University of Oldenburg
Faculty II – Computer Science, Law and Economics
Software Engineering Group

2014

Method to Derive Energy Profiles for Android Platform

11.03.2014

written by: Andrey Saksonov

reviewed by: Andreas Winter

Abstract

To improve quality of mobile applications on Android platform in sense of energy-efficiency, the programmers need appropriate tools. One of the method to estimate energy consumption of mobile applications is Energy Profiling (for example, using reference implementation - Android Power Profiles). This method allows to estimate energy consumption online, i.e. without using any external devices, while using reference data obtained via prior using of offline measurements tools.

The first-class entity of this method is Energy Profile of target device that contains information about distinct energy consumption of each component. There are at least two reasons why it is may be needed to derive Energy Profiles for specific Android device. First is inappropriate quality of built-in Android Power Profile for most of the devices presented on market. The significant improvement may be achieved even using reference power model developed for Android Power Profiles while using updated (i.e. derived for concrete target device) Power Profile. The second reason is using non-reference power models. Of course many engineers may consider using of specific power models that are suitable for Energy Profiling of very specific applications. In this case, the will need to have a method to obtain Energy Profile for this private power model.

This thesis describes the method of deriving various Energy Profiles for Android mobile devices. The following points are considered in this thesis: choosing appropriate hardware and architecture of software needed to automate the process of deriving Energy Profiles for Android mobile devices. The method was evaluated using test Android device and satisfactory improvement of estimation of energy consumption using reference power model (Android Power Profiles) was observed.

Contents

1 *Introduction*

Nowadays, smartphones and other mobile devices with mobile operating systems consume a lot of energy. Users are forced to charge their phones at least once a day (see example of power consumption statistics on Figure 1. Android Battery Usage - Sony Xperia ZL). To improve user experience on mobile devices, developers try to optimize energy consumption of their applications. However, usually it is done not in appropriate manner and existing applications suffers from energy consumption bugs. Quotation from article by Philippe Michelon: “According to a study made by P. Vekris: “55% of 328 applications using wakelocks do not follow our policies for no-sleep bugs” [2012]. Some major applications have been released with No-Sleep bugs." [1] To improve application battery consumption appropriate tools are needed. This thesis focuses on improving existing methods of Energy Profiling for mobile applications.

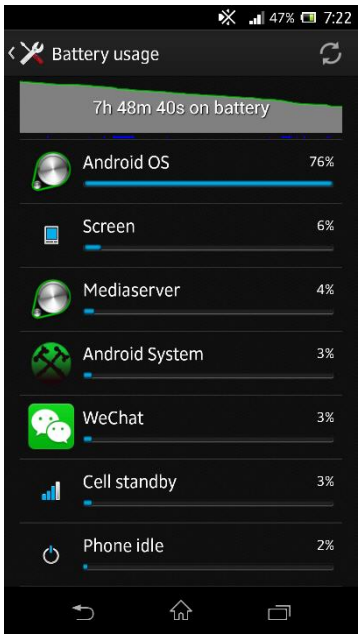


Figure 1. Android Battery Usage - Sony Xperia ZL

1.1 Motivation

Android OS is a multitasking operating system for mobile devices [2]. On mobile platforms, one of the most significant quality criteria of applications is their energy-efficiency [3]. The Energy consumption index is so important that many application developers include tests for energy-efficiency of mobile application in the QA phase of application development [4]. Control over energy consumption of mobile applications helps to increase battery life of mobile devices. Battery Life can be viewed from two separate perspectives. It can refer to the operating time of devices without recharging the battery, which is the primary goal on mobile devices. In

addition, it may refer to the count of cycles a battery can be charged (Li-Ion battery, which is most used in consumer electronics can be recharged only limited number of cycles). However, these two ways of improving battery life is interconnected. Optimizing battery life of mobile devices obeys the general approach in software engineering: measure before optimizing. Processes of such measures for detecting "hot spots" in application's energy consumption called "Energy Profiling". Necessary to distinguish the *offline* and *online* measurements. Offline measurements usually done by using external measuring device with a "reference" device for testing. It is even recommended to measure with a "fake" test battery, which is just source of direct current with fixed level of voltage. This approach helps to minimize interference of battery properties on measured values. On other hand, online measurements usually is estimation done programmatically by software on device (e.g. every phone should be able to display current battery level) or some values pulled from diagnostic hardware of the Android device (if present). Therefore, online measurements usually done using results of reference offline measurements. There exist three main methods of measuring battery consumption online on Android platform [5]:

1. *BatteryManager API* is an application interface available through libraries of Android SDK. It allows measuring the following parameters: current battery state (charging/charged/discharging), source of charging – USB/PSU/Wireless PSU (some devices on Android platform are supporting wireless charging – Qi), level of battery charge in percent, estimated battery wear – Good / Cold / Dead (Overheat) / Dead (Overvoltage), battery temperature and current voltage [6]. This API allows only rough calculations, as the step for changing values is big and time of changing the value is not determined (it is updated by system service with interval specified by vendor of device).

2. *Linux Kernel Index Nodes*. Data provided by the *sysfs* – subsystem of Linux kernel, which exports information about devices and drivers from the kernel device model to user space [7]. Usually, battery information is available through files in node `/sys/class/power_supply/battery`. File set and its contents depend on the mobile device model. On many devices (e.g. Samsung, ASUS) these files contains only current voltage information. On certain Motorola devices currently detected current (amperage) and estimated full charge capacity are available [8]. On some HTC devices, these files may be moved to another location or data can be presented in non-standard format. There is an open source application available, which suffers from issues with this technique – *CurrentWidget* [9]. The author of this application chose the way of supporting number device on the market via "Factory" software concept (in object-oriented computer programming, a factory is an object for creating other objects, an abstraction of a constructor, and can be used to implement various allocation schemes [10]). This approach is hard to use to perform tests on wide range of devices, as every device will need separate support by testing tool/framework, which is hard to achieve on extremely fast growing market of Android devices.
3. Approximation with *Android Power Profiles* is the most accurate method for collecting power data. This method consists in pulling per application statistics about component usage of mobile device from system service – `android.os.BatteryStats` [11]. This service logs time of component usage by applications (in milliseconds) in system journal. Most obvious solution is to approximate consumed power by formula: $q = I \times t$, where I is average component power drain in mA and t is the time of component usage is ms. OEMs of Android devices have to provide file with component's battery consumption

information (Android Power Profile) and ship it with firmware of devices. Google provides the recommended approach for collecting this data. This file with Android Power Profile placed to overlay file system before building runtime for the specific device in following location

`device///frameworks/base/core/res/res/xml/power_profile.xml` [12]

and available in runtime via resources of `framework-res.apk` package.

Therefore, Android platform do not provide suitable instruments for precise power measurement. The BatteryManager API can be used in simple situations when it is only necessarily to compare values obtained by running the same scenario, as this method allows comparing in terms of “more” or “less” energy were spent. Nevertheless, making two different measurements under different loads using BatteryManager API it is hard to exactly define the difference between the obtained values.

The Linux Kernel has a mature infrastructure for providing different information about power consumption and it is good candidate to be the generic approach for measuring power consumption, but it is originally was designed on PC platform, where powerful interface – ACPI – is available and, what is more important, supported by hardware [13]. On ARM and MIPS devices, which is the biggest part of Android devices present on market [14], there is no such support to provide rich information about battery state from hardware. As a result, many Android devices are not supporting this method – system files are empty or not present in virtual file system. On the other side, Intel provides the tool for precise monitoring power consumption on Intel x86 based Android devices [15]. However, the number of Intel x86 based devices through Android platform is not very significant [16].

The most practically applicable method, approximation with Android Power Profiles, relies on the quality of Power Profiles. Experiments show that

standard Power Profiles shipped with devices contains big fault and sometimes completely irrelevant (e.g., some users reported practically impossible values in Power Profile provided by the vendor's firmare: http://forum.xda-developers.com/showthread.php?t=1732722). In other cases firmware update cause significant permutations in Power Profile due to unknown reasons [5].

There are many existing methods for energy-consumption profiling, but measured values cannot be compared between different devices and measuring environments until we have precise Power Profiles. Getting accurate values of energy-consumption in various scenarios will allow doing comparison across many devices and applications and tracks the absolute numbers of consumed energy. Such data could be used for decision making in field of applications refactoring (process of improving code quality [15]) – refactoring, of course, should be done if profit is major compared to resources needed for refactoring.

For collecting precise power data on the Android platform, it is necessarily to develop a method of automated collecting the average energy-consumption values for separate components of mobile devices. This will allow improving power-measuring techniques on Android Platform.

Main components include (according to what components usage is tracked via `android.os.BatteryStats` service): Display, Bluetooth, Wi-Fi, DSP, GPS, GSM/UMTS, and CPU. Almost all devices may operate in different states (e.g. display has many levels of backlight, power consumed on every level is different). However, some devices are not present in the list. Android OS itself, for instance, do not track GPU 3D accelerator usage, so for games energy profiling other techniques should be applied (for example, Trepn Profiler diagnostic tool by Qualcomm Corp. [16]).

1.2 Approach

In this thesis, we will use an offline (hardware-based) approach for measuring average power consumption for each device component based on battery current measuring, what will lead to deriving energy profile for concrete device. Software on devices will execute series of predefined tests (*component test scenarios*) against certain components of mobile devices, while a digital measuring device will be collecting power consumption of the device's component (in mA). We will call derived power model "Energy Profile" to distinguish it from "Power Profile", which is built-in Android entity. It is also possible to supply Energy Profiles, which are not following Android power model (i.e. define other reference test scenarios).

In real world scenarios, components of device cannot operate fully isolated - the result of every measurement is aggregated power consumption of number of device components. The approach recommended by main vendor of Android platform, Google Inc., is just to subtract "standby" energy-consumption of device from energy consumption of devices in scenario when certain component is loaded on certain level (Power Profile) [12]. However, some device components always operate and we cannot simply switch them off (CPU in our case). For calculating power consumption of such components it may be possible to solve algebraic linear equations system, consisting of sum of power consumptions of number of components and total power consumption of device in different scenario. Another issue, which need to be solved, is application isolation – we cannot guarantee that CPU is not used by other applications (e.g. background services [17]). Therefore, such calculations can be not very precise and this hypothesis and obtained values should be validated.

To validate calculated Energy Profile we are proposing the following approach:

1. Estimate device battery lifetime in certain (fixed) test scenario (*validation test scenario*) using obtained Energy Profile
2. Perform chosen test scenario until battery is fully drained and determine the real life battery lifetime
3. Compare estimated lifetime with derived battery lifetime

If estimated time is between 90-110% of real lifetime then Energy Profile is "Good". Otherwise, the test, in which this value was obtained originally, should be reworked until we will get reliable values, which fit this requirement. Such validation tests could be done against provided by vendors Power Profiles to determine how much the Power Profile were improved (or not) for certain device.

Accordingly, the expected results of research in this work are divided into three parts:

1. Developing component test scenarios for measuring power under different components load
2. Choosing hardware for measuring current and developing software for automated tests running
3. Developing validation test scenarios and measuring the "precision" of derived Energy Profiles, reworking component test scenarios if needed

However, it is not guaranteed that all three parts are fully covered by this thesis due to unexpected limitations of Android API that make things harder to measure.

1.3 Related Works

There are number of existing works in the field related to the Android Power Consumption topic. This section gives a short overview of the works that was used as an inspiration for this master thesis.

1.3.1 PowerTutor

PowerTutor [20] an application that was developed by University of Michigan Ph.D. students Mark Gordon, Lide Zhang and Birjodh Tiwana under the direction of Robert Dick and Zhuoqing Morley Mao at the University of Michigan and Lei Yang at Google [21]. It is able to indicate the power consumed by major system components such as CPU, network interface, display, and GPS receiver and group this power consumption by the appropriate applications. The primary goal of the application is to be able to track power consumption changes after modifying the application architecture and implementation details.

PowerTutor uses its own power consumption model built by direct measurements of the defined device power management states. This model generally provides power consumption estimates with 5% fault. This is quite precise; however, this model was built only for the HTC G1, HTC G2 and Nexus One Android mobile phones.

The last update of the application was made in April, 2013. However, the commit history on the GitHub repository currently does not look very active and now the project looks abandoned by its creators. The application is an open source and the source code is available at GitHub [22].

1.3.2 Little Eye

Little Eye [23] is a commercial profiler tool, aimed mostly on Android Platform. In particular, it provides a functionality for monitoring power consumption of certain applications installed on the Android device. It consists of two parts - the agent application that should be installed on the Android device and the desktop application that connects to the agent on the Android device and analyzes stream of telemetry from the device [24]. The user interface of the application presented on the screenshot (see Figure 2. Little Eye).

There are two power models available in the tool – first is based on the reference power consumption of the Nexus One Android phone and the second is based on the SAMSUNG Galaxy Nexus Android phone. Power models use internal Energy Profiles obtained by offline measurements [25].

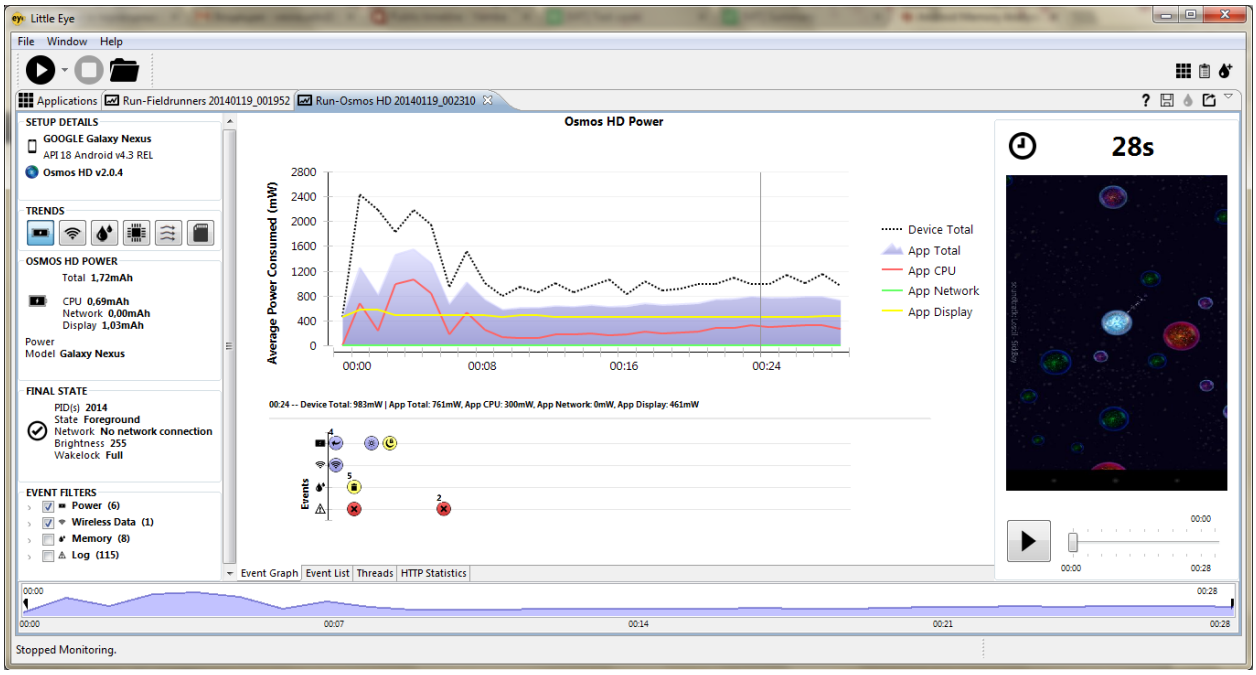


Figure 2. Little Eye

This is good demonstration that the Energy Profiling is an actual topic. However, this application is closed source and was selling using subscription model, i.e. to use this app it was needed to pay every year. The situation even worse now, because the company was acquired by Facebook [25] and now application is available only for prior customers – even the trial version of the application is not available for download anymore.

1.3.3 General Questions

There are also were a group of researchers from the University of California supported by SUPINFO International University [27]. The overview of the work done by this group is available through the video record of the presentation made by Frank Maker, Eric Jung, and Yichuan Wang [28].

Through the talk, the following questions were discussed:

1. Mobile Architecture – speaker was focused on explaining the differences between the best practices of programming for desktop platforms and mobile platforms
2. Measuring Power/Energy - difference between energy and power was discussed, differences in battery technology and why it is not expected to have a significant improvement of the batteries in near future
3. Software Optimizations - different ideas were discussed that may help to lower the power consumption of the mobile applications

The slides of this talk are available at SlideShare (hosting provided by LinkedIn) [29].

1.4 Work Packages

Roughly, this thesis may be divided into four main parts – choosing hardware and investigating the ways to collect measured values, investigating possibilities of the Android Platform to control power states of the mobile device, implementing software helpers for testing automation and analyzing results.

1.5 Structure

This thesis contains five main chapters – Introduction, Foundations, Setup Test Environment, Results and Conclusion. Each chapter has short announce at the beginning and divided into smaller sections. Some sections may have sub-sections.

2 *Foundations*

This section describes the mandatory prerequisites, which are needed to develop the system to derive Android Energy Profiles. The following topics are required to be explained:

1. Electricity Fundaments (“2.1 Battery Capacity”, “2.2 Measuring Voltage & Amperage”)
2. Hardware Fundaments (“2.3 Choosing Hardware for Test Environment”)
3. Discussing Power Model (“2.5 Energy Profiles”)
4. Technical Fundamentals (“2.6 Java Internals”, “2.7 Android OS”)

Information available in this section may be referenced in next sections. It is strongly recommended not to skip this section.

2.1 Battery Capacity

A battery's *capacity* is the electric charge, which can be deliver by battery at the certain voltage level (usually it referred as “nominal voltage”). The more electrode material contained in the cell the greater its capacity. A small cell has less capacity than a larger cell with the same chemistry, although they develop the same open-circuit voltage [18]. Capacity is measured in units such as amp-hours (Ah), or milliamp-hours (mAh). The *rated capacity* of a battery is usually expressed as the product of 20 hours multiplied by the current that a new battery can consistently supply for 20 hours at 68 °F (20 °C), while remaining above a specified terminal voltage per cell. For example, a battery rated at 100 Ah can deliver 5A over a 20-hour period at room temperature [19]. Batteries that are stored for a long period or that are discharged at a small fraction of the capacity lose capacity due to the presence of generally irreversible *side reactions* that consume charge carriers without producing current. This phenomenon is known as internal self-discharge. Another effect that is important when doing measurements

with batteries is that full-charge capacity is decreased with time. This may be considered by noticing the voltage of fully-charged capacity. Usually batteries in mobile devices should be replaced after two years of exploitation [20].

