As soon as the user wakes the device by moving it, turning on the screen, or connecting a charger, the system exits Doze and all apps return to normal activity.

Doze restrictions

The following restrictions apply to your apps while in Doze:

- Network access is suspended.

- The system ignores wake locks.
- Standard `AlarmManager` alarms (including `setExact()` and `setWindow()`) are deferred to the next maintenance window.
    -
    - If you need to set alarms that fire while in Doze, use `setAndAllowWhileIdle()` or `setExactAndAllowWhileIdle()`.
    - Alarms set with `setAlarmClock()` continue to fire normally — the system exits Doze shortly before those alarms fire.
- The system does not perform Wi-Fi scans.
- The system does not allow sync adapters to run.
- The system does not allow `JobScheduler` to run.

Doze checklist

- If possible, use GCM for downstream messaging.
- If your users must see a notification right away, make sure to use a GCM high priority message.
- Provide sufficient information within the initial message payload, so subsequent network access is unnecessary.
- Set critical alarms with `setAndAllowWhileIdle()` and `setExactAndAllowWhileIdle()`.
- Test your app in Doze.

Adapting your app to Doze

Doze can affect apps differently, depending on the capabilities they offer and the services they use. Many apps function normally across Doze cycles without modification. In some cases, you must optimize the way that your app manages network, alarms, jobs, and syncs. Apps should be able to efficiently manage activities during each maintenance window.

Doze is particularly likely to affect activities that `AlarmManager` alarms and timers manage, because alarms in Android 5.1 (API level 22) or lower do not fire when the system is in Doze.

To help with scheduling alarms, Android 6.0 (API level 23) introduces two new `AlarmManager` methods: `setAndAllowWhileIdle()` and `setExactAndAllowWhileIdle()`. With these methods, you can set alarms that will fire even if the device is in Doze.

Note: Neither `setAndAllowWhileIdle()` nor `setExactAndAllowWhileIdle()` can fire alarms more than once per 15 minutes per app.

The Doze restriction on network access is also likely to affect your app, especially if the app relies on real-time messages such as tickles or notifications. If your app requires a persistent connection to the network to receive messages, you should use Google Cloud Messaging (GCM) if possible.

To confirm that your app behaves as expected with Doze, you can use adb commands to force the system to enter and exit Doze and observe your app's behaviour.

3.6 Monitoring the Battery Level and Charging State

In this section

1. **Determine the Current Charging State**
2. **Monitor Changes in Charging State**
3. **Determine the Current Battery Level**
4. **Monitor Significant Changes in Battery Level**
5. **Intents and Intent Filters**

When you're altering the frequency of your background updates to reduce the effect of those updates on battery life, checking the current battery level and charging state is a good place to start.

The battery-life impact of performing application updates depends on the battery level and charging state of the device. The impact of performing updates while the device is charging over AC is negligible, so in most cases you can maximize your refresh rate whenever the device is connected to a wall charger. Conversely, if the device is discharging, reducing your update rate helps prolong the battery life.

Similarly, you can check the battery charge level, potentially reducing the frequency of—or even stopping—your updates when the battery charge is nearly exhausted.

Determine the Current Charging State

Start by determining the current charge status. The `BatteryManager` broadcasts all battery and charging details in a sticky `Intent` that includes the charging status.

Because it's a sticky intent, you don't need to register a `BroadcastReceiver`—by simply calling `registerReceiver` passing in `null` as the receiver as shown in the next snippet, the current battery status intent is returned. You could pass in an actual `BroadcastReceiver` object here, but we'll be handling updates in a later section so it's not necessary.

```
IntentFilter ifilter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent batteryStatus = context.registerReceiver(null, ifilter);
```

You can extract both the current charging status and, if the device is being charged, whether it's charging via USB or AC charger:

```
// Are we charging / charged?
int status = batteryStatus.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
boolean isCharging = status == BatteryManager.BATTERY_STATUS_CHARGING ||
                             status == BatteryManager.BATTERY_STATUS_FULL;

// How are we charging?
int chargePlug = batteryStatus.getIntExtra(BatteryManager.EXTRA_PLUGGED, -
1);
boolean usbCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_USB;
boolean acCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_AC;
```

Typically you should maximize the rate of your background updates in the case where the device is connected to an AC charger, reduce the rate if the charge is over USB, and lower it further if the battery is discharging.