2.2 Measuring Voltage & Amperage

There can be a difference when measuring a battery voltage when the battery is under load and not under load. When a battery is under load, it is connected to the circuit in which it is intended to be used and the circuit or device is turned on. For example, a mobile phone battery is under load when it is installed in a phone and the phone is turned on. For devices such as cell phones, which do not draw much current from the battery, the battery voltage can typically be accurately measured when the battery is not under load. However, for larger batteries in which the current draw can be higher, such as car batteries, the battery voltage can drop dramatically when it is under load. To measure voltage of the battery with voltmeter it is possible simply make a circuit with battery and voltmeter [21].

In order to measure the current flowing, we need to connect a load to the battery (this means that battery should installed and device powered on) and connect an ammeter *in series* with the load. This measurement will give us the current flowing and not the total producible current of the battery. Such measurement is referred to as *drained capacity*. In other words, here capacity viewed as the amount of time a battery can put a given current. If take out the battery we can observe 4 or 3 pin connector. In this work we are interesting only in *power contacts* which is usually marked as plus (“+”) and minus (“-“). The rest battery contacts (one or two) are management communication or/and temperature sensor contacts accordingly.

In order to *constantly* measure amperage it is required to synchronize time between Android device, which is under measurement and measuring

device. Approach used in this thesis is following: every test logs start time and finish time, while measuring device constantly measures the amperage with fixed interval. After finishing the tests, two logs are combined with each other, which allows to know amperage, load and timing at the same moment.

2.3 Choosing Hardware for Test Environment

To measure flowing current on operating cell phone during continuous period of time the digital ammeter is needed. In order to keep multiple values of multiple measurements in some database ammeter with digital interface is preferred (e.g. USB or RS-232). Another two important things about choosing measurement device to perform experiments described in this master thesis is ability to manipulate device programmatically, so, such device should have a public and documented API to control it and device measuring precision. Flowing current in cell phones according to specification from the vendors varies between ~5mA in standby modes up to 300-350 mA in full loaded scenarios.

There are many professional tools, like Moonson Power Monitor (~750,- EUR) or Rigol DS1052E (~360,- EUR) are available (approximate prices are valid for the time of writing this thesis). They provide many options for doing electrical equipment measurements. However, they are complicated and integrating them into custom measurement system requires a significant effort due to commercial closed source software for manipulation. In this thesis, we will use tiny specialized device, which is very cheap and has only one function (to measure amperage) and open source management software available. For measurements in this work, Yoctopuce Yocto-Amp [22] device is used.

2.3.1 About Yoctopuce

“Yoctopuce is a company based in Geneva, Switzerland. It has been founded by three engineers with the intent of enabling anyone to create

simple systems to automate daily tasks, implement original ideas or simply build home automation gadgets" [23]. Yoctopuce products include many different types of devices: electrical sensors, environmental sensors, actuators, displays, etc. All devices may be connected with another device such as PC with USB interface and have internal flash memory to memorize measurement results.

The software toolbox called *VirtualHub* is available for Yoctopuce USB devices [24]. It allows to:

- configure and test Yoctopuce devices
- remotely control Yoctopuce devices through network
- control Yoctopuce devices with languages which do not provide a direct access to USB devices, such as JavaScript and PHP

It can either be used in command-line, or started as a service/daemon. The *VirtualHub* software is available for Windows, Mac OS X and Linux (both Intel and ARM). It can be freely downloaded from Yoctopuce website. For unmanaged languages such as C/C++ native libraries available and allow to control devices directly without using VirtualHub middleware [25]. Also, there is so-called "Command Line API" available. This API consists of pack of precompiled native executable binaries, which have only one function, i.e. they represents one function from VirtualHub. Part of this API, `YCurrent` application, is used in this thesis to communicate with Yoctopuce device.

Documentation for both VirtualHub software and API libraries is also available for free. Example usage of YCurrent consists from the following call in terminal: `C:\> YCurrent.exe YAMPMK01-12C90.current1 get_currentValue`. Here `YCurrent.exe` is Windows binary file, `YAMPMK01-12C90` is logical name of connected device (serial number by default), `current1` is logical name of the

sensor and `get_currentValue` is API function, which returns measured amperage [26].

2.3.2 Yoctopuce Yocto-Amp USB Electrical Sensor

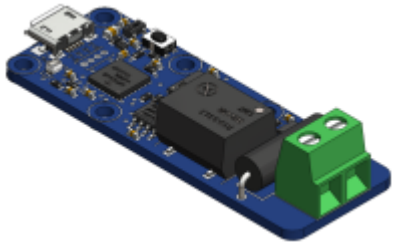


Figure 3. Yoctopuce Yocto-Amp (taken from [22])

Yoctopuce Yocto-Amp USB Electrical Sensor is a digital ammeter that allows you to measure current automatically. It can provide quite precise digital measures (2 mA, 1%). It works with direct current (DC) as well as alternating current (AC) for which it provides the RMS value (5 mA or 3%) [22]. It can be connected to the PC via USB and accessed programmatically directly via native API libraries using languages such as C/C++. In addition, it may be accessed with Java/Python/PHP applications using VirtualHub middleware (or Command Line API). Such technical characteristics is fulfill needs for measurement experiments of this master thesis. It costs 60 CHF and is available online via official Yoctopuce shop.

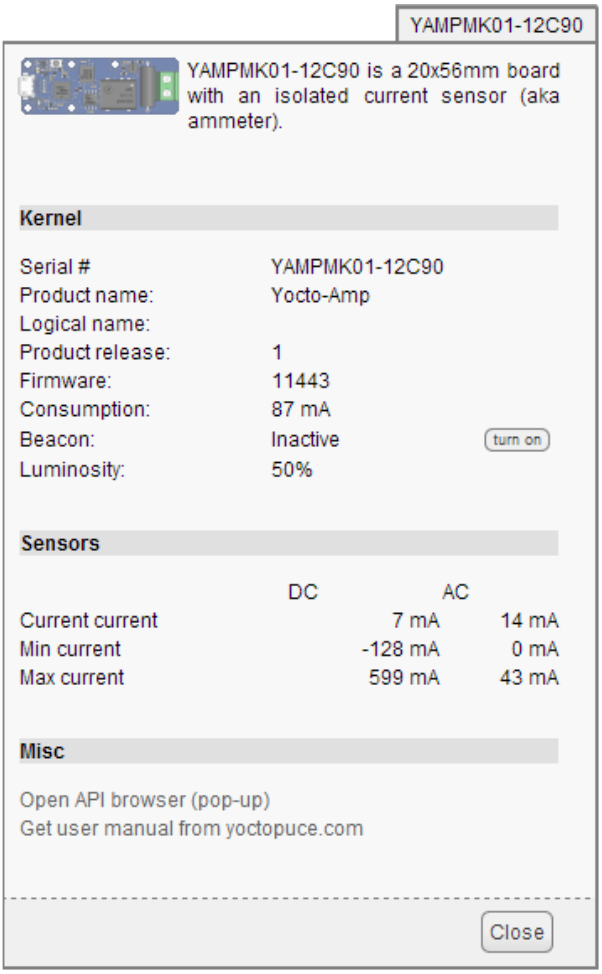


Figure 4. VirtualHub Software

To start measuring current, it is required to connect "-" contact of battery with "-" contact of Yocto-Amp and connect the "-" contact of the phone with the "+" contact of Yocto-Amp. If start VirtualHub and go to the http://localhost:4444 and then choose the device, the current current values may be observed. Example of such values are on Figure 4. VirtualHub Software.

There is example of using Yocto-Amp for measuring amperage of Nokia 105 phone (probably, the cheapest cell phone in the world). For this example, we will use `YCurrent` binary from the Command Line API and simple Windows Batch script:

```
1. @echo off
2. :loop
3. YCurrent.exe YAMPMK01-12C90.current1 get_currentValue
4. goto loop
```

This script produces the following output (amperage in mA after “equals” sign):

```
OK: YAMPMK01-12C90.current1.get_currentValue = 0
OK: YAMPMK01-12C90.current1.get_currentValue = 4
OK: YAMPMK01-12C90.current1.get_currentValue = 13
OK: YAMPMK01-12C90.current1.get_currentValue = 23
OK: YAMPMK01-12C90.current1.get_currentValue = 26
OK: YAMPMK01-12C90.current1.get_currentValue = 46
OK: YAMPMK01-12C90.current1.get_currentValue = 67
OK: YAMPMK01-12C90.current1.get_currentValue = 83
OK: YAMPMK01-12C90.current1.get_currentValue = 87
```

The output is easy to parse programmatically, which allows us to pipeline the measurements into database (with timestamps). In this case, we will be able to trace the current values at any moment of time and confront these values with profile of the load on device.

2.3.3 Connecting Ammeter to the Phone

It is good idea to use the plastic battery adapters for connecting the ammeter to device’s battery. The design models of such adapters for the test phone, HTC Desire, are given on Figure 5 and Figure 6. However, three attempts of printing these models on the MakerBot Replicator 2 [39] was made and none of them give an adapters of quality good enough to provide robust electrical contact.

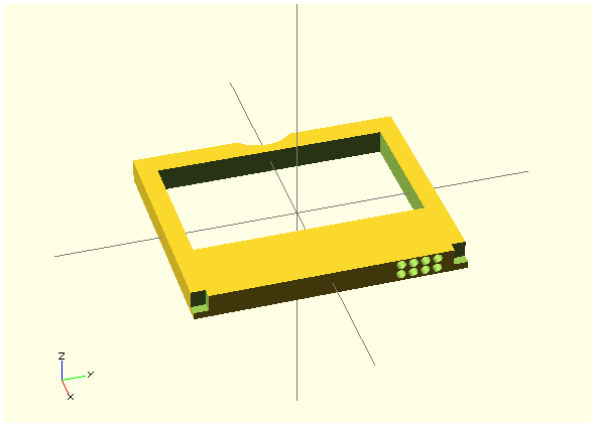


Figure 6. Battery Stub 3D Model

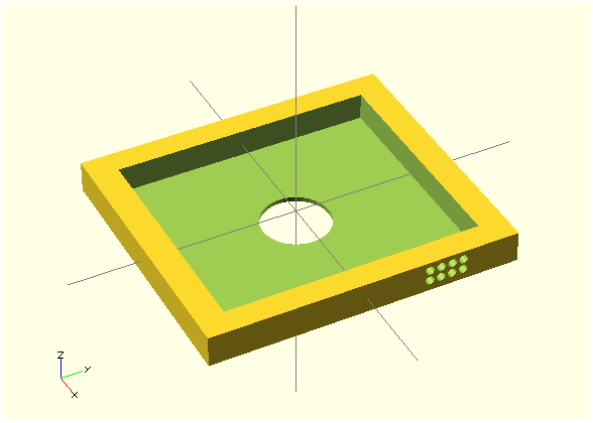


Figure 5. Battery Holder 3D Model

In this work the ammeter connected directly in series (there is no direct contact between battery and corresponding battery contact of the phone itself) to the "+" contact of the battery (see photo on Figure 7).

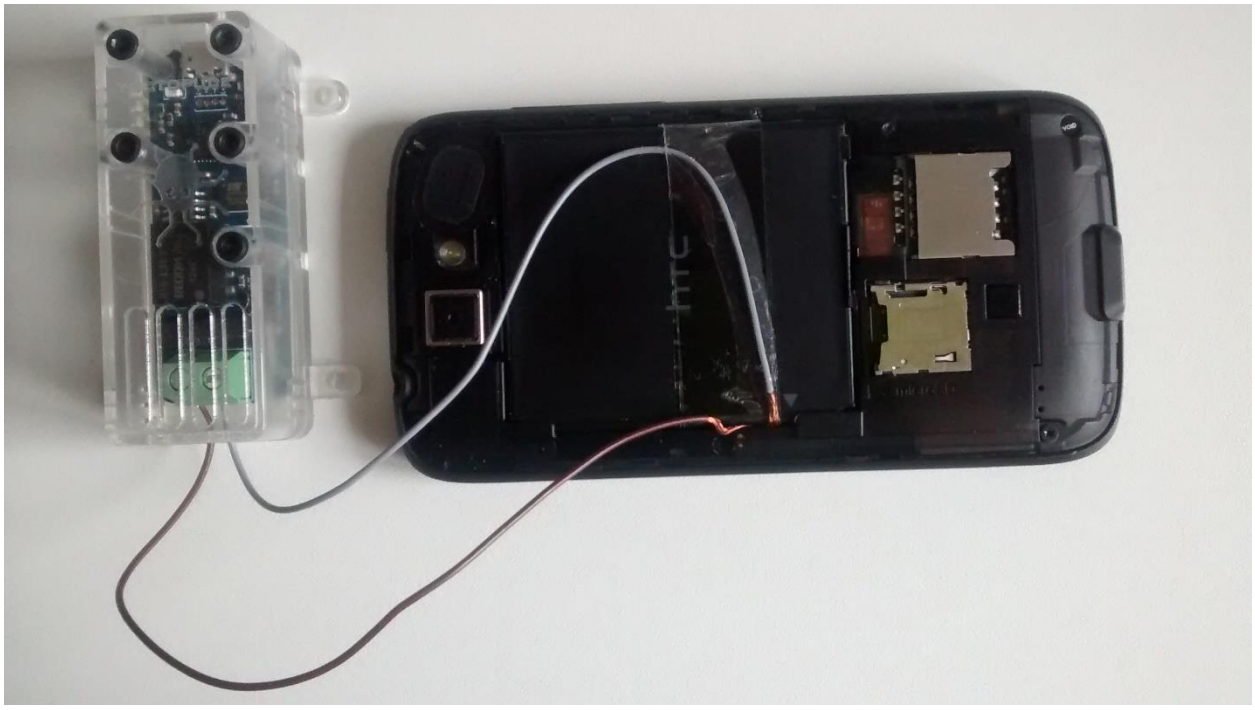


Figure 7. Yoctopuce Yocto-Amp Connection

2.4 Benchmarking

Benchmark is the act of running some program to measure some kind of performance. In our case, the device benchmarking is understood as the process of running predefined set of trials to derive Energy Profile under different load profiles. To let results be more precise and consistent, the same

trial should be run multiple times. In our case, the device load (power state) is fixed for a certain amount of time (e.g. 1 min.) and during this load, measurement software is continuously pulling the current amperage values from measuring device and store this amperage values into database.

To minimize the influence of the random factors, after analyzing of the values trend, some time series values for certain load profile maybe filtered out. After that, the average and the standard deviation across rest results is re-calculated. However, such speculations on the measured values may be modified in any way, according to the goal of the measurement session. Looking for the best way of calculating the average across the time series values in power benchmarking case is subject for the separate research and is not covered by this thesis – the simple arithmetic average across whole time series is used.

2.5 Energy Profiles

In this thesis, sometimes, the term “Energy Profile” is used. “Energy Profile” comparing to the Android Power Profile is defined in more general way and refers to the average power consumption in *any user-defined scenario*. This means that this value is not bind (yet) to any determined power model.

Nevertheless, the main goal of this thesis is to provide the way of producing Android Power Profiles, using of the “Energy Profile” term made many statements from the research applicable to the measurements for another power models.

2.5.1 Android Power Profiles

The power profile is where the device manufacturer needs to provide current consumption values for various components and their states in order to approximate the actual battery drain caused by these components over

time [12]. Power consumption of components is specified in milliamps, and can be fractional specifying microamps.

Usage of Power Profiles to approximate power consumption is straightforward: using tracked by Android Framework battery per-application statistics it is possible to get the usage time before the test, run a test in which energy consumption should be measured, get the new usage time, subtract first usage time value from the second and multiple the time by average component consumption from Power Profile. This approach is used in Android operating system itself in "*Settings*" application. For instance, to attribute the cost of keeping the display on for a duration of time, the framework gathers brightness levels and times spent at each level (quantized to some number of bins). The power profile values specify how many milliamps of current are required to keep the display on at minimum brightness and at maximum brightness. The time spent at each brightness level can then be multiplied by an interpolated display brightness cost to compute an approximation of how much battery was drained by the display component. Here is an example of Android Power Profile from ASUS Nexus 7 (2012) device running Android 4.4:

NONE	CPU_IDLE	CPU_AWAKE	CPU_ACTIVE
0.0	3.8	54.6	100.0
WIFI_SCAN	WIFI_ON	WIFI_ACTIVE	GPS_ON
100.0	2.9	3.1	29.7
CPU_SPEEDS	BT_ON	BT_ACTIVE	BT_AT_COMMAND
[107.0, .., 148.0]	1.4	14.0	0.0
SCREEN_ON	SCREEN_FULL	RADIO_ON	RADIO_SCAN
256.0	318.0	1.2	1.2
RADIO_ACTIVE	AUDIO	VIDEO	BATT_CAPACITY
71.5	14.1	54.0	3260.0

Table 1. Android Power Profile - ASUS Nexus 7

Note: “BT_” prefix and the “BLUETOOTH_” prefix refers to the same values of the Android Power Profile. Sometimes, the short variant is used in table headers due to limited space of the page width.

In addition, Power Profile contains an array of CPU speeds (in KHz), on which processor can operate, but there is no way to query this information via internal API (only number of such steps, which, nevertheless, is enough for estimation). Also, some values is redundant, e.g. “Audio” & “Video” values are supposed to contains energy drained by DSP during audio and video playback, but there is no way to track activity time of these DSPs. It may be possible to build power model for such scenarios, if application itself will keep tracking usage of DSPs, but this approach is hard to generalize.

The recommended way is to measure the current (usually the average instantaneous current) drawn on the device at a nominal voltage. However, manufacturers of the devices are allowed to use provided by components suppliers’ values in device Power Profiles. This is not very accurate and leads to errors in Power Profiles and imprecise approximation of energy consumed by the applications.

Measuring the current drawn by components at nominal voltage can be accomplished using a bench power supply or using specialized battery-monitoring tools (such as Monsoon Solution Inc.’s Power Monitor [27] and Power Tool [28] software). However, this approach is simplified in this thesis – instead of measuring with special bench power supply, we are using real cell phone’s battery with in series connected ammeter.

Many examples of the Android Power Profile XML files may be found at the Git repository [31] of the Replicant project. Replicant is the open-source fork of Android source code, which tries to provide “free-as-speech” firmware for modern Android platform devices [32].

2.6 Java Internals

Usually Java applications run on reference JVM (core component of Java SE platform) implementation – HotSpot by Oracle Corp [29]. In case of Android, Android Compiler converts Java bytecode into Dalvik Executable Format for the Dalvik Virtual Machine [30]. Java compiler and both HotSpot and Dalvik have many known limitations. As we are going to deal with hidden APIs of Android operating system, some deep knowledge about Java Internals are required. Necessarily minimal Java background is provided in the next paragraphs.

2.6.1 Classloaders

Java Classloaders are the classes, which are responsible for loading classes into the Java Virtual Machine [31]. Usually classes are loaded on demand. This is means that class are not loaded until the Java application try to use some class which is not loaded yet. If class if not available in runtime and JVM will try to implicitly load the class, it will leads to the `java.lang.NoClassDefFoundError`. There are also few methods in JDK, which allow to explicitly try to load the class:

```
1. Class.forName(String className)
2. ClassLoader.findSystemClass(String name)
3. ClassLoader.loadClass(String name)
```

Calls of these methods in case of class absence will produce `java.lang.ClassNotFoundException`. There are three standard classloaders:

1. *Bootstrap* – implemented on the JVM level and does not provide feedback to the Java Runtime Environment (i.e. it cannot be controlled within runtime). This classloader maintains the loading of jars located in `$JAVA_HOME/lib`. Therefore, `rt.jar` (standard Java library) is loaded with this classloader. So, if you will try to obtain the classloader from JDK's classes, you will always get `null`. Alternatively, you may control set of classes,

which are bootstrapped by providing `-Xbootclasspath` command line option to the java binary at start.

2. *System Classloader* – implemented on the JRE level and can be obtained via `java.lang.Class.getClassLoader()` method. This classloader loads the classes, which are listed in `$CLASSPATH` environment variable. It is possible to control the loading of system classes with command line option `-classpath` or via system option `java.class.path`.
3. *Extension Classloader* – classloader for extensions. This classloader loads classes, located in `$JAVA_HOME/lib/ext`. It is possible to control loading of extensions via system option `java.ext.dirs`.

The important notice about classloaders, is that they are organized into *hierarchy*. The right to load the class recursively delegated from the inferior classloader to the most supreme classloader. This approach allows loading the class with the classloader that are most close to the base classloader. Therefore, rule of the widest scope of visibility is applied. Visibility scope understood as follows: every classloader keeps track of the classes, which were loaded by this classloader. Set of such classes forms the visibility scope.

In case when JVM need to run the code from some class, the process of locating any class (e.g. `MyClass`) may be described as follows:

1. System Classloader tries to find `MyClass` class in own cache
   1.1. If class found, loading is done
   1.2. If class not found, loading is delegated to the Extension Classloader
2. Extension Classloader tries to find `MyClass` class in own cache
   2.1. If class found, loading is done
   2.2. If class not found, loading is delegated to the Base Classloader
3. Base Classloader tries to find `MyClass` class in own cache

3.1. If class found, loading is done

3.2. If class not found, Base Classloader tries to load `MyClass` class

3.2.1. If loading is successful, loading is done

3.2.2. Else, the control goes to the Extension Classloader

3.3. Extension Classloader tries to load `MyClass` class

3.3.1. If loading is successful, loading is done

3.3.2. Else, the control goes to System Classloader

3.4. System Classloader tries to load `MyClass` class

3.4.1. If loading is successful, loading is done

3.4.2. Else, exception `NoClassDefFoundError` is generated

The important observation here is that classes are resolved in runtime (generally, so called "dynamic linking") and System Classloader have higher priority than other classloaders. In practice, this means if your Android application contains its own implementation of class, let's say, `com.android.util.PowerProfile`, it will never be loaded. Instead, the System Classloader will load the class with same name from Android Runtime shadowing your implementation, as it have higher priority. Therefore, it is possible to compile the Android application, which uses internal API's (official Android SDK lacks these classes) just by copying source code from AOSP sources to the application source folder (or even put implementation with empty stubs, but with same interface). In runtime, the correct version of Android Runtime from the device will be used to load the core classes.

2.6.2 Java Language Specifications – Constant Inlining

If use the approach for access the hidden APIs described in previousparagraph, it is important to mind, that implementation of this APIs is subject to change. For example, it is possible that vendor modified some code from AOSP with its own implementation. Problem comes to scene in this case – constant inlining. Java compiler (`javac`) always inlines "… `static final` …" variables in places of usage (except for instances of `enum` and `null`

references) [32]. To avoid this, it is suggested never use the constants directly. Reading the field value with Java Reflection API will solve the problem in this case (value will be obtained in run-time, and not in compile-time). However, it may slightly affect the performance of the Android application.

2.6.3 Java Reflection API

Reflection (synonym – type introspection) is the process, when application is able to track and modify own structure and behavior in runtime. Reflection allows retrieving information about fields, methods and constructors of the classes. It is possible to do transformations over the fields and methods. Reflection in Java used via classes in packages `java.lang` and `java.lang.reflect`. Using Java Reflection API, it is possible to [33]:

1. Determine the class of the object
2. Retrieve information about class modifiers
3. List all fields and methods of the class
4. Create instance of class, which name is unknown in compile-time
5. Get and set value on the field
6. Call the method, which name is unknown in compile-time

We will use reflection to retrieve the constants which is unknown in compile time (because of “fake” implementation of Android Internal API classes in compile time of our application) to avoid constant inlining. Example of Java Reflection API usage in this case might be the following:

```
1.    import com.google.common.base.Optional;
2.    import java.lang.reflect.Field;
3.    import static java.lang.reflect.Modifier.isStatic;
4.    public class ReflectionUtils {
5.     public static <T> Optional<T> getDeclaredFieldValue(Class<?>
clazz, Object obj, String fieldName, Class<T> retType) {
6.      Optional<T> res = Optional.absent();
7.      try {
8.       Field field = clazz.getDeclaredField(fieldName);
9.       if (isStatic(field.getModifiers())) {
```

```
10.       res = Optional.fromNullable((T) field.get(null));
11.      } else {
12.       res = Optional.fromNullable((T) field.get(obj));
13.      }
14.     } catch (Exception e) {
15.      // something wrong ☹
16.     }
17.     return res;
18.    }
19.   }
```

2.7 Android OS

Android is an open-source operating system maintained by Open Handset Alliance (Open Handset Alliance is a consortium of 84 companies, which aimed to deliver open standards for mobile devices) [34]. To develop, build, test and debug applications Android Software Development Kit (Android SDK) is available. Google Corporation leads maintaining of this SDK, provides support, and updates tools for developers. Unfortunately, not all APIs are exposed to third-party developers, there are exists a number of APIs which are "internal" and not available via Android SDK. Power Profile API is one of such APIs. It is used only by built-in application *"Settings"* which is shipped with generic Android image. We will need to use special hacks described in previous section to access this APIs, as we need compare derived Energy Profile with existing Android Power Profile presented on device.

2.7.1 Android SDK

Android Software Development Kit contains a number of tools for creating applications for Android OS [35]. These tools include IDEs (Eclipse-based Android Developer Tools and IntelliJ-based Android Studio), emulator (AVD – Android Virtual Device emulator based on QEMU project), debugger (ADB – Android Debug Bridge), tools for assembling Java applications (Android SDK) and C/C++ applications (Android NDK). Currently these tools available for Linux, Mac OS X and Windows platforms. Android compiler produces bytecode in `*.dex` *(Dalvik Executable)* format for

Dalvik Virtual Machine from the common Java code. Therefore, to create Android applications JDK (Java Development Kit) is also needed. Most Android versions support Java 6 and latest version (Android 4.4 KitKat) supports Java 7 natively. However, in practice all Java 7 features except "try-with-resources" [40] are known to work normally on all Android versions ("try-with-resources" feature requires support from the core library and this library cannot be updated separately for old devices).

2.7.1.1 Android Debug Bridge

Android Debug Bridge is the command line tool for connecting to the Android device or the emulator running Android OS using host development computer. It supports Windows, Linux and Mac OS X hosts [51]. It consists from three components:

1. client on the your development machine (`adb` binary)
2. server, which runs in background on development machine
3. daemon on the Android device (or emulator)

When `adb` command is invoked from the development machine's shell, firstly, it will check if server on development machine is up and running. If not, it will bootstrap the server and bind it to the TCP port. When the server up and running it listens for the incoming client requests. After receiving a command from the client, it will setup the connection with the device's background daemon to start communication with the device. Android Debug Bridge supports debugging with multiple devices simultaneously. The list of all connected to the development machine devices can be obtained with `adb devices` command. Another useful command is `adb shell` that allows you to connect to the remote shell of the Android device and execute commands directly.

On the Android OS version 4.2.2 and higher there is one additional step. After issuing first command from the development machine the Android OS will show a dialog asking to accept the new RSA key. This mechanism was introduced for security purposes and prevents "silent" control of the device via debug tools. It ensures that user is able to unlock device and accept the key. Import note that this mechanism is requires `adb` version 1.0.31 and higher (part of Android SDK Platform Tools 16.0.1 and higher) to debug on a device running Android 4.2.2 or higher. Otherwise, the Android Debug Bridge client will end up with `device offline` state on the development machine.

Figure 8. ADB prompts RSA key authorization

2.7.1.2 Android Recovery Utility

The *recovery* is special Android boot mode, which boot up to the text-based utility, which allows flashing the device with images packed in zip archives [37]. For example, you may find factory images provided by Google for their reference Android devices (Google Nexus series) on the Android Developers portal [38]. In general, such images may be downloaded from the vendor's websites. To flash the image with recovery the image zip file should be placed on the external SD card, which is then put into the Android device before booting to the recovery.

Recovery mode allows you even flash different operating system on the Android device (like Mozilla's Firefox OS) [39]. In addition, it allows performing some maintenance operations like cleaning cache partition, resetting permissions on the system partition, etc. There a number of popular community-driven recoveries with enhanced functionality available. The most popular of them is CWMR (ClockworkMod Recovery [40]) and TWRP (Team Win Recovery Project [41]). For flashing HTC Desire phone, which is used as main testing phone in this master thesis CWM recovery was used.

Figure 9. ClockworkMod Recovery

To reboot into recovery there are usually two options. First is to power on phone holding some predefined combination of keys (may be known from vendor's documentation). The second option is to use Android Debug Bridge to reboot the device into recovery mode by executing `adb reboot recovery` command from the development host machine.

2.7.1.3 Fastboot

Fastboot is the tool, which comes with Android SDK and allows flashing the partitions of the Android device from the development host machine (no SD card needed in general) [42]. It may be viewed as an alternative to the recovery mode for flashing Android OS.

Fastboot mode is useful to update the device's firmware without copying the image to the SD memory card or the internal memory of the device. In addition, fastboot is used to perform some device-specific operations, like unlocking the bootloader of the Google Nexus devices. To load the image with fastboot utility, first, you need to boot device into

fastboot mode. It may be accomplished by using special key combination (see the manufacturer's documentation for your device) or by using `adb` command: `adb reboot bootloader`. To check that device is connected to the development host the following command may be used: fastboot devices. To flash the partition of the device the following command format is used: `fastboot flash <partition> <partition>.img`, where <partition> is the partition you want to flash. Common partitions include *boot*, *recovery*, *userdata* and *system*. However, some devices may use different number of partitions. For example, HTC One X device contains more than 20 different partitions [58].

2.7.2 Activity Component

"An activity is a single, focused thing that the user can do" [59]. This, basically, means that Activity represents one screen of the Android application. However, there other options of using Activities – Activity may be used as floating window (via a theme with `windowIsFloating` set) or be a part of another Activity using the `ActivityGroup`. There are two important methods, which should be implemented while using Activity:

- `onCreate(Bundle)` – this is the method where Activity is usually is initialized. Here, the method `setContentView(int)` should be called to initialize the UI layout of the Activity.
- `onPause()` – in this method all changes made by user (e.g. data input) should be saved

To be able to start Activity using another Activities (i.e. to provide navigation path to this Activity) Activity should be presented in `AndroidManifest.xml` as corresponded `<activity/>` declaration.

Navigation between different Activities are managed with an *activity stack*. An activity has four states (description of states taken from [59]):

- If an activity in the foreground of the screen (at the top of the stack), it is active or running.
- If an activity has lost focus but is still visible (that is, a new non-full-sized or transparent activity has focus on top of your activity), it is paused. A paused activity is completely alive (it maintains all state and member information and remains attached to the window manager), but can be killed by the system in extreme low memory situations.
- If an activity is completely obscured by another activity, it is stopped. It retains all state and member information, however, it is no longer visible to the user so its window is hidden and it will often be killed by the system when memory is needed elsewhere.
- If an activity is paused or stopped, the system can drop the activity from memory by either asking it to finish, or simply killing its process. When it is displayed again to the user, it must be completely restarted and restored to its previous state.

These states may be described using the diagram (see Figure 10. Activity Lifecycle (taken from)).

For the benchmark application, it means that Activity should be avoided to contain the benchmark code, because any long-running operations may be interrupted due to Activity Lifecycle events. Therefore, other Android application components like *Services* should be used to run benchmark tests scenarios. However, some actions (like changing brightness level programmatically) may be done using only Activity (it guarantees that action was completed as response to user intent). Therefore, event-driven model are used to communicate between foreground application

components (*Actvities*) and background application components (e.g. *Services*).

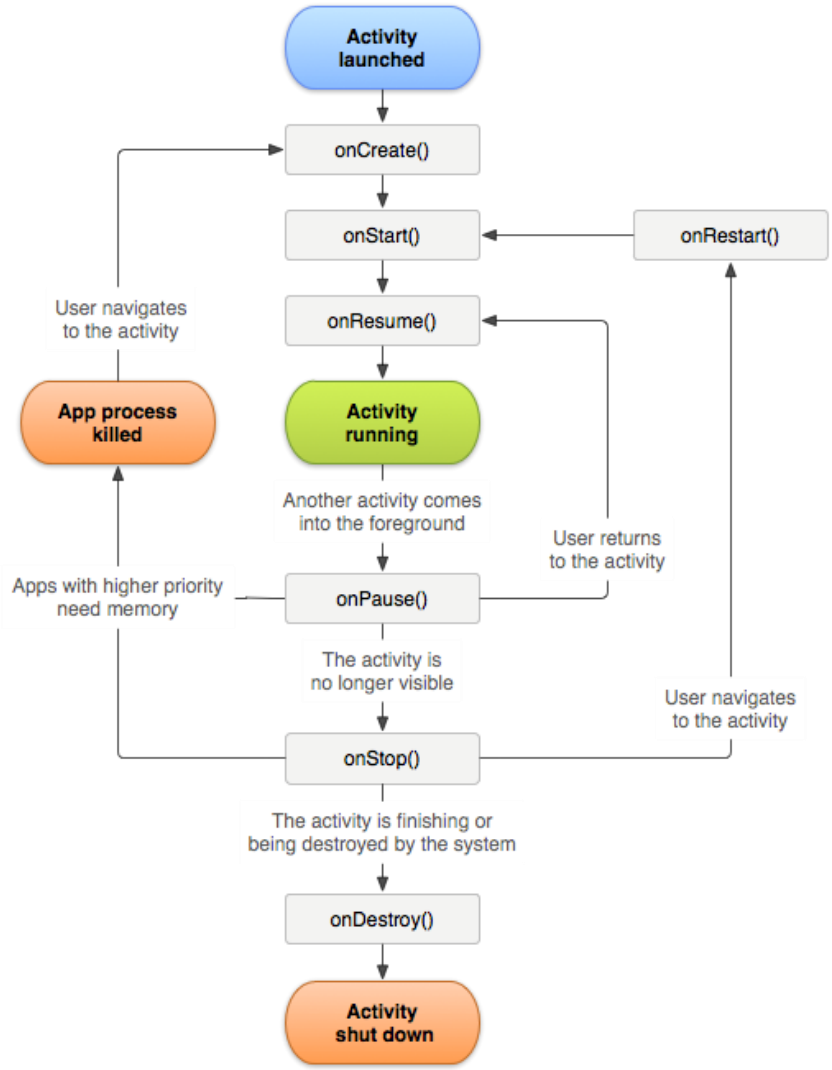


Figure 10. Activity Lifecycle (taken from [59])

2.7.3 Background Tasks (Services & AsyncTask)

There are two possible ways of executing long-running operations on the Android mobile devices – using of AsyncTask and using of Service classes.

The first solution is to use `AsyncTask` class. The main purpose of this class is to run long operations that are bound to `Activity`. By default, all methods of the `Activity` class are run on so-called *UI Thread*. In case of long-

running operations in these methods, it will cause the hanging of UI. In addition, if the method is taking more than 5 seconds, the Android OS will show the “Application Not Responding" (ANR) dialog allowing user to force close the application. To avoid this, the `AsyncTask` class should be used. It allows to perform background operations and send results back on the UI thread without having to manipulate threads manually. Ideally, it should be used for short operations (few seconds at the most). An `AsyncTask`, essentially, is defined by three generic types, called `Params`, `Progress` and `Result`, and 4 methods that need to be overridden called `onPreExecute`, `doInBackground`, `onProgressUpdate` and `onPostExecute`. It does not suite the scenario, when it is needed to keep threads running for a long periods of time. In such cases, it is recommended to use the plain Java APIs provided by the `java.util.concurrent` package such as `Executor`, `ThreadPoolExecutor` and `FutureTask`.

The second option for running background tasks is `Service`. The `Service` class is providing a way to maintain long-running operations in background and does not have any user interface [60]. Another application component may trigger a start of service and the service will continue to run, even if user switched to another application (and corresponding application’s `Activity` was destroyed). Service may be used in two ways:

1. *Started* – service is started, when another application component starts it by invoking the `startService()` method. Once started, service may run indefinitely, even if client component already destroyed. When the operation is done, the service should stop itself by calling `stopSelf()` method.
2. *Bound* – service is bound, when another application component binds it by invoking the `bindService()` method. A bound service provides a client-server interface that allows client components (i.e. Activity) to interact with the service, send requests, get

results (also via IPC if Service is running in separate process). Bound service is only running while it have alive clients bound to it. Multiple application components (i.e. activities) may bind to a service, however, if last application component is unbind the service is destroyed.

Like activities (and some other components), service component must be declared in application's `AndroidManifest.xml` file. The `Service` is represented with `<service/>` xml tag inside the `<application/>` xml tag. Also, including the xml `android:exported` attribute into `<service/>` tag and setting it to "false" prevents other applications from being able to start or bind to the declared `Service`.