Monitor Changes in Charging State

The charging status can change as easily as a device can be plugged in, so it's important to monitor the charging state for changes and alter your refresh rate accordingly.

The `BatteryManager` broadcasts an action whenever the device is connected or disconnected from power. It's important to receive these events even while your app isn't running—particularly as these events should impact how often you start your app in order to initiate a background update—so you should register a `BroadcastReceiver` in your manifest to listen for both events by defining the `ACTION_POWER_CONNECTED` and `ACTION_POWER_DISCONNECTED` within an intent filter.

```
<receiver                              android:name=".PowerConnectionReceiver">
                                                                  <intent-filter>
     <action  android:name="android.intent.action.ACTION_POWER_CONNECTED"/>
                                                                              <action
android:name="android.intent.action.ACTION_POWER_DISCONNECTED"/>
                                                                 </intent-filter>
</receiver>
```

Within the associated BroadcastReceiver implementation, you can extract the current charging state and method as described in the previous step.

```
public   class   PowerConnectionReceiver   extends   BroadcastReceiver   {
                                                                    @Override
        public   void   onReceive(Context   context,   Intent   intent)   {
         int status = intent.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
                              boolean     isCharging     =     status     ==
BatteryManager.BATTERY_STATUS_CHARGING                                    ||
                               status == BatteryManager.BATTERY_STATUS_FULL;
```

```
        int chargePlug = intent.getIntExtra(BatteryManager.EXTRA_PLUGGED,
-1);
                        boolean usbCharge = chargePlug ==
BatteryManager.BATTERY_PLUGGED_USB;
                         boolean acCharge = chargePlug ==
BatteryManager.BATTERY_PLUGGED_AC;
                                                                        }
}
```

Determine the Current Battery Level

In some cases it's also useful to determine the current battery level. You may choose to reduce the rate of your background updates if the battery charge is below a certain level.

You can find the current battery charge by extracting the current battery level and scale from the battery status intent as shown here:

```
int level = batteryStatus.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
int scale = batteryStatus.getIntExtra(BatteryManager.EXTRA_SCALE, -1);

float batteryPct = level / (float)scale;
```

Monitor Significant Changes in Battery Level

You can't easily continually monitor the battery state, but you don't need to.

Generally speaking, the impact of constantly monitoring the battery level has a greater impact on the battery than your app's normal behavior, so it's good practice to only monitor significant changes in battery level—specifically when the device enters or exits a low battery state.

The manifest snippet below is extracted from the intent filter element within a broadcast receiver. The receiver is triggered whenever the device battery becomes low or exits the low condition by listening for `ACTION_BATTERY_LOW` and `ACTION_BATTERY_OKAY`.

```
<receiver                                android:name=".BatteryLevelReceiver">
<intent-filter>
      <action     android:name="android.intent.action.ACTION_BATTERY_LOW"/>
     <action     android:name="android.intent.action.ACTION_BATTERY_OKAY"/>
                                                                  </intent-filter>
</receiver>
```

It is generally good practice to disable all your background updates when the battery is critically low. It doesn't matter how fresh your data is if the phone turns itself off before you can make use of it.

CHAPTER 4: Analysis and Results

4.1 Ageing of Lithium-ion

The lithium-ion battery works on ion movement between the positive and negative electrodes. In theory such a mechanism should work forever, but cycling, elevated temperature and aging decrease the performance over time. Manufacturers take a conservative approach and specify the life of Li-ion in most consumer products as being between 300 and 500 discharge/charge cycles.

Evaluating battery life on counting cycles is not conclusive because a discharge may vary in depth and there are no clearly defined standards of what constitutes a cycle (BU-501: Basics About Discharging). In lieu of cycle count, some device manufacturers suggest battery replacement on a date stamp, but this method does not take usage into account. A battery may fail within the allotted time due to heavy use or unfavourable temperature conditions; however, most packs last considerably longer than what the stamp indicates.