Therefore, the `Service` application component if fulfill the benchmark applications needs for running test scenarios.

2.7.4 Alarm Managers

For creating benchmark for Android Platform, it is important to know how other applications may be started in the system. The class `AlarmManager` provides access to the system alarm services [61]. It allows the applications to be scheduled to run at some point in the future. When an alarm goes off, the specified `Intent` [62] is automatically broadcasted causing the subscribed applications to wake up (if application is not started, it will be started) and start processing the incoming `Intent`. In addition, Android's documentation says that "Registered alarms are retained while the device is asleep (and can optionally wake the device up if they go off during that time), but will be cleared if it is turned off and rebooted".

Therefore, rebooting the device before starting the benchmark minimizes the possibility of waking up of the third-party applications (it is also good idea to temporarily remove widgets from the main screen as they have corresponding activities, which is able to register such alarms).

2.7.5 SQLite Database

There are built-in database management system in Android Platform, which is essentially built on top of SQLite database [63]. SQLite is a fast, embedded database and it is designed to have a good performance on the mobile devices like and Android mobile devices.

The Android Framework provides few classes to work with SQLite database on the Android mobile devices. The first class is `SQLiteOpenHelper`. It provides a skeleton for maintaining the database file itself. `SQLiteOpenHelper` provides two convenient methods for doing so:

1. `onCreate()` – bootstraps the database from the DDL scripts (i.e. creates tables in case of database absence)
2. `onUpgrade(int oldVersion, int newVersion)` – provides a way to apply migration scripts in case of changing database schema between application versions (i.e. alters tables if they are stale)

Class `SQLiteOpenHelper` provides methods `getReadableDatabase` and `getWritableDatabase`, which returns the instance of the `SQLiteDatabase` class. This class provides all methods for doing CRUD operations with the database (i.e. `insert()`, `query()`, `update()`, `delete()`).

2.7.6 Using Internal and Hidden APIs

First, it is necessarily to explain how APIs are became inaccessible in Android SDK. This is achieved in very straightforward way: libraries on device actually consist of two files: `core.jar` and `framework.jar` that are found in `/system/frameworks/` directory on the device. The file `android.jar` (located in Android SDK platform directory `$SDK_DIR/platforms/platform-X/android.jar`, where `X` is API level, it can be 18 or 19 or any other number) from the Android SDK, which is used for building applications, contains only public APIs of those libraries (deployed on real devices). All implementations of methods in actual bytecode in `android.jar` file is

replaced with something roughly equal to the following Java code: "`throw new RuntimeException("Stub!");`". It is not necessarily for set of APIs of android.jar and libraries on devices (`core.jar` and `framework.jar`) to be equal. Vendors may modify any library and provide their own extensions. However, subset, which is "public" and exposed to the `android.jar`, should be compatible with reference Google's implementation. This is achieved via mandatory certification (if vendor wishes to use "Android" name and to be able preinstall Google Play Services on the devices) from Google. First step of this certification, Android Compatibility Test Suite checks compatibility of the core library [44].

Android has two types of APIs, which are not accessible from Android SDK. The first type is located in package `com.android.internal.*` and second is various classes across whole Application framework [45] marked with `@Hidden` annotation (removes Javadoc from resulting `android.jar` file) and `@hide` annotation (removes the class files from resulting `android.jar` file). For hiding device-specific internal APIs (the example of API which is contain such hidden parts may be *S Pen SDK* by Samsung [46]) there is also a tool called `mkstubs`, it is used for developing so-called "SDK Addons". However, in core AOSP project `mkstubs` tool is not used. Therefore, it is possible to remove these @hide annotations from the code or disable appropriate annotation processor from the build script and build your own version of `android.jar`, which will contain any APIs you want to access.

However, it is no very convenient to build Android SDK every time new version comes out. Slightly modified approach may be used – just to copy the whole source code of internal class to source tree of your application. This approach is easier to maintain by the developer.

2.7.7 Root Access

We need to perform some operations on the device require rights of the built-in *root* user. In UNIX operating systems, the superuser [47] may be named with any name include *baron* in BeOS and *avatar* in some other commercial UNIX distributives. However, in Linux and hence in Android superuser by convention named *root*. To promote the shell to the superuser shell `su` command is used.

In most stock Android images for the devices `su` binary is absent and password from *root* user is unknown. There are two possible scenarios for retrieving the Root Access. First, when bootloader of the phone is not locked (e.g. all Google's Nexus series devices), it is possible to use `fastboot oem unlock` command and then use `fastboot` utility to flash the modified kernel (flash custom `boot.img` to the *boot* partition of the internal memory). However many Android smartphones manufacturers are known to prevent access to the bootloader of the phone (for example, widely known HTC's technology *Secure Boot* which prevents loading images without digital signature [58]). In this case, only some known *vulnerability exploitation* is possible to retrieve the Root Access.

In this thesis, one of the phones included in benchmarks is HTC Desire. It is running original Android 2.3.3 stock ROM for developers, downloaded from HTCdev.com [48]. To get the Root Access for HTC Desire, toolkit provided by some anonymous hackers from this link is used: http://revolutionary.io/.

2.8 Implementation Technology Stack

This section covers additional tools and frameworks used for implementing the software built for this master thesis. In addition, the corresponding section usually provides the information why the particular tool was chosen and how it differs from the analogs.

2.8.1 The Scala Language

Scala is a multi-paradigm language, which runs on top of JVM [69]. It allows mixing object-oriented concepts and functional concepts in one place. In addition, it provides seamless interoperation with Java language. It means that any Java library may be used from the Scala language.

The measurement software we are going to implement is a kind of *ETL* [70] tool. In this case, using of the Scala language have significant benefits over the Java language due to rich Scala collections framework [71]. For example, consider the following Java code (it uses the Google Guava [72] library):

```
1.    FluentIterable.from(strings)
2.      .filter(new Predicate<String>() {
3.         public boolean apply(String string) {
4.           return CharMatcher.JAVA_UPPER_CASE.matchesAllOf(string);
5.         }
6.       })
7.      .transform(new Function<String, Integer>() {
8.         public Integer apply(String string) {
9.           return string.length();
10.        }
11.      });
```

In Scala language, the same can be done using following one-liner:

```
1. strings.filter(CharMatcher.JAVA_UPPER_CASE.matchesAllOf(_)).map(
   _.length)
```

Moreover, it can be dramatically improved reducing the number of iterating through the collection from two to one applying the small change to the code:

```
1. strings.filterWith(CharMatcher.JAVA_UPPER_CASE.matchesAllOf(_)).
   map(_.length)
```

Therefore, Scala collection framework allows writing less code in ETL programs because functions and laziness are first-class citizenships of the Scala language.

2.8.1.1 SBT

SBT (acronym for the *Simple Build Tool*) is the build tool written in Scala language and de-facto is standard for assembling programs written in Scala language [73]. The main benefit of using sbt is that it requires almost no configuration for simple projects. However, if addition features are needed for the project setup, it can be easily done because sbt uses plain Scala code to describe build definitions. Using of type-safe language (i.e. Scala) in build definitions dramatically decrease numbers of errors in these build definitions.

Another strong side of the sbt is supporting documentation generation using the `scaladoc` utility, which integrates with GraphViz [74] to generate diagrams in the documentation of Scala classes.

2.8.1.2 Scala Process API

Scala Process API is a wrapper around Java's `Process` and `ProcessBuilder` classes to provide more convenient and idiomatic Scala API for dealing with native processes in operating systems [75]. Basically, the API is divided into three parts (description taken from [75]):

- Indicating what to run and how to run it
- Handling a process input and output
- Running the process

While there is no need to know underlying Java classes to use the API, it should be taken into account which boundaries they impose to the API. For example, it is not possible to retrieve a *process id* for the executed process.

In our case, handling the process output is important thing. The described Scala Process API will be used to handle output of the native measurement software command-line tools. The I/O of the running process may be controlled using the `scala.sys.process.ProcessIO` object, which

should be passed to the code that runs the external process. It introduces the significant benefit over plain Java Process API, because the code, which is responsible for running external process and the code, which is responsible for handling process I/O are separated from each other (in Java API, the code responsible for running the external processes in the same time has to handle processes I/O).

2.8.2 HyperSQL Database

HSQLDB (*H*yper*SQL D*ata*B*ase) is the RDBMS written in Java. It provides a fast multithreaded and transactional database engine with small memory footprint. It may operate as in-memory and disk-based tables database, supports embedded and server modes [76]. It supports almost all features from the ANSI-92 SQL Standard [77]. It also supports many extensions to the Standard, including syntax compatibility modes, which provides a way to run a vendor-specific queries designed for other databases.

The main feature, which was taken into account while choosing embedded database engine for the measurement software tool was stored database format. Internally, HSQLDB used simple *database logs,* which consist of the sequence of the DML statements (`INSERT` and `UPDATE`). It make the process of exporting the measured raw data to other applications very simple, because all you need in case of relational databases, is to run the script against the database.

2.8.3 Gradle

As was already discussed, the SBT tool is used in case of building Scala programs. However, the software for Android Platform is still implemented using Java language. For assembling the Android package, the Gradle build system is used.

There are two build systems available for assembling packages for the Android – Ant [78] and Gradle [79]. The first build system, Ant, comes with

stable version of the official Android IDE – Android Development Tools, a plugin for Eclipse IDE [80]. However, in the meantime the primary IDE for developing for Android is brand-new Android Studio [81] based on IntelliJ IDEA [82]. It uses new project format that is based on the Gradle build tool. Therefore, the build system of choice in the meantime is Gradle.

Gradle combines the concepts well known by using Ant and Maven build tools like dependency management through Maven [83] and Ivy [84] artifact repositories and declarative style of build description files. For the build files, it uses domain-specific language based on Groovy [85] programming language.

2.8.4 Guava

Guava is an umbrella library project by Google for the Java language. It consists from following parts: collections, caching, primitives support, concurrency libraries, common annotations, string processing and I/O [72].

It is very useful library for programming for Android Platform, especially for working with collections. The architecture of the collection component were motivated by Java generics introduced in Java 5. Although generics may dramatically improve the productivity of programmers, the standard Java Collection Framework (JCF) does not provide sufficient functionality. Its complement Apache Commons Collections has not adopted generics in order to keep backward compatibility. Because of this, two engineers Kevin Bourrillion and Jared Levy developed an extension to JCF that provides additional generic classes such as multisets, multimaps, bimaps, and immutable collections.

In addition, the Guava library introduces some functional concepts to the Java language (functional paradigm will be available natively in Java language starting from the Java 8). It will help to keep the coding style consistent between the desktop measurement application written in Scala

language and the benchmark application for Android Platform written in Java language.

2.8.5 SL4J

SL4J [86] is a Simple Logging Façade for Java. It provides an abstraction for different logging frameworks allowing the end-user to provide the desired logging framework in *deployment* time. Due to unified SL4J API, the code that using logging itself is has no dependency on the logging framework, but it depends only on small `sl4j-api.jar`. If no implementation of SL4J is found in runtime, then application fallbacks to a no-operation implementation (this means that logging is disabled and stubs are used for the logging methods calls). It is a good practice to use SL4J for libraries and frameworks projects, because it allows the user of library/framework to choose the desired logging implementation and maintain the logging in convenient manner.

In our case, the *Logback* [87] logging implementation is used for the desktop measurement tool and *Android SL4J* [88] logging implementation is used for the Android benchmark application.

2.8.5.1 ScalaLogging

ScalaLogging is a wrapper library for *SL4J* and *Log4j 2* libraries to provide a lazy-style logging on top of these frameworks [89]. It considered being a good practice to avoid the corresponding methods calls if the given log level is not enabled:

```
1. if (logger.isDebugEnabled)
2.   logger.debug(s"Some ${expensiveExpression} message!")
```

It is also true for parameterized log methods of the *Logback* library:

```
1. if (logger.isDebugEnabled)
2.   logger.debug("Some {} message!", expensiveExpression)
```

However, this idiom heavily clutters the code and decrease the code readability. The ScalaLogging use the new feature of the Scala language

called *macros* (macros were introduced in Scala language in version 2.10) to provide log level check automatically. Therefore, the log level checks will be added to the following code automatically at compile time:

```
1. logger.debug(s"Some ${expensiveExpression} message!")
```

It significantly improves the readability by hiding ugly log level checks inside the macro.

2.8.5.2 SLF4J Android

Because Android Framework has its own logging framework built-in and does not implement the SL4J API, using of the libraries that are dependent on the SL4J API will lead to the falling back of these libraries to the SL4j no-operation logging implementation. However, logging provided by third-party libraries is a good source of information while debugging applications. The *SL4J Android* library provides a logging implementation that forwards all calls to the SL4J API to the logger provided by Android platform. This allows to libraries based on SL4J logging API to write their logging messages to the Android log circular buffer, and then these messages may be viewed using logcat [90] tool from Android SDK.

2.8.6 OpenCSV

OpenCSV is very simple parser for the CSV file format (essentially, comma-separated values) [91]. It will be used in some of the ETL operations of the desktop measurement software tool. The CSV format is considered to be used because of its simplicity and portability.

2.8.7 Apache POI

The Apache POI library [92] will be used for exporting the collected data from the measurement tool database to the Microsoft Excel [93] file format. The histograms and graphs provided in chapter 4. Results are drawn using Microsoft Excel.

3 *Setup Test Environment*

In this section, we are going to describe how component power consumption is measured in this thesis and how then Energy Profiles are calculated using the measurement values. These tests include scenarios, which collects data for calculating following values from the Android Power Profiles:

1. *CPU_IDLE*
2. *CPU_AWAKE*
3. *CPU_ACTIVE*
4. *BLUETOOTH_ACTIVE*
5. *BLUETOOTH_ON*
6. *GPS_ON*
7. *WIFI_ON*
8. *WIFI_SCAN*
9. *WIFI_ACTIVE*
10. *SCREEN_ON*
11. *SCREEN_FULL*

3.1 Defining Component Tests Set

Many tests (actually, all non-CPU tests) should be developed in a way to minimize upstream power-consuming dependencies. Here, by power dependencies we mean another hardware components that are involved into scenario, e.g. determining location with GPS requires using not only the GPS hardware itself, but also some efforts from the CPU. Therefore, measured consumption in the *GPS_ON* test is not pure GPS hardware power consumption, but a total of GPS hardware consumption and the CPU (at least). The open question is which CPU state consider as the “base consumption” in non-CPU tests. In this work, the *CPU_AWAKE* is always used, but in reality, this is most likely the intermediate state between the

CPU_AWAKE and *CPU_ACTIVE0*. To get the more precise results the following solution may be applied: each non-CPU tests should be ran in every CPU state. If the difference between the pure CPU test and the same scenario, but with powered on non-CPU component remains the same across different CPU states, then this difference – pure consumption of this, non-CPU, component. However, if the difference if different across CPU states, then maintaining the component loads the CPU and "base consumption" for this component should be assumed as the higher CPU power state than *CPU_AWAKE*.

The isolation of power consumers on the Android platform is a challenging task, because of the Android application lifecycle. Even if there are no any running applications at the start of the test, there is no guarantee that no application will be run simultaneously with the our Energy Benchmark. It leads to some unpredictable raise of the power consumption of the device and this methodology do not answers the question how to separate power consumption of the currently observable component from other possible "noise". The only attempt to solve the issue consists from filtering out most outstanding values from the measured series. However, it might be not legal for the components that might use adaptive algorithms. For example, all wireless devices will require more power if signal strength is low. Therefore, it is possible that if during *BLUETOOTH_SCAN*, *WIFI_SCAN* of other tests that involving radio devices some new devices with low signal will appear in radio ester it might cause short increase of power consumption of the scanning device to recognize the capabilities of the new device in radio ester [49]. In this thesis, it is assumed legal for radio devices to cause short raise of power consumption and the outstanding values are not filtered (however, it needs to be strictly proven).

Another technical question, which should be solved, is how to set up required power state on the Android device. Some operations (like a

switching off/on screen, various radio devices) are done using Android SDK functionality provided for the applications developers. However, operations as locking CPU frequency or changing screen brightness silently (not involving user interaction with the Android device) requires low-level operations with the Linux command line tools (Root Access is needed).

Tests, which cannot be automated (like *CPU_IDLE*) and tests that require a big effort to be automated (like *BT_ACTIVE*, *BT_AT_COMMAND*, etc.) may be done manually. In this case, the operator just sets up the phone using common "Settings" menu of the Android phone and starts the measuring session manually. However, this thesis focuses more on cases that can be automated in graceful manner providing a convenient Android software for measuring Android Power Profiles values. In addition, this software may be used as a framework for building Energy Profiles using different power models (not only reference Power Profiles).

All tests, except the screen ones running with screen and all components that are not subject of the test powered off. The benchmark application manipulates the screen-off timeout setting it to the minimal value and waits for the screen power off timeout. For running scenarios that are defined in Android benchmark application the following code snippet is used (Java code):

```
1. protected static void doWork(LoopWorker worker) {
2.  long startTime = System.currentTimeMillis();
3.  long estimatedEndTime = startTime + DURATION;
4.  long currentTime;
5.  long logTime = -1;
6.  do {
7.   worker.next();
8.   currentTime = System.currentTimeMillis();
9.   if (logTime == -1 || currentTime - logTime > 5_000) {
10.   sLogger.info("Benchmarking.. Current CPU frequency: {}",
   CPUMonitor.getCurrentFrequency());
11.   logTime = currentTime;
12.  }
```

```
13.  } while (currentTime < estimatedEndTime &&
     !Thread.currentThread().isInterrupted());
14. }
```

Different implementations of the “loop test” should provide their own implementation of the `LoopWorker` interface to do some work in cycle, until time is up.

3.1.1 CPU Benchmarks

First, CPU benchmarks set contains the *CPU_IDLE* scenario, which cannot be automated by Android application. Any application activity like logging “Benchmark Start” timestamp (we need such timestamps to bind measured values to the test scenarios) will cause wake up and device will not be in desired state (it will be in *CPU_AWAKE* state instead of *the CPU_IDLE*). To measure the *CPU_IDLE* value manually the device, which currently is in sleep mode needed. Android device need some time to go sleep after screen if off. In addition, some applications can cause No-Sleep bugs [1], which prevents the device from going to sleep. To set up a trustworthy test, device is used immediately after factory reset [49] is done. Therefore, no other applications is installed and this minimizes the risk of No-Sleep bugs. It needs to be pointed, that it is possible that stock Android image contains software with such bugs, so the measured values anyway should be checked for adequacy. To prevent running any applications in response to the system events on the phone with applications installed the *Autostarts* [62] Android application may be used. It allows to forbid running the applications in background. However, the application requires Root Access (see section 2.7.7 Root Access).

The next major value is *CPU_AWAKE,* this value is important because it has downstream dependencies, e.g. other Power Profile values depend on how precise *CPU_AWAKE* was measured. This happens because any other tests require CPU activity to perform operations to maintain optional devices

(Wi-Fi, Bluetooth, and GPS). In addition, the energy benchmark itself requires some little efforts from the CPU for switching hardware states and logging timestamps to the database. *CPU_AWAKE* scenario defined as following: the CPU frequency is locked using ability of the Linux Kernel to control CPU governor directly with following commands in the shell:

```
# echo temporary > /sys/power/wake_lock
# echo userspace >
/sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
# echo 245000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq
# echo 245000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq
# echo 245000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
```

To undo the effect of these commands:

```
# echo temporary > /sys/power/wake_unlock
# echo ondemand >
/sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
# echo 998000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq
```

Note: if execute these commands “as-is” on other device the desired effect may not be achieved. This particular command is applied *to the first logical core* of the CPU (`cpu0`). Therefore, to control power state of multi-core CPU you need to issue these commands for each available core of the CPU (replacing `cpu0` with `cpu1`, `cpu2`, `cpu3`, etc.).

Android application on the Android device reads the current values of this Linux Kernel Nodes and saves them (to restore original values after test end), issues these commands and then writes journal timestamp “Benchmark Start” to the built-in Android SQLite database. Then it runs empty cycles of `Thread.sleep()` Java instructions for the predefined amount of time (all tests in this thesis use a *one minute* time window for collecting current measurements). After time is ran out, application writes “Benchmark End” timestamp to the database and restores the power state, which was preserved before starting the test. The code of the *CPU_AWAKE* scenario `LoopWorker` cycle is following:

```
1. public void next() {
2.  counter++;
3.  ThreadUtils.sleep(3);
4. }
```

In general, this test case might be described using the following preuso-code:

```
1. -> Disable Wi-Fi
2. -> Disable Bluetooth
3. -> Acquire Partial Wake Lock
4. -> Turn Off Screen/Wait for Screen Off Timeout
5. -> Lock CPU Frequency at Minimum Level
6. -> Run the LoopWorker
```

The *CPU_ACTIVE* test scenarios simulating the load doing some calculation job (calculating the digits of the Pi number) with predefined CPU frequency. Actually, *CPU_ACTIVE* is not a scalar value, but a set of values that can be viewed as a vector. These values correspond to the different operating CPU frequencies. It is supposed to match the all-possible CPU frequencies of the device's CPU. However, vendors may not follow this rule. For example, our test phone, HTC Desire, has following CPU frequencies in the Power Profile XML file (obtained via using Java Reflection API on the `com.android.internal.os.PowerProfile` class): [245000.0, 384000.0, 460800.0, 499200.0, 576000.0, 614400.0, 652800.0, 691200.0, 768000.0, 806400.0, 844800.0, 998400.0]. However, if you read the states that are reported by the CPU governor driver you will get different CPU frequency series:

```
$ cat /sys/devices/system/cpu/cpu0/cpufreq/stats/time_in_state
```

The output if this command is following:

```
245000 4940688
384000 56809
460800 56067
499200 55898
576000 53777
614400 48705
652800 48743
691200 48672
768000 48816
```

```
806400 49540
844800 49031
883200 732
998400 86111
```

Surprisingly, CPU governor driver reports the extra value that are not presented in the Power Profile XML file (883200 KHz). To preserve the structure of derived Power Profile same as the default Power Profile (to be able to do a comparison) we also skip measuring this extra value (according to the output on previous page, it is used very rarely). However, *on other devices* this may lead to noticeable discrepancy, when calculating consumed energy using Android Power Profiles (the solution for this is presented in 4.1 CPU Energy Profile section, it will be shown that for CPU power consumption linear approximation will give good result). To measure CPU_ACITVE activity power consumption the simplest solution is to use endless mathematical calculations. For debugging purposes (it easy to notice in this case that the CPU is *actually* works faster and "setup" part of the test was done correct) calculation of the Pi number is used. Of course, any other endless calculation activity (without waiting for I\O) will suite fine. The example of the `LoopWorker` for the *CPU_ACTIVE* scenario (this code calculates terms in the Madhava–Leibniz series [51] that very slowly approximates the Pi number):

```
1. public void next() {
2.  counter++;
3.  BigDecimal currentTerm;
4.  if (counter % 2 == 0) {
5.    currentTerm = NEG_FOUR.divide(oddTerm, SCALE,
   RoundingMode.HALF_UP);
6.  } else {
7.   currentTerm = POS_FOUR.divide(oddTerm, SCALE,
   RoundingMode.HALF_UP);
8.  }
9.  oddTerm = oddTerm.add(POS_TWO);
10. piNumber = piNumber.add(currentTerm);
11.}
```

In general, the scenario itself may be described using following pseudo code:

```
1. -> Disable Wi-Fi
2. -> Disable Bluetooth
3. -> Acquire Partial Wake Lock
4. -> Turn Off Screen/Wait for Screen Off Timeout
5. -> Lock Frequency at Desired Level
6. -> Run LoopWorker
```

This scenario should be executed for each CPU frequency that is mentioned in the original Power Profile. The average of each measurement session will produce the *CPU_ACTIVE* vector value.

3.1.2 GPS Benchmark

The GPS radio module has significant difference from other radio devices in the Android device. It works only as receiver trying to catch and decipher the long-wave radio signals from the GPS satellites. Therefore, it have only two states – *GPS turned off* and *GPS turned on*. Android Power Profile XML file has only one scalar value for the GPs device – *GPS_ON* entry.

However, *Android Location Services* (it is a system interface to access location information on the Android devices) provides different ways to retrieve device's coordinates. There are additional *Location Providers* like cellular network information analysis and Wi-Fi network information analysis. This scenario should ensure that only GPS Location Provider is used and Wi-Fi and cellular modules are both completely off.

In addition, it is needed to keep in mind that GPS depends on the CPU. Therefore, the *GPS_ON* test scenario is actually is extension of the *CPU_AWAKE* test scenario. This test turns on the GPS hardware and subscribes to receive the location updates of the device. It is important to notice that, in general, the GPS consumption does not depend on fact was the current location calculated or the calculation was failed do due low GPS signal. After the test's time is up, GPS hardware is turn off and the previous

power state of the device (which was preserved before the test start) is restored. To get a GPS hardware consumption metric, we assume subtraction of the *CPU_AWAKE* energy from the average value during the measurement session. Therefore, the same `LoopWorker` as for *CPU_AWAKE* test scenario is used. The following pseudo code may be used to describe this scenario:

```
1. -> Disable Wi-Fi
2. -> Disable Bluetooth
3. -> Acquire Partial Wake Lock
4. -> Turn Off Screen/Wait for Screen Off Timeout
5. -> Lock Frequency at Minimum Level
6. -> Start GPS
7. -> Run LoopWorker
8. -> Stop GPS
```

3.1.3 Screen Benchmark

The screen hardware like CPU has many power states because of different brightness levels. The colors displayed while screen is on can also affect power draw on certain screen technologies. Another important thing that powered on screen prevents the system from going to the standby mode (however, it is not relevant for our methodology, as we are holding *partial wakelock* in all test scenarios for more predictable results).

It is important to keep in mind that “screen” refers not only to the graphical display itself, but also for a touchscreen and navigation buttons backlight. Despite the fact that screen may have many different power states, the reference Android Power Profile defines only two non-trivial values for the screen. First value is *SCREEN_ON* refers to the screen powered on with completely disabled backlight. Depending on the display technology and the driver implementation, it may not be possible to switch off backlight completely. If backlight switching off is not possible, the solution is to measure power consumption at the lowest possible backlight value and at the greatest possible backlight value. Then extrapolate the obtained values to get approximation of the power consumption for the screen with disabled backlight (actually, it is more important that calculated value for the minimal

brightness should be precise, as applications anyway will not be able to disable backlight on such device). The following pseudo code is used to describe this test case:

```
1. -> Disable Wi-Fi
2. -> Disable Bluetooth
3. -> Acquire Screen Wake Lock
4. -> Turn On Screen/Wake Up Device
5. -> Lock Frequency at Minimum Level
6. -> Adjust Screen Brightness to Minimum Level
7. -> Run LoopWorker
8. -> Restore Screen Brightness Settings to User Level
```

Second screen Power Profile value is *SCREEN_FULL*. This value defines *additional* power for the backlight on maximum brightness. To calculate the power for the intermediate backlight steps a fraction on this value based on the current brightness should be added to the initial *SCREEN_ON* value.

To manipulate the screen brightness the built-in Android Framework functionality may be used. However, this method have significant restriction – due to security reasons the application allowed only to change the brightness of currently shown *Activity*. However, it is not very suitable for benchmarking purposes. For benchmark application, it is better to control screen brightness directly using following commands in the shell:

```
1. # echo 100 > /sys/class/leds/lcd-backlight/brightness
2. # echo 0 > /sys/class/leds/button-backlight/brightness
```

The second command is applicable only to the devices that have buttons backlight. Providing values less than minimal possible brightness or values that are greater than maximum possible brightness has no effect. It is safe to pass zero for the minimal brightness scenario and pass big enough value for the maximum brightness scenario. On the test device, HTC Desire, the lowest effective value is 30 and the greatest effective value is 250.

Screen scenarios also depend on the *CPU_AWAKE* value as CPU is running to draw the UI on the display. As in other scenarios, the CPU frequency is kept as low as possible and no additional work on the benchmark thread is done. Therefore, *CPU_AWAKE* value subtraction is assumed for all screen scenarios. Scenarios are using the same `LoopWorker` as *CPU_AWAKE* test scenario. The following pseudo code may be used to describe the test case:

```
1. -> Disable Wi-Fi
2. -> Disable Bluetooth
3. -> Acquire Screen Wake Lock
4. -> Turn On Screen/Wake Up Device
5. -> Lock Frequency at Minimum Level
6. -> Adjust Screen Brightness to Maximum Level
7. -> Run LoopWorker
8. -> Restore Screen Brightness Settings to User Level
```

3.1.4 3G/Bluetooth/Wi-Fi Benchmarks

Measuring power consumption of the radio interfaces hardware is most challenging task and few assumptions are needed before the test is set up. All radio interfaces (Cellular, Bluetooth, Wi-Fi) may be disabled by the user completely avoiding power consumption by these hardware components. Measuring the cellular radio module power consumption is not covered in this thesis due to complexity of the setup for this test scenario (e.g. measuring *RADIO_ON* values require influence on the incoming GSM signal strength).

Therefore, the first value refers to the powered on, but not actively used component. Such values as *RADIO_ON*, *BLUETOOTH_ON*, *WIFI_ON* indicate additional power consumption of the mobile device when corresponding radio interface is turned on. However, these scenarios may have some noticeable undesirable raises of energy consumption due to external reasons. Such effects include broadcast traffic in Wi-Fi network, receiving service information in cellular network, responding to the scan requests of the other devices and so on. To avoid influence of these effects

simple filtering on the measurement time series may be applied. After raw data is collected, the average (a) and the standard deviation (s) of the series [51] are calculated. Then, all values that are not in the interval $[a - s, a + s]$ filtered out from the series. The average of the left values is assumed a Power Profile value. As usual, it is required to subtract the *CPU_AWAKE* amount of power from the raw measurements data to calculate the pure radio device consumption.

Note: the *RADIO_ON* Power Profile value is a vector value, it should contains a vector of values that are corresponds to the power consumption of the cellular radio module with different signal strength of the cellular network.

The all "ON" scenarios may be in general described using the following pseudo code:

```
1. -> Disable Other Radio Interfaces
2. -> Enable Radio Interface
3. -> Acquire Partial Wake Lock
4. -> Turn Off Screen/Wait for Screen Off Timeout
5. -> Lock CPU Frequency at Minimum Level
6. -> Run the LoopWorker
```

The next common radio interfaces Power Profile value is power consumption in scanning mode. Due to unknown reasons, Bluetooth Power Profile does not include value corresponding to the scanning process. However, in this thesis such value is referred as *BLUETOOTH_SCAN* value. The analogic value for the Wi-Fi hardware is *WIFI_SCAN* entry. For cellular radio module, the entry *RADIO_SCAN* refers to the state when this module is paging the cellular network radio tower. Wi-Fi hardware and Bluetooth hardware implement radio scanning in different ways due to differences in functionality of these wireless protocols. The Android Framework provides slightly different API for Wi-Fi networks scanning and Bluetooth neighbor devices scanning. In case of the Bluetooth it is possible to request scanning,

register *Broadcast Receiver* and receive the discovered devices information once they are discovered. The API also allows querying of the Bluetooth adapter for the current scanning state. However, the Wi-Fi Manager API in the Android Framework lacks such capabilities. The way Wi-Fi Manager API is used in the Android Framework is following: the method `startScan()` in `WifiManager` used to start/restart Wi-Fi scanning and then you should use the *Broadcast Receiver* to receive the notification that Wi-Fi scanning is done. After such notification is received, it is safe to call `getScanResults()` method to get list of the detected Wi-Fi networks. However, calling this method in general is not safe and may produce many different exceptions. Therefore, the `LoopWorker` implementation style is different for Bluetooth and Wi-Fi scanning test scenarios. The example of the `BluetoothScanningWorker` used in *BLUETOOTH_SCAN* scenario:

```
1. public void next() {
2.  counter++;
3.  if (!BluetoothManager.isDiscovering()) {
4.   BluetoothManager.startDiscovery();
5.   ThreadUtils.sleep(12);
6.  } else {
7.    ThreadUtils.sleep(0.25);
8.  }
9. }
```

The analog for the *WIFI_SCAN* is the `WiFiScanningWorker`:

```
1. public void next() {
2.  counter++;
3.  WiFiManager.startScanAccessPoints();
4.  ThreadUtils.sleep(0.1);
}
```

This set of scenarios may be in general described using the following pseudo code:

```
1. -> Disable Other Radio Interfaces
2. -> Enable Radio Interface
3. -> Acquire Partial Wake Lock
4. -> Turn Off Screen/Wait for Screen Off Timeout
```

```
5.  -> Lock CPU Frequency at Minimum Level
6.  -> Run the LoopWorker
```

The last set of wireless values set is called “active”. *RADIO_ACTIVE*, *WIFI_ACTIVE*, *BLUETOOTH_ACTIVE* Power Profile entries refer to the some usual activity done by the corresponding hardware. In case of Wi-Fi it is transmitting data over the network (downloading some file from the internet, or browsing some web pages will be adequate), in case of Bluetooth it is supposed to be transmitting the MP3 file stream to the A2DP-profile enabled wireless headset [12]. In case of the cellular radio module it is state when cellular module is transmitting/receiving (i.e. both making call and downloading files via 3G connection is suitable, Google’s documentation allows different interpretation of the scenario in this case). In this thesis, only *WIFI_ACTIVE* scenario is covered. Other “active” scenarios might be implemented using same approach. *WIFI_ACTIVE* scenario uses the following implementation of the `LoopWorker` helper:

```
1.  public void next() {
2.   counter++;
3.   InputStream urlInputStream = null;
4.   try {
5.    urlInputStream = new BufferedInputStream(url.openStream());
6.    BufferedReader reader = new BufferedReader(new
    InputStreamReader(urlInputStream));
7.    String line;
8.    int bytes = 0;
9.    while ((line = reader.readLine()) != null) {
10.    bytes += line.getBytes().length;
11.   }
12.   sLogger.info("Web document {} contains {} bytes.", url,
    bytes);
13.  } catch (UnknownHostException e) {
14.   sLogger.warn("Internet connection is not available!");
15.  } catch (IOException e) {
16.   sLogger.error("Network I/O error!", e);
17.  } finally {
18.   if (urlInputStream != null) {
19.    try {
20.     urlInputStream.close();
21.    } catch (IOException e) {
22.     // ignore
```

```
23.    }
24.   }
25.  }
26. }
```

This set of scenarios may be in general described using the following pseudo code:

```
1.  -> Disable Other Radio Interfaces
2.  -> Enable Radio Interface
3.  -> Acquire Partial Wake Lock
4.  -> Turn Off Screen/Wait for Screen Off Timeout
5.  -> Lock CPU Frequency at Minimum Level
6.  -> Run the LoopWorker
```

Android platform source code also mentions *BT_AT_COMMAND* Power Profile XML file item (in particular, constant `com.android.internal.os.PowerProfile#POWER_BLUETOOTH_AT_CMD` points to this). However, except the source code comment *"Power consumption when Bluetooth driver gets an AT command"* there is no any other traces of this entry, neither through Android developers forums, nor official Google's documentation for Android platform. Inspecting the Power Profile XML files from real devices also does not give an understanding of this entry purposes, because all devices, that we have seen contains some dummy (big enough to be sure that it is fake value) or zero mA.

3.1.5 Missed Components

There are many other components in typical Android mobile device, which are not covered by Android Power Profiles energy model. Such components may include GPU, Sound (Headphones & Speakers) and Vibration. However, usage of these components significantly affects the total power consumption of the mobile device. This may lead to significant discrepancy between estimated time and real operating time, when Android Power Profiles are used for prediction of the Battery Life time.

Therefore, using of Android Power Profiles should be avoided for prediction Battery Life time in scenarios, where these components are used

hard. Such scenarios include games, music listening, video watching, etc. In fact, almost every common usage scenario are not fully covered by the Android Power Profiles entries. The positive side of this question is that, in most cases, programmer have no influence on these factors – power consumption of the speakers and vibration cannot be decreased programmatically without affecting functionality (i.e. decreasing volume of length of vibration). Therefore, measuring power consumption of the sound and vibration are rare use-case of the usage of Android Power Profiles.

However, different power models may be used more effectively than the Android Power Profiles. Using the technique from this thesis allows to implement different scenarios according to needs of the power model. Extension of the benchmark application and the analyzing software with new scenarios for new Energy Profiles does not require significant effort. This technique also allows you to validate the hypothesis of the component usage by the application. If the offline measured total power consumption while using the application equals (with some fault of course) to the sum of the components power usage from the Energy Profile, then you have good enough model to work with this particular application. It also might be a good idea to use different power models for different types of apllications.