The performance of a battery is measured in capacity, a leading health indicator. Internal resistance and self-discharge also play roles, but these are less significant in predicting the end of battery life with modern Li-ion.

Figure 1 illustrates the capacity drop of 11 Li-polymer batteries that have been cycled at a Cadex laboratory. The 1,500mAh pouch cells for mobile phones were first charged at a current of 1,500mA (1C) to 4.20V/cell and then allowed to saturate to 0.05C (75mA) as part of the full charge saturation. The batteries were then discharged at 1,500mA to 3.0V/cell, and the cycle was repeated. The expected capacity loss of Li-ion batteries was uniform over the delivered 250 cycles and the batteries performed as expected.

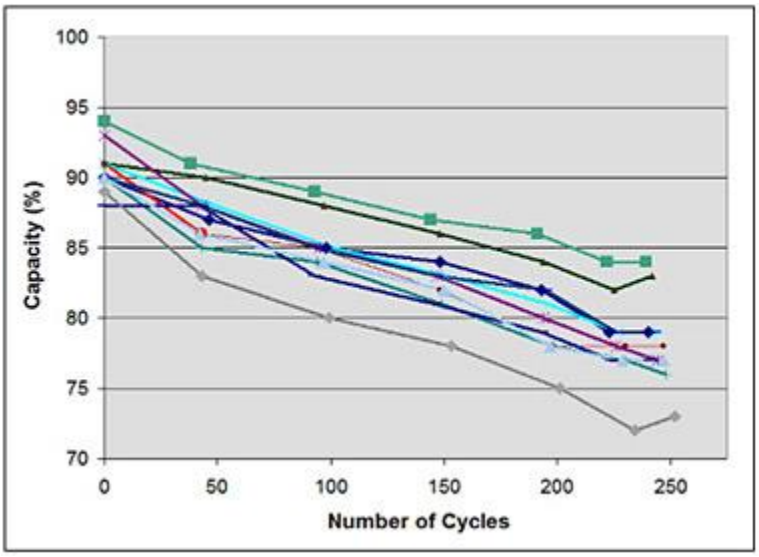

Figure 8: Capacity drop as part of cycling. Eleven new Li-ion were tested on a Cadex C7400 battery analyzer. All packs started at a capacity of 88–94% and decreased to 73–84% after 250 full discharge cycles. The 1500mAh pouch packs are used in mobile phones.

Although a battery should deliver 100 percent capacity during the first year of service, it is common to see lower than specified capacities, and shelf life may contribute to this loss. In addition, manufacturers tend to overrate their batteries, knowing that very few users will do spot-checks and complain if low. Not having to match single cells in mobile phones and tablets, as is required in multi-cell packs, opens the floodgates for a much broader performance acceptance. Cells with lower capacities may slip through cracks without the consumer knowing.

Similar to a mechanical device that wears out faster with heavy use, the depth of discharge (DoD) determines the cycle count of the battery. The smaller the discharge (low DoD), the longer the battery will last. If at all possible, avoid full discharges and charge the battery more often between uses. Partial discharge on Li-ion is fine. There is no memory and the battery does not need periodic full discharge cycles to prolong life. The exception may be a

periodic calibration of the fuel gauge on a smart battery or intelligent device. (BU-603: How to Calibrate a “Smart” Battery)

Table 2 compares the number of discharge/charge cycles Li-ion can deliver at various DoD levels before the battery capacity drops to 70 percent. All other variables such as charge voltage, temperature and load currents are set to average default settings.

Depth of discharge	Discharge cycles
100% DoD	300–500
50% DoD	1,200–1,500
25% DoD	2,000–2,500
10% DoD	3,750–4,700

Table 1: Cycle life as a function of depth of discharge.

A partial discharge reduces stress and prolongs battery life. Elevated temperature and high currents also affect cycle life.