3.1.6 Battery Capacity Benchmarking

Let function $f = f(t)$, where f – instant value of amperage, flowing through the battery at the time moment $t \in [0; T]$. Then battery capacity C drained at the moment T may be calculated using following formula:

$$C = \int_0^T f(t)\,dt \qquad (1)$$

Note: fast discharging (discharging with high flowing current) causes the decreasing of the battery capacity for the current cycle. Therefore, it is

important to measure battery capacity discharging the battery with the common usage scenario to get the results applicable to further calculations.

Therefore, for estimating battery capacity any common usage scenario may be used (e.g. streaming video from the internet using headphones at 80% sound level would be fine). In our methodology, we use special application called Nova Battery Tester [65] [66]. It has the "Long Benchmark" tool, which will drain the battery from 98% to 1%. The application also tries to estimate the battery capacity by itself.

Because f – continuous function, the First Mean Value Theorem for Integrals [56] may be used to simplify the calculations. If F – average value of f on the interval $[0;\ T]$, then $C\ = F \times T$.

3.2 Automated Measuring Technique

Maintaining all measuring scenarios manually is time-consuming process and this process requires precise setup of the initial settings of the device. To reduce the probability of mistakes during measurement session a number of helpful tools were developed to automate the measuring process.

The general idea of the way used to derive Energy Profiles may be described using the data flow diagram (see Figure 11. Data Flow Diagram).

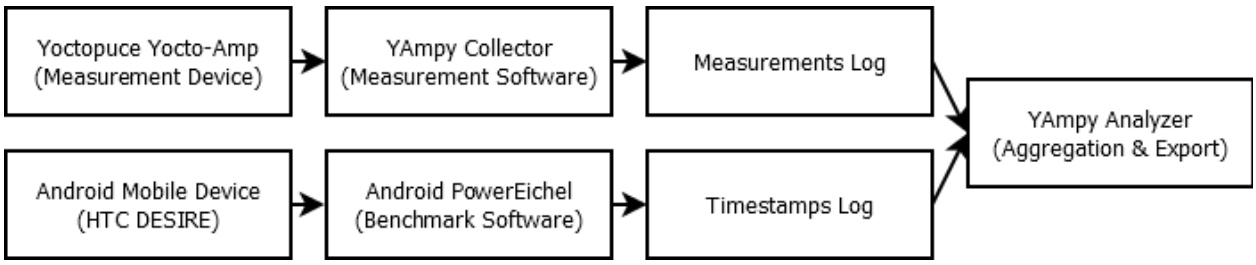


Figure 11. Data Flow Diagram

As it is seen from the diagram, the software is divided into two parts – Measurement Software for the desktop and Benchmark Software for the Android mobile device itself. Next sections give an overview of these tools.

3.2.1 Measurement Software Tool (YAmpy Application)

YAmpy is the working codename of helper measurement software. It is responsible for pulling current measurements from the Yoctopuce Yocto-Amp ammeter and keeping track of these measurements in the database.

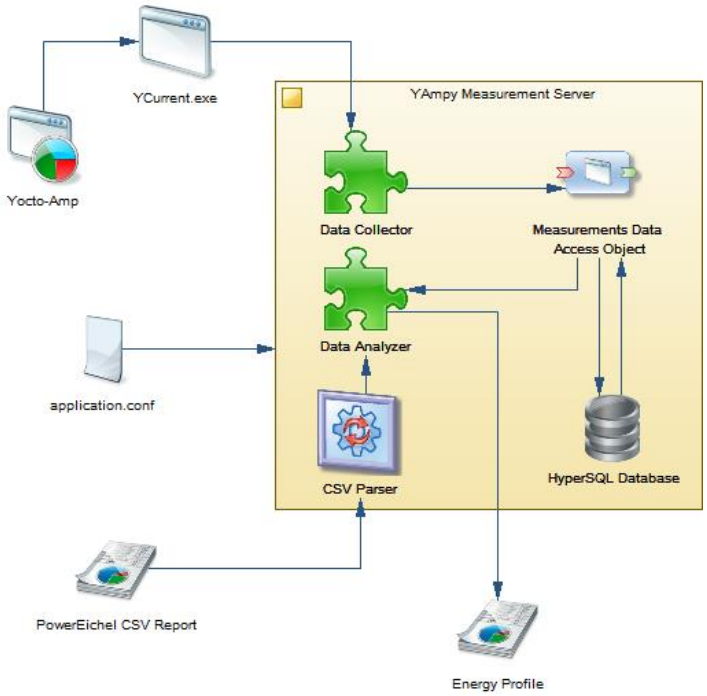


Figure 12. YAmpy Application Architecture

Also, it contains the export module called “analyzer”, which is responsible for simple data aggregation.

This application is written in Scala language and uses SBT as a build tool. The database layer is covered up by HyperSQL database. The class hierarchy diagram presented at Figure 13. YAmpy Class Hierarchy Diagram. Logically, the application architecture may be learned from the diagram presented at the Figure 12. YAmpy Application Architecture.

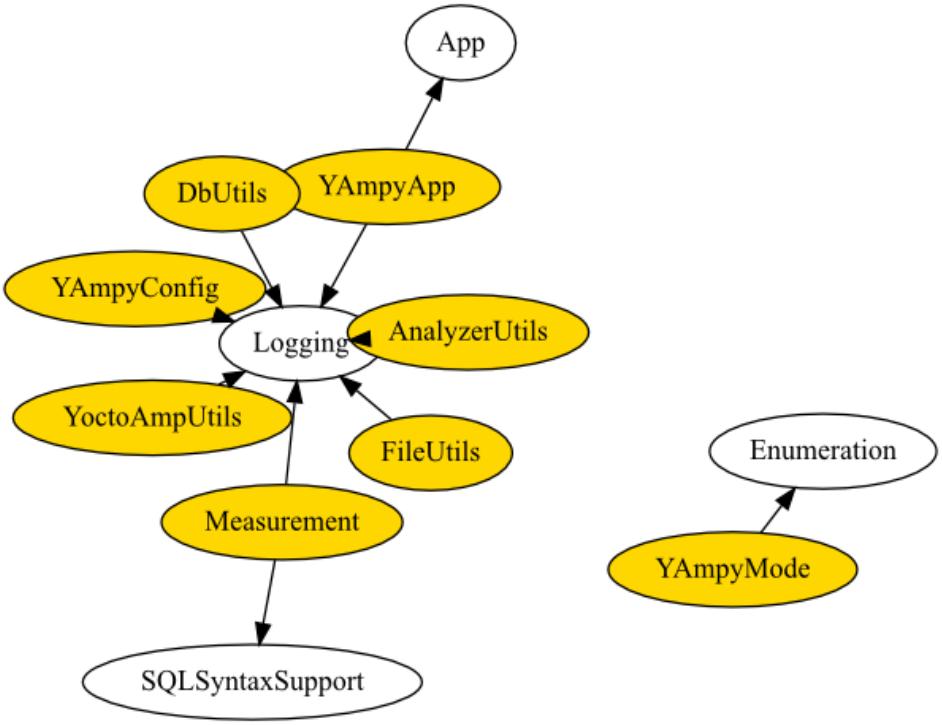


Figure 13. YAmpy Class Hierarchy Diagram

The usage of the application may be known using the build-in help command line argument: `java -jar yampy-main.jar -help`.

3.2.2 Android Energy Benchmark (PowerEichel Application)

PowerEichel is the name of Android application, which helps to setup tests for Android Power Profiles usage scenario.

PowerEichel is written in Java language with help of Android SDK and Guava library and uses Gradle build tool for assembling the package, which can be installed on the Android device.

It is guaranteed that this application will work on the HTC Desire phone (it was tested against it). However, to support other Android mobile devices, some modifications may be required. It includes the adding checks

against the device model application running on and returning correct paths to the system files application working with. Also, application may have issues with running on new versions of Android. It was tested against Android 2.3.3. However, there are some restrictions in the new versions of the Android operating systems, which will prevent the application from correct functioning (basically, it refers to the revoking of some permissions from being able to use from third-party applications).

Usage of the application is very intuitive (see screenshots at Figure 14. Android Energy Benchmark Application).

Figure 14. Android Energy Benchmark Application

After the *"Start Benchmark"* button pressed, the user interaction is not required anymore. All tests, which are built-in, will be automatically executed by the benchmark application.

To see concrete implementation of the test cases used in this work see class `org.powereichel.android.benchmark.service.EnergyBenchmarkService` in the `powereichel-gradle` project (see *Appendix A* for CD Contents).

3.3 Defining Validation Technique

This section provides the description of possible validation scenario of the derived Energy Profile value (it is applicable not only to Android Power Profile).

3.3.1 The Role of the Battery Capacity in Validation

To make a validation of the adequacy of the derived value (i.e. it represents the scenario it supposed to use in) it is need to know the total battery capacity of the Android mobile device. For brand new batteries, it may be possible to use *designed battery capacity* (information provided by the manufacturer of the battery). However, for phones that was in use it is expected to have battery degradations. In this case, it is import to measure by offline methods (i.e. with hardware) the *real battery capacity*.

3.3.2 Estimation of the Battery Life using Power Profile

Figure 15. Nova Battery Tester

First, the real battery capacity of the phone should be benchmarked. It may be done using Nova Battery Tester or any other activity that will drain the battery from 100% to 1% accompanying by the offline measurements of the amperage. In this thesis the combination of these tools are used: while using the Nova Battery Tester application for the battery draining, the measurement software (see section 3.2.1) are used for tracking the amperage of the discharge (in this case, the current flowing through the battery may be viewed as speed of the battery discharge). During the 221 minutes (3 hours and 41 minutes) 30488 values of instant amperage were registered by the measurement tool. For this usage scenario, the measurement tool does not provide a built-in analyzer for time series data. So, for the measurement session the custom “TAG” (again, see

section 3.2.1) was used for data distinction. The following SQL queries give all needed data for further calculations of the battery capacity:

```
1. select AVG(MEASUREMENT_VALUE) from MEASUREMENTS where MARKER =
   'BATTERY_CAPACITY_TEST_001'
2. select MIN(MEASUREMENT_TIMESTAMP) from MEASUREMENTS where MARKER
   = 'BATTERY_CAPACITY_TEST_001'
3. select MAX(MEASUREMENT_TIMESTAMP) from MEASUREMENTS where MARKER
   = 'BATTERY_CAPACITY_TEST_001'
```

In our case, the following input data was observed:

AVG (VALUE)	*MIN (TIMESTAMP)*	*MAX (TIMESTAMP)*
331	2014-01-30 10:30:15	2014-01-30 13:11:28

Table 2. Benchmark Data - HTC Desire [Battery Capacity]

Therefore, the battery capacity in this case:

$$331\ mA \times (221\ \div 60)\ h = 1219\ mA \times h \qquad (2)$$

After the total battery capacity is known, it is possible to start make estimations of the Battery Life in different scenarios. For example, the assumption for the video playback scenario is that the total energy consumption is *CPU_ACTIVE_N* (the power state of the CPU for each video file should be checked) and *WIFI_ACTIVE* plus *SCREEN_ON* plus relative brightness level (0..1) multiplied by *SCREEN_FULL* value. If then divide the battery capacity by the estimated power consumption the time, which phone should be able to operate in such mode should be derived. After this, it is possible to check empirically the correctness of the estimated time.

4 *Results*

This section describes the observed issues with the measurement sessions and the measurement results. Android Power Profile was derived for the test phone, HTC Desire (running Android 2.3.3). The tools described in previous section were used for deriving this Power Profile.

Before proceed with reviewing values derived using direct hardware measurement (*offline measurement*), let review the original Power Profile supplied by the vendor of our test phone (see Table 3).

NONE	CPU_IDLE	CPU_AWAKE	CPU_ACTIVE
0.0	2.8	0.0	66.6
WIFI_SCAN	WIFI_ON	WIFI_ACTIVE	GPS_ON
100.0	2.9	120.0	170.0
CPU_SPEEDS	BT_ON	BT_ACTIVE	BT_AT_COMMAND
[84.0, .., 168.4]	0.3	142.0	35690.0
SCREEN_ON	SCREEN_FULL	RADIO_ON	RADIO_SCAN
100.0	160.0	3.0	70.0
RADIO_ACTIVE	AUDIO	VIDEO	BATT_CAPACITY
300.0	88.0	88.0	1390.0

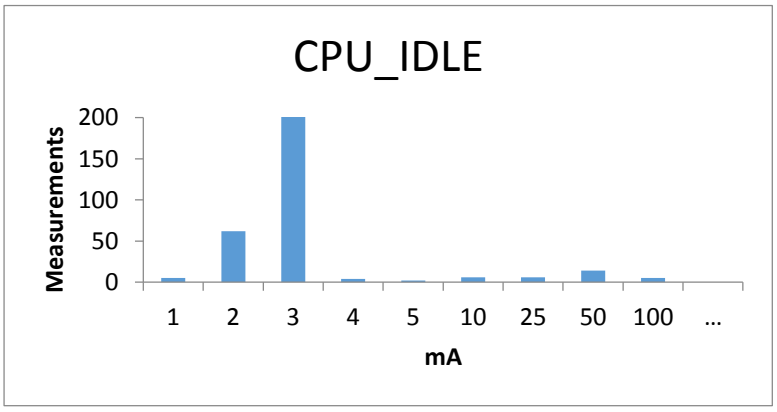
Table 3. Android Power Profile - HTC Desire

First, as we explained before, the entry *BT_AT_COMMAND* contains dummy value because it is not used in current implementation of Android Power Profiles. Second, *CPU_AWAKE* entry looks to be missed (zero mA does not qualified to look like real value). Other values doesn't look very suspicious, however, comparing to the ASUS Nexus 7 Power Profile (see Table 1. Android Power Profile - ASUS Nexus 7) the difference for Wi-Fi and Bluetooth modules looks huge. It is unlikely that there is so big difference in reality.

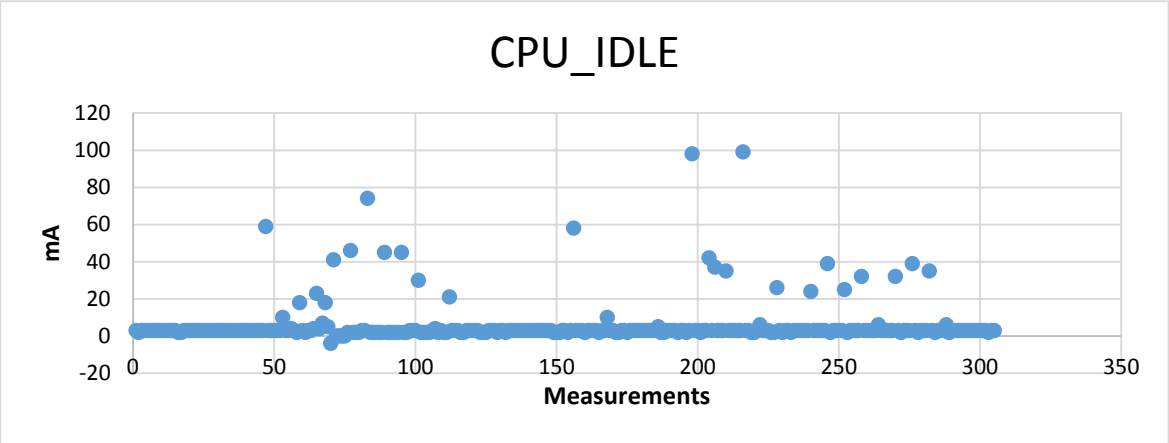
Next sections describe the method of converting raw measured values to the Energy Profile values (according to the Android Power Profile Energy Model). All descriptions are structured in a similar way to provide a good way to analyze the data. Description of each Power Profile entry includes list of the dependencies of the entry (basically, the list of values that should be subtracted from the total measured device's consumption), count of instant measured values, example of the time series values, average and standard deviation in series and a histogram with a graph to observe measured values visually.

4.1 CPU Energy Profile

The first scenario, *CPU_IDLE,* is set up manually. The test phone, HTC Desire, was rebooted and then we wait until device goes to the standby mode (screen is off because of the timeout). This scenario have not any dependencies. Because the beginning and the ending of the measurement session were set up manually, the measurement was last longer than default 1 minute tests that are done by Android application benchmark. In total 305 values were observed. The average is 6 mA and the standard deviation is 12 mA. If we will look at the histogram (Histogram 1. CPU_IDLE) and the graph (Graph 2. CPU_IDLE) we may notice that there are suspicious significant pikes.

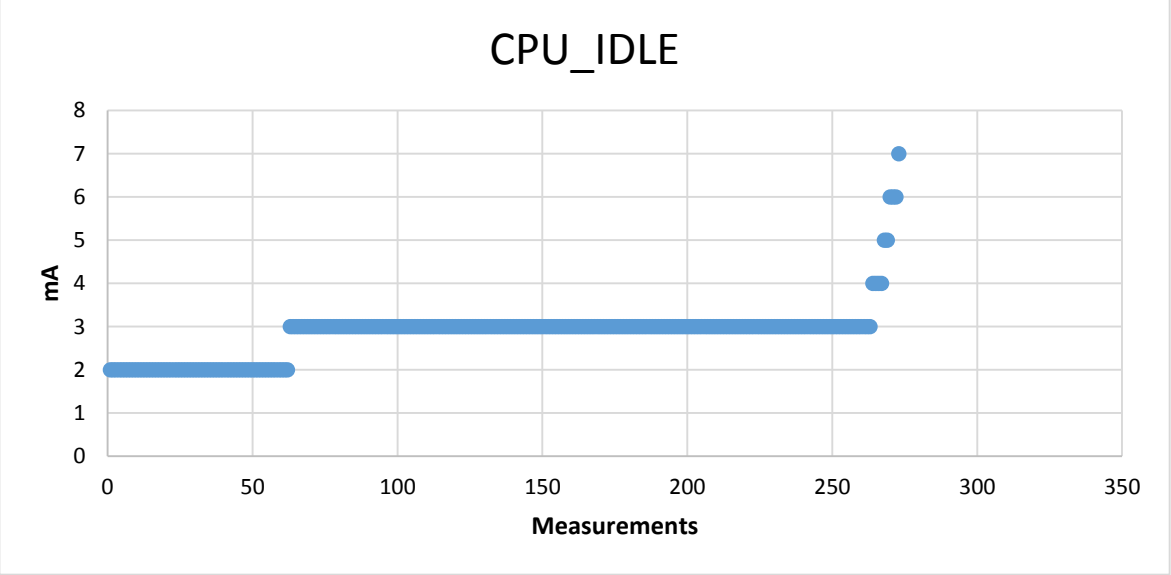


Histogram 1. CPU_IDLE



Graph 2. CPU_IDLE

This energy consumption pikes may belong to the accidental CPU wakeups because of the triggering of the scheduled Alarm Managers (this is the mechanism of the Android framework, which allows scheduling to run a program in the future [56]). Therefore, it might be good idea to filter out the outstanding values in this case. The average with filtered out values that are equals of greater than 10 mA is 2.8 mA and the standard deviation is 0.6 mA. The sorted graph (Graph 1. CPU_IDLE [Sorted] – values less than 10 mA are not included) gives the overview of the values used for calculating such average.

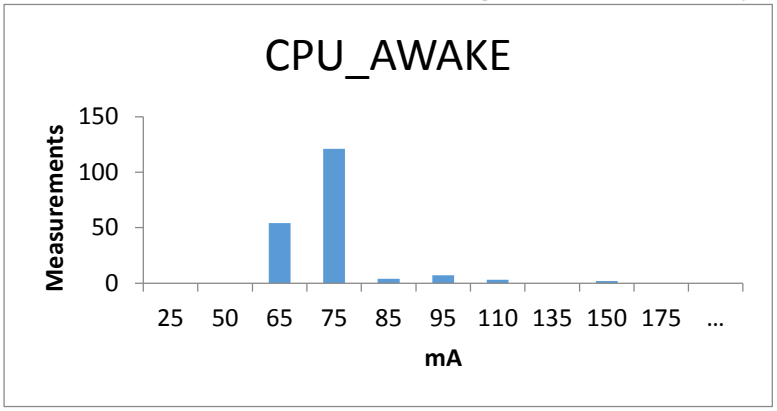


Graph 1. CPU_IDLE [Sorted]

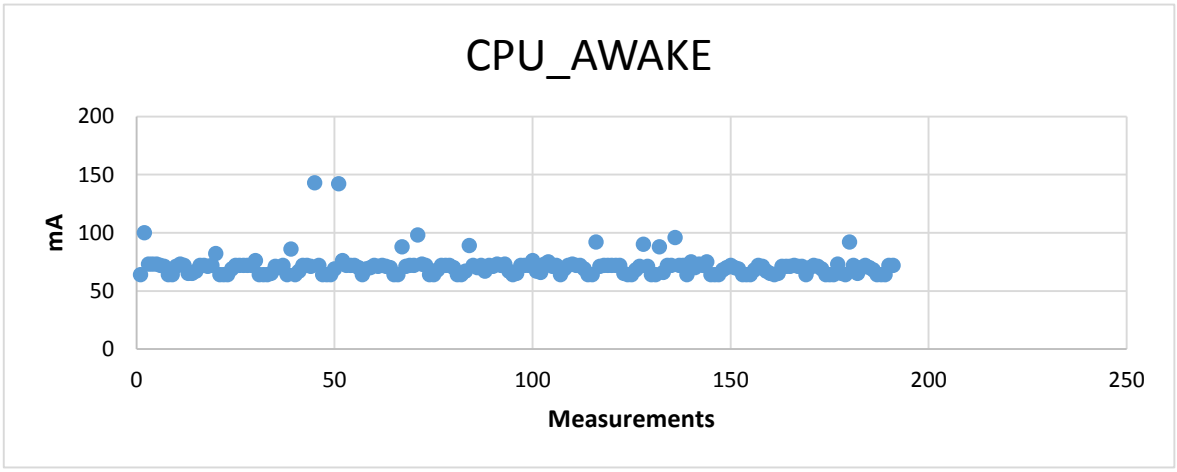
The next scenario is *CPU_AWAKE*. It is dependent from *CPU_IDLE* scenario as it is defined as *additional* amount of power,

when CPU is not in deep sleep state. This scenario is the part of automated pack of energy tests built in Android benchmark application implemented for this thesis (see section 3.2.2). In this case, measurement session report from our measurement tool (see section 3.2.1) is used. This report contains 191 measured values. The average value is 71 mA and the standard deviation is 9 mA. The histogram (Histogram 2. CPU_AWAKE) and the graph (Graph 3. CPU_AWAKE) give the overview of the character of the changes in power draining.

In this case, there are also few outstanding values that may be filtered in

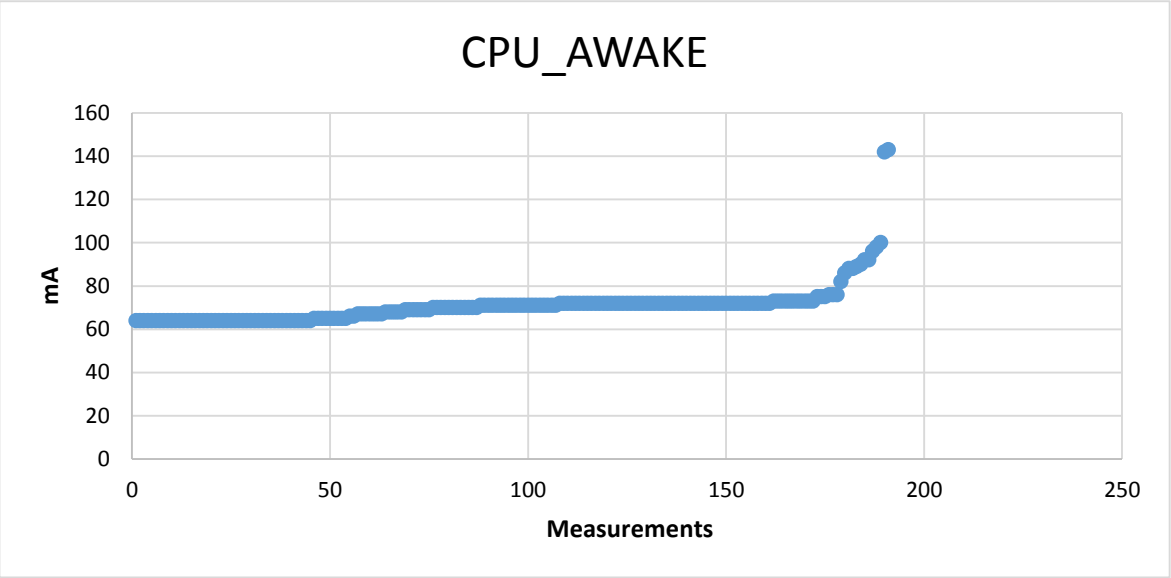


Histogram 2. CPU_AWAKE



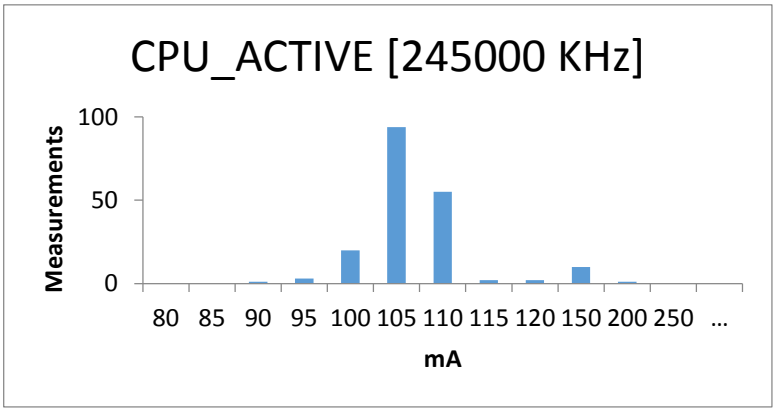
Graph 3. CPU_AWAKE

order to have more smooth results. However, if take look at sorted graph, it may be noticed that there are very few such values and they does not affect the average significantly. Sorted graph is presented on Graph 4. CPU_AWAKE [Sorted].



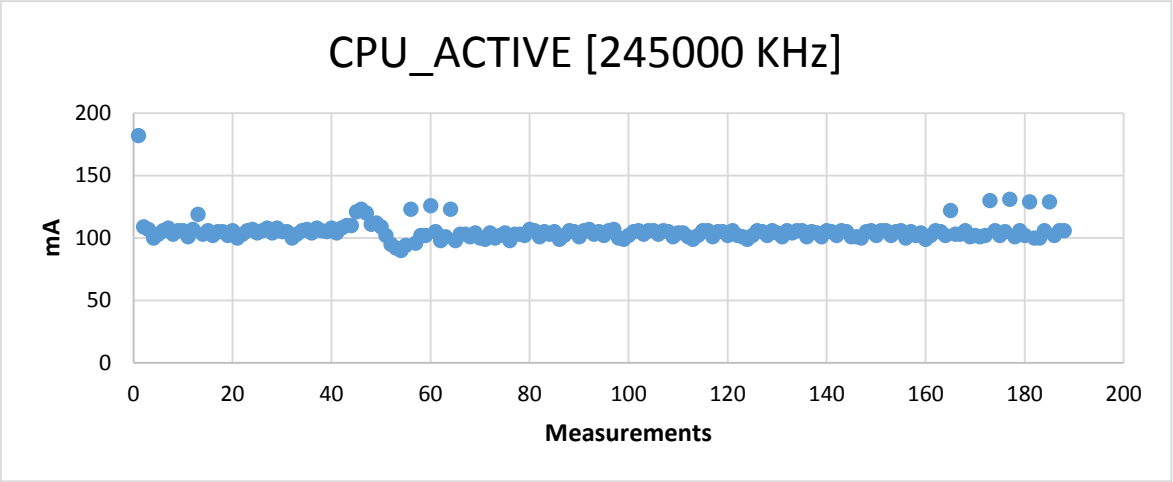
Graph 4. CPU_AWAKE [Sorted]

The *CPU_ACTIVE* entry is a vector value; therefore, it actually contains a set of values that should be obtained using the same scenario, but CPU clock is fixed at different frequencies. Our test device, HTC Desire, may operate at 12 different CPU frequencies. We will give the overview for the edge cases scenarios (with lowest and highest CPU frequency clock). These values are dependent on *CPU_IDLE* and *CPU_AWAKE* scenarios (it is good idea to subtract "raw" values measured during *CPU_AWAKE* scenario). For the *CPU_ACTIVE_0* (CPU runs at 245 MHz) scenario 188 instant amperage values were measured during 60 seconds. The average value is 105 mA and the standard deviation is 8 mA. The histogram (Histogram 3. CPU_ACTIVE



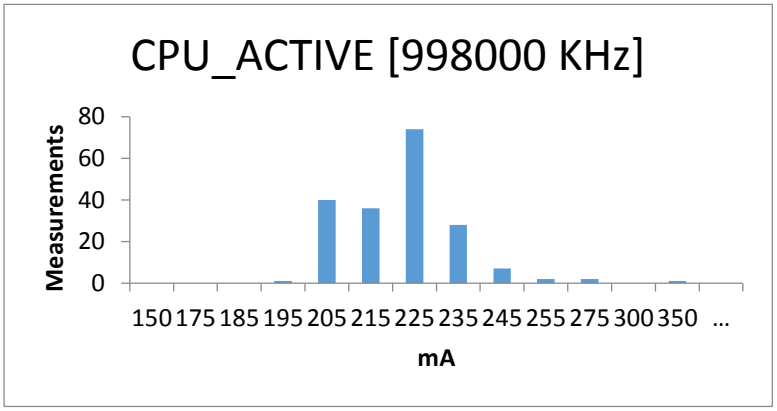
Histogram 3. CPU_ACTIVE [245 Mhz]

[245 Mhz]) and the graph (Graph 5. CPU_ACTIVE [245 MHz]) give an overview of the power consumption of the CPU under load.

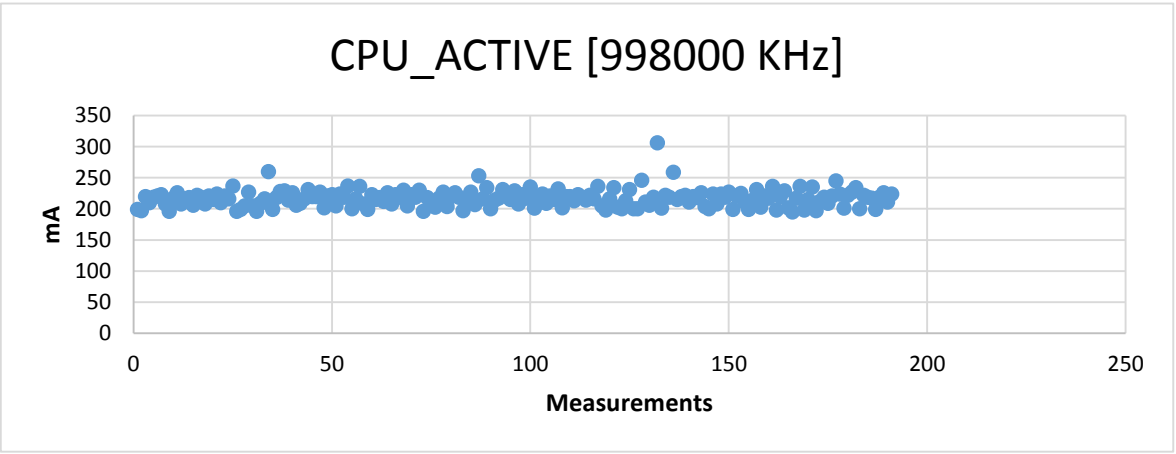


Graph 5. CPU_ACTIVE [245 MHz]

The same scenario also was run at 998 MHz (*CPU_ACTIVE_11*). In this case, 191 instant amperage values were measured. The average value is 217 mA and standard deviation is 13 mA. The histogram (Histogram 4.



Histogram 4. CPU_ACTIVE [998 MHz]



Graph 6. CPU_ACTIVE [998 MHz]

CPU_ACTIVE [998 MHz]) and the graph (Graph 6. CPU_ACTIVE [998 MHz]) give the overview of the power draining character.

To summarize the CPU measurement session results, take an overview of all “raw” measured values (see Table 4. CPU Energy Profile – HTC Desire).

IDLE	*AWAK*	*ACTIV0*	*ACTIV1*	*ACTIV2*	*ACTIV3*	*ACTIV4*	*ACTIV5*
6±12	**71±9**	**105±8**	**129±6**	**142±9**	**147±11**	**157±7**	**163±9**
		ACTIV6	*ACTIV7*	*ACTIV8*	*ACTIV9*	*ACTIV10*	*ACTIV11*
		***167*±11**	***176*±14**	***186*±9**	***191*±12**	***197*±11**	***217*±13**

Table 4. CPU Energy Profile – HTC Desire

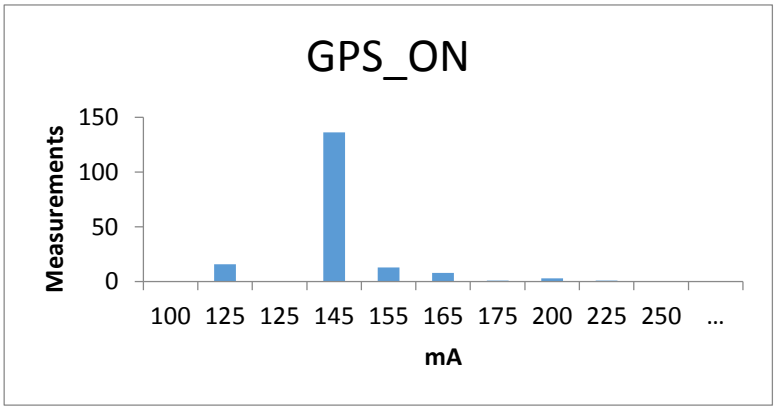
To convert this data to the actual Android Power Profile, it is needed to subtract the dependencies to get the “pure” *additional* values (see Table 5. Android Power Profile - HTC Desire [CPU]).

IDLE	6 mA
AWAKE	65 mA
ACTIVE [245 MHz]	34 mA
ACTIVE [384 MHz] (+139 MHz)	58 mA (+24 mA)
ACTIVE [460.8 MHz] (+76.8 MHz)	71 mA (+13 mA)
ACTIVE [499.2 MHz] (+38.4 MHz)	76 mA (+5 mA)
ACTIVE [576 MHz] (+76.8 MHz)	86 mA (+10 mA)
ACTIVE [614.4 MHz] (+38.4 MHz)	92 mA (+6 mA)
ACTIVE [652.8 MHz] (+38.4 MHz)	96 mA (+4mA)
ACTIVE [691.2 MHz] (+38.4 MHz)	105 mA (+9 mA)
ACTIVE [768 MHz] (+76.8 MHz)	115 mA (+10 mA)
ACTIVE [806.4 MHz] (+38.4 MHz)	120 mA (+5 mA)
ACTIVE [844.8 MHz] (+38.4 MHz)	126 mA (+6 mA)
ACTIVE [998.4 MHz] (+153.6 MHz)	146 mA (+20 mA)

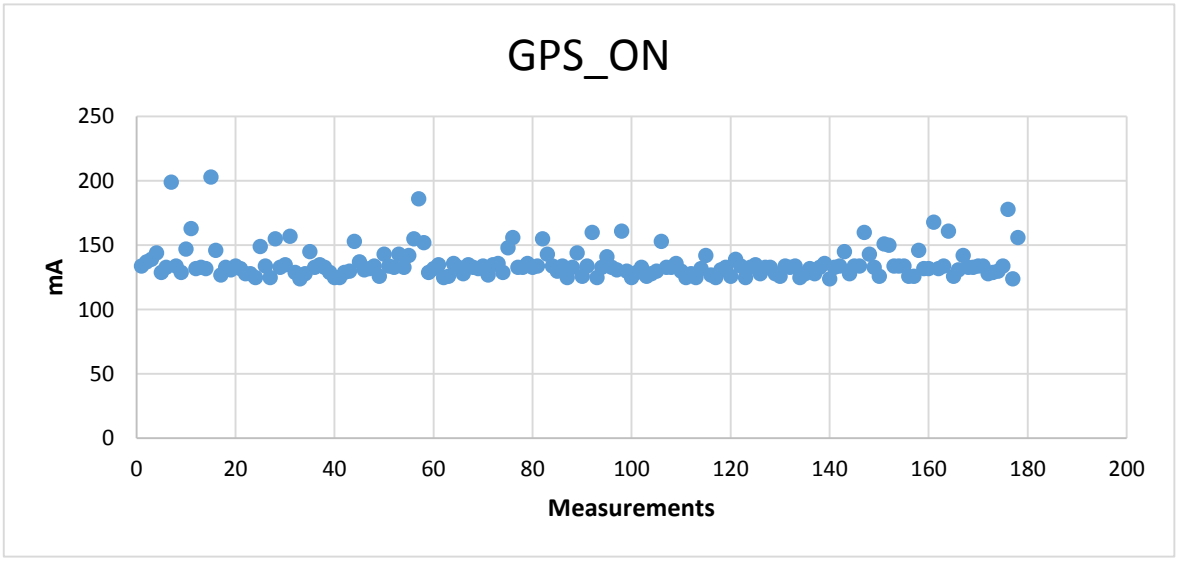
Table 5. Android Power Profile - HTC Desire [CPU]

4.2 GPS Energy Profile

The GPS hardware tests set consists only of the one test scenario – *GPS_ON*. In this, scenario the GPS Location Provider constantly updates the current location of the device. This scenario is dependent on the *CPU_AWAKE* scenario (however, it is open question – it may be better idea to use *CPU_ACTIVE_0* instead after investigating how much actually GPS Location Provider requires CPU efforts). Our test application subscribes for location updates for the 1-minute interval. In this scenario, 178 instant amperage values were registered. The average value is 136 mA and the standard deviation is 12 mA. The histogram (Histogram 5. GPS_ON) and graph (Graph 7. GPS_ON) may give a better "big picture" of character of power draining of the GPS. It is



Histogram 5. GPS_ON



Graph 7. GPS_ON

expected that radio device have more unstable energy consumption graph than CPU due to nature of how radio devices are functioning (they are adapting to the external signal strength).