Lithium-ion suffers from stress when exposed to heat, so does keeping a cell at a high charge voltage. A battery dwelling above 30°C (86°F) is considered *elevated temperature* and for most Li-ion a voltage above 4.10V/cell is deemed as *high voltage*. Exposing the battery to high temperature and dwelling in a full state-of-charge for an extended time can be more stressful than cycling. Table 3 demonstrates capacity loss as a function of temperature and SoC.

Temperature	40% charge	100% charge
0°C	98%	94%
25°C	96%	80%
40°C	85%	65%
60°C	75%	60% (after 3 months)

Table 2: Estimated recoverable capacity when storing Li-ion for one year at various temperatures. Elevated temperature hastens permanent capacity loss. Not all Li-ion systems behave the same.

Most Li-ions charge to 4.20V/cell, and every reduction in peak charge voltage of 0.10V/cell is said to double the cycle life. For example, a lithium-ion cell charged to 4.20V/cell typically delivers 300–500 cycles. If charged to only 4.10V/cell, the life can be prolonged to 600–1,000 cycles; 4.0V/cell should deliver 1,200–2,000 and 3.90V/cell should provide 2,400–4,000 cycles.

On the negative side, a lower peak charge voltage reduces the capacity the battery stores. As a simple guideline, every 70mV reduction in charge voltage lowers the overall capacity by 10 percent. Applying the peak charge voltage on a subsequent charge will restore the full capacity.

In terms of longevity, the optimal charge voltage is 3.92V/cell. Battery experts believe that this threshold eliminates all voltage-related stresses; going lower may not gain further benefits but induce other symptoms.

Table 4 summarizes the capacity as a function of charge levels. (All values are estimated; Energy Cells with higher voltage thresholds may deviate.)

Charge level (V/cell)	**Discharge cycles**	**Capacity at full charge**
[4.30]	[150 – 250]	~[114%]
4.20	**300 – 500**	**100%**
4.10	600 – 1,000	~86%
4.00	1,200 – 2,000	~72%
3.92	2,400 – 4,000	~58%

Table 3: Discharge cycles and capacity as a function of charge voltage limit. Every 0.10V drop below 4.20V/cell doubles the cycle but holds less capacity. Raising the voltage above 4.20V/cell would shorten the life.

Guideline: Every 70mV drop in charge voltage lowers the usable capacity by 10%.

Most chargers for mobile phones, laptops, tablets and digital cameras charge Li-ion to 4.20V/cell. This allows maximum capacity, because the consumer wants nothing less than optimal runtime. Industry, on the other hand, is more concerned about longevity and may choose lower voltage thresholds. Satellites and electric vehicles are such examples.

For safety reasons, many lithium-ions cannot exceed 4.20V/cell. (Some NMC are the exception.) While a higher voltage boosts capacity, exceeding the voltage shortens service life and compromises safety. Figure 5 demonstrates cycle count as a function of charge voltage. At 4.35V, the cycle count of a regular Li-ion is cut in half.

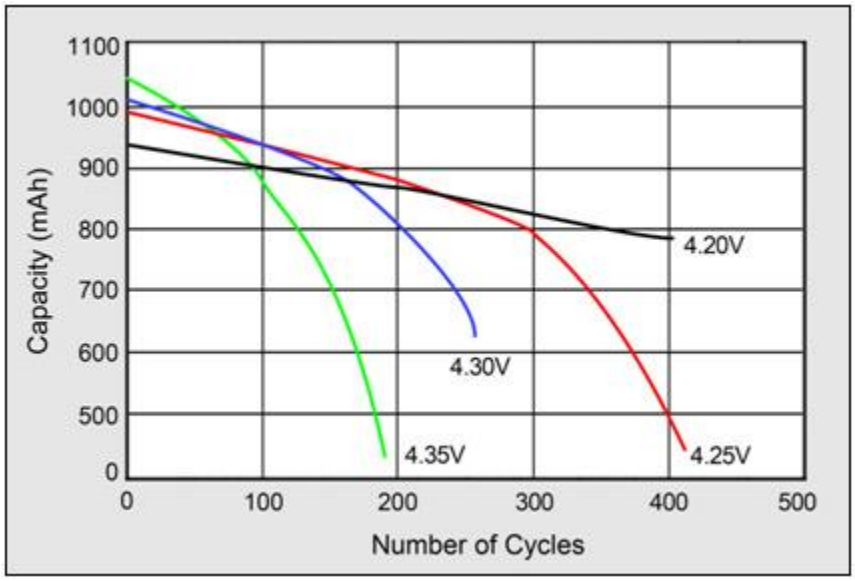

Figure 9: Effects on cycle life at elevated charge voltages. Higher charge voltages boost capacity but lowers cycle life and compromises safety.

Besides selecting the best-suited voltage thresholds for a given application, a regular Li-ion should not remain at the high-voltage ceiling of 4.20V/cell for an extended time. The Li-ion charger turns off the charge current and the battery voltage reverts to a more natural level. This is like relaxing the muscles after a strenuous exercise.

4.2 Results

- We constructed our concept of synchronous charging after reading many reports, surveys and research papers.
- There was no conclusive evidence with us that would confirm that our current concept is applicable as well as feasible.
- We made a flowchart that would lead us to our goal:
  - Application
  - Gather data

  - Android Application
  - Gather data
  - Make a library
  - Contribute to AOSP

- /sys/class/power_supply/battery/ which gives some info/control over charging issues. In particular there is charging_enabled which gives the current state (0 not charging, 1 charging) and may be writable on some phones
- $adb shell
- $ cat /sys/class/power_supply/battery/charging_enabled
- 1
- There is also a file charger_control which sounds promising and is writable by root.
- There are different drivers for battery associated with different android versions and manufacturers.
- e.g. /sys/module/msm_battery/parameters/usb_chg_enable
- Root access is also required to modify these system files programmatically.

In the end our concept of not charging through USB worked fine on devices with root privileges. The other two concepts of doze and sleep in our application were really successful.

CHAPTER 5: CONCLUSION

During the course of this project, we understood the role of the Open Handset Alliance, Android governance, the Android work-flow and the process of contributing code, how to root and what it means to root an Android handset. In particular we learned of Android's battery management system and its various intricacies. We mapped out multiple solutions to the problem of Android battery life and battery health. During our research of the android framework and working of its open-source segment we located a repository on Github that would allow us to turn On or Off the battery charging of the connected handset.

There were three possible solutions to the problem:

1) Make a contribution to AOSP and have it pulled.

2) Make an android app and use this to optimize battery charging (Requires us to root handset)

3) Make a desktop app instead to cut USB power Off from the source.

The first method is the best but there is a resource gap and other uncertainties, we are currently making progress on the other two front but not without barriers:

In the third solution we have a problem finding a way to turn off power for usb ports, its variable for different machines and largely hardware dependent.

We are actively working on two fronts on the second solution, while using kivy the documentation on using Pygenius for to import the git repository is unclear, whereas on android we lack a device we can root. But with the course of the project we were successful in implementing our concept on various devices with specific kernels and root privileges.

REFERENCES

Papers:

[1] Adaptive Battery Management on Smartphones - Deborah Estrin, Faculty, PI Hossein Falaki, Ramesh Govindan

[2] An Efficient Energy Management System for Android Phone - Young-Seol Lee and Sung-Bae Cho

[3]Energy-Aware Computing for Android Platforms - Hung-Ching Chang & Abhishek R Agrawal, Department of Computer Science, Virginia Tech.

[4] Android Smartphone: Battery saving service - International Conference on Research and Innovation in Information Systems

Web Links:

[1] http://research.cens.ucla.edu/urban/2011/part04.pdf [2] https://ieeeplore.ieee.org

[2] http://www.cl.cam.ac.uk/~nv240/papers/IEEE_EnergyManagementTechniques_Survey.pdf

[3] http://scape.cs.vt.edu/wp-content/uploads/2012/08/ITJ12_Android_Energy-Aware.pdf

[4] http://www.embedded.com/design/power-optimization/4438556/USB-battery-charging-protocols-Android-based-design

[5] http://superuser.com/questions/460721/charging-mobile-through-laptop-any-ill-effects

[6] http://www.extremetech.com/computing/115251-how-usb-charging-works