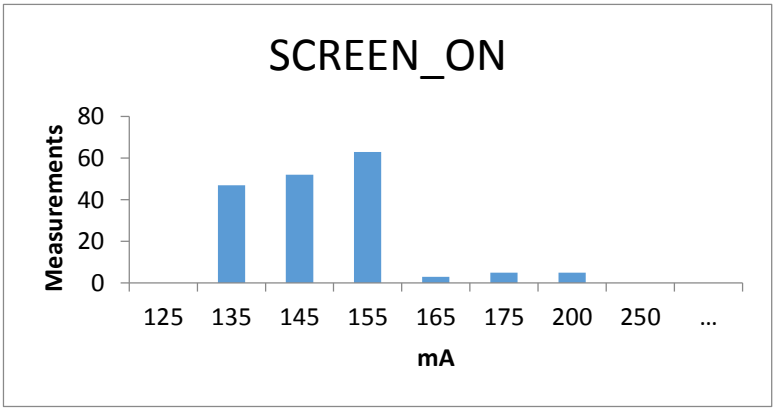
After normalizing the result (subtracting *CPU_AWAKE* value), the Power Profile entry value for the GPS_ON scenario is 65 mA.

GPS_ON	**65mA**

Table 6. Android Power Profile - HTC Desire [GPS]

4.3 Screen Energy Profile

First scenario, *SCREEN_ON* is related to the state when screen is running at minimal brightness. Actually, value defined as powered on screen *without* powered on backlight. However, it is quite hard to distinguish for certain display technologies is backlight off or just running at small brightness. We will assume that in our case, the backlight is completely off. Another assumption should be made regarding the screen color. For different screen technologies different colors on screen give different power consumption. Our test use the mostly white screen with black text on it (almost the same conditions as reading e-book). The screen scenarios depend on the *CPU_AWAKE* entry. The screen is turned on for 1 minute and during this time 175

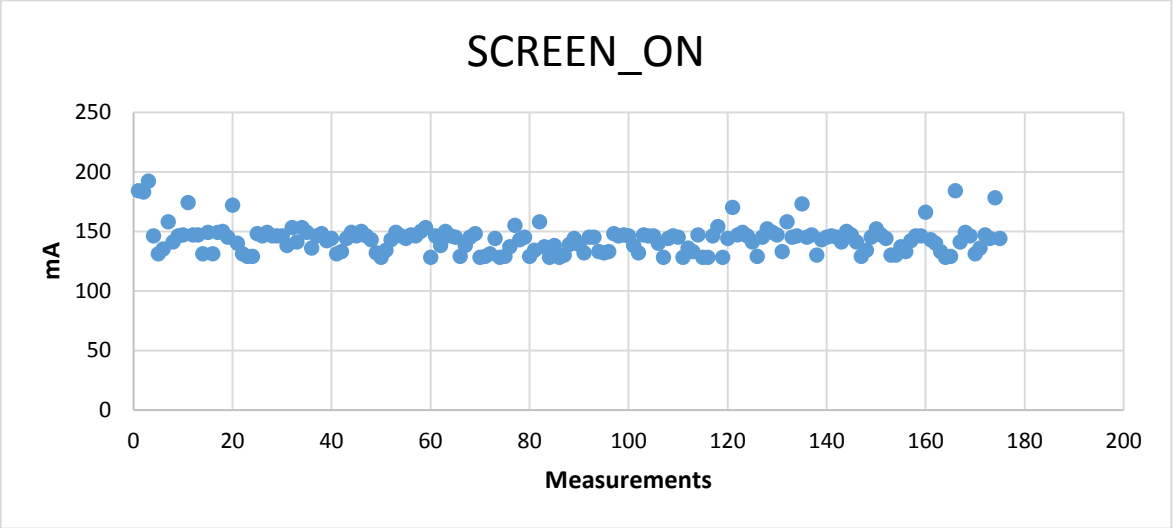


Histogram 6. SCREEN_ON

instant amperage values were registered with our software. The average value of the series is 143 mA and the standard deviation is 12 mA. The

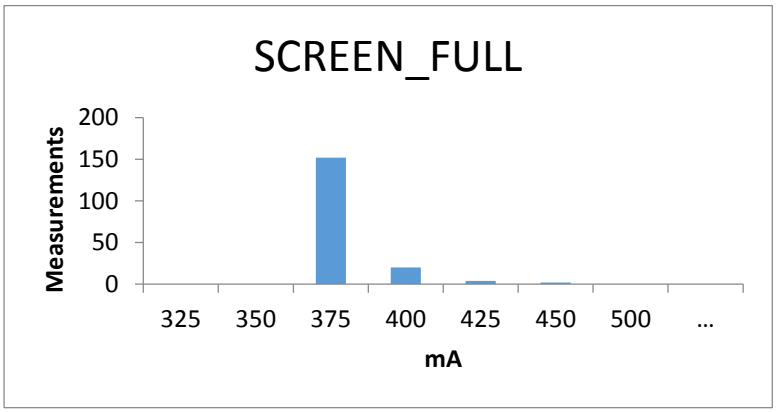
histogram (Histogram 6. SCREEN_ON) and the graph (Graph 8. SCREEN_ON) give the overview of the character of the screen power draining.

There are almost equal probabilities of consuming any value in the interval of 135 ~ 155 mA. Few outstanding values may belong to the other Android system activities (in case of powered on screen, the device operates normally and any application may react on the system-wide broadcast messages).



Graph 8. SCREEN_ON

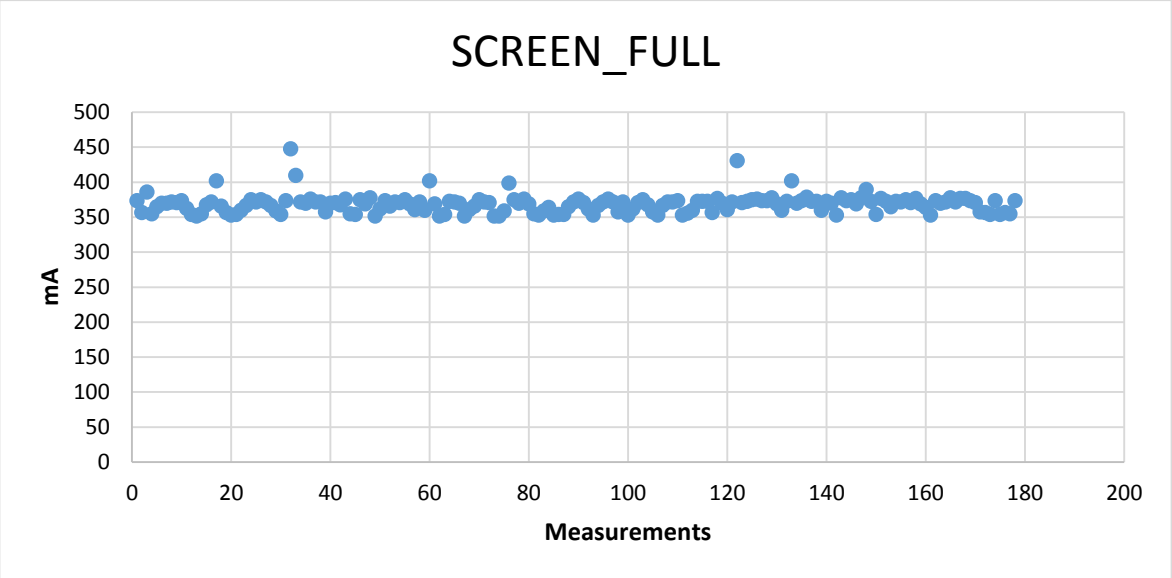
Next values, *SCREEN_FULL* refers to the full-brightness state of the screen backlight. The entry in Android Power Profile defined as *additional* power needed for the backlight. Therefore, *SCREEN_FULL*



Histogram 7. SCREEN_FULL

value depends not only on *CPU_AWAKE* entry, but also on the *SCREEN_ON* value (so, it is possible just to subtract the "raw" values of the *SCREEN_ON* test scenario). During common 1-minute test, 178 instant amperage values were measured. The average value is 368 mA and the standard deviation is 13 mA. The histogram (Histogram 7. SCREEN_FULL) and the graph (Graph 9. SCREEN_FULL) give an overview of the character of the screen power draining.

The result of the scenario is very stable, there are almost no outstanding values. This is because of the relatively high energy consumption of the



Graph 9. SCREEN_FULL

primary target hardware itself. In this case, other background activities have almost no influence on the measured values.

After calculating the "pure" values, which are show only *additional* power consumption of the appropriate components the following Android Power Profile (screen-only) takes the place (see Table 7. Android Power Profile – HTC Desire [Screen]).

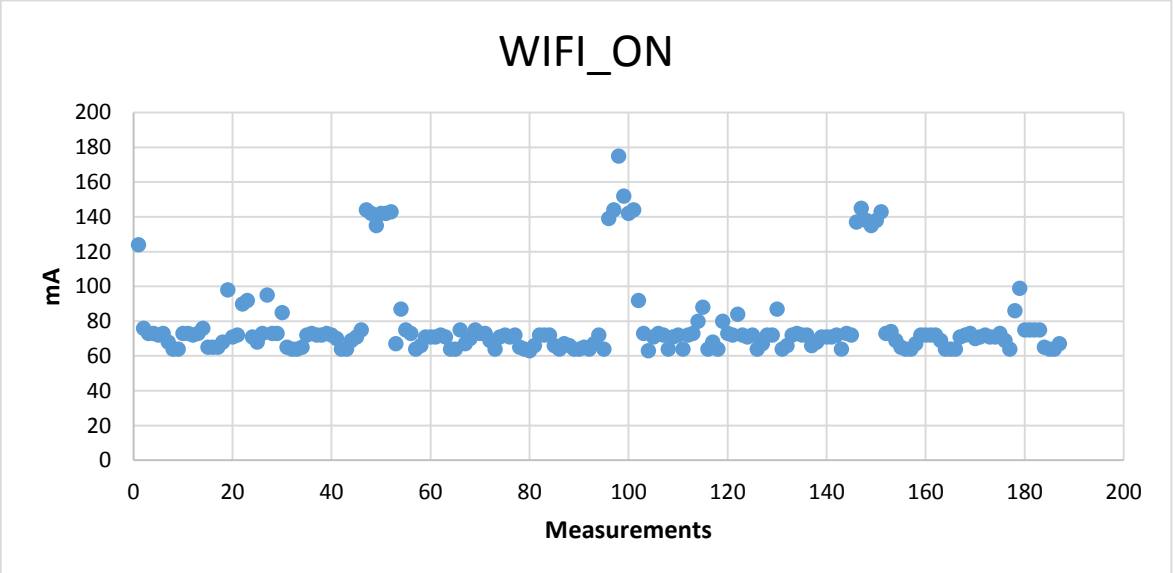
SCREEN_ON	*SCREEN_FULL*
72 mA	**225 mA**

Table 7. Android Power Profile – HTC Desire [Screen]

4.4 Radio Energy Profiles

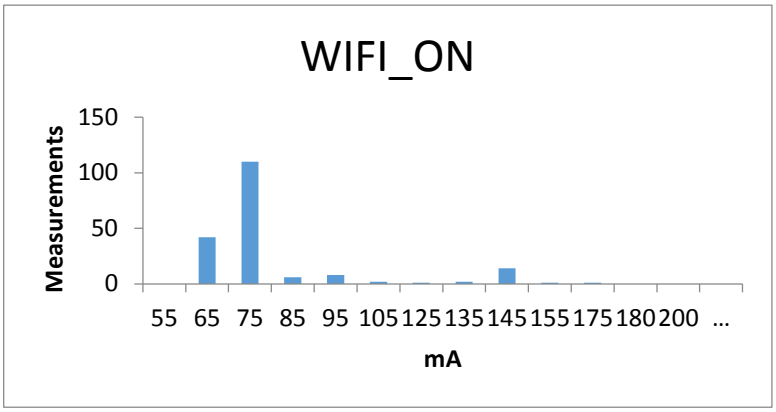
For the radio devices, we have full coverage with test scenarios only for the Wi-Fi hardware. Bluetooth Power Profile is incomplete (*BT_AT_COMMAND* is missing because it is not documented and we don't know how the test should be set up and *BT_ACTIVE* value is missing because we were unable to find suitable Bluetooth Headset with A2DP profile). The Radio Module (GSM) Power Profile is absent because all *RADIO* values are vector-based values and require many measurements under different external conditions (i.e. base tower strength signal) to derive them. However, the general idea behind the scenarios for the radio devices remains the same.

First value, *WIFI_ON*, refers to the additional power consumption when the Wi-Fi module is simply turned on, *connected to the network* and not doing any tasks. This test, however, depends on the *CPU_AWAKE* test scenario, as device is waken up, but doing nothing. During standard 1-minute test, 187 instant amperage values were measured. The average value is 78 mA and the standard deviation is 22 mA. Big standard deviation value is noticeable and it caused by some activity which constantly appearing after each 20 seconds of the test,



Graph 10. WIFI_ON

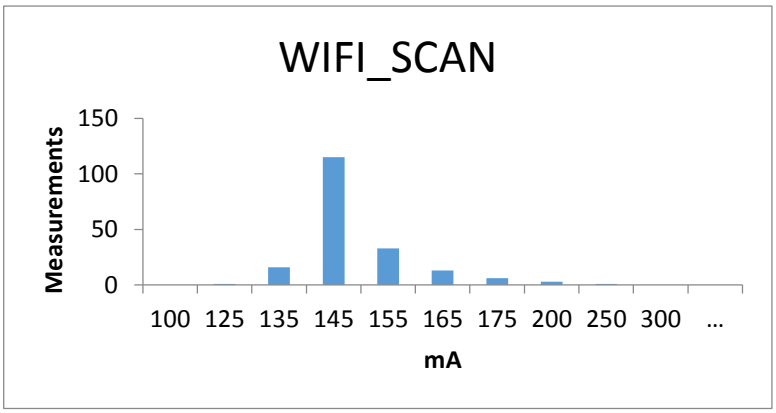
which may be observed on the histogram (see Histogram 8. WIFI_ON) and the graph (see Graph 10. WIFI_ON).



Histogram 8. WIFI_ON

This activity may be caused by some application, which is triggered when network connection is available (push updates, location updates, etc.) or by some service traffic in Wi-Fi network (e.g. responding to the DHCP and DNS requests).

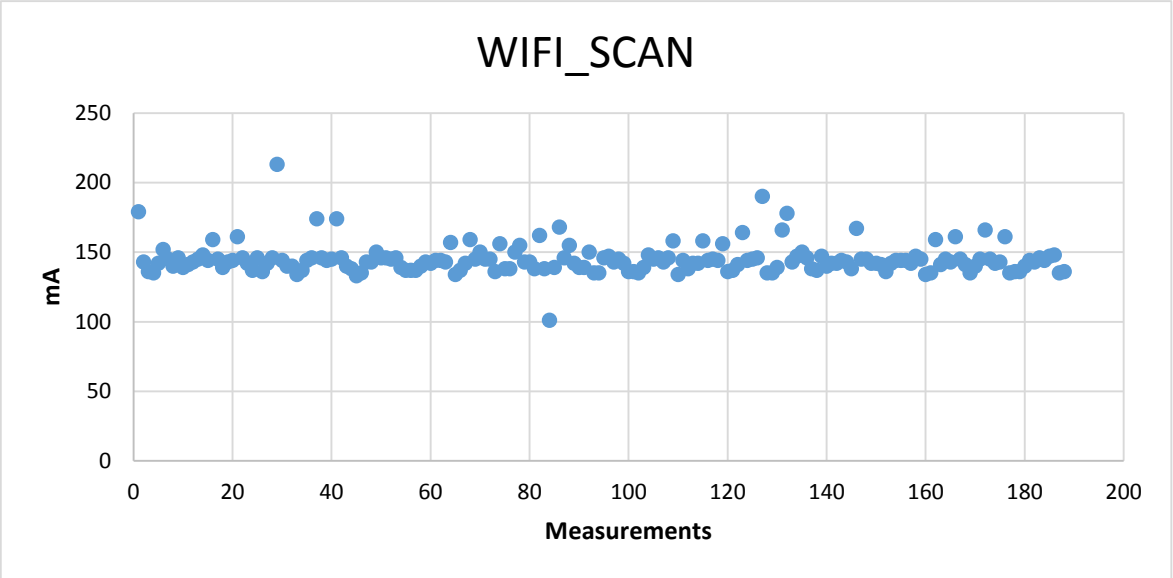
Second value, *WIFI_SCAN*, refers to the scanning for the available Wi-Fi networks. In addition to the *CPU_AWAKE*, it depends on the *WIFI_ON* entry (again, it is just needed to subtract "raw" *WIFI_ON* value from the measured one in the *WIFI_SCAN* test). During 1-minute scanning test, our measurement software registered 188 instant amperage values. The average value is 144 mA and the standard



Histogram 9. WIFI_SCAN

deviation is 10 mA. The graph (Graph 11. WIFI_SCAN) and the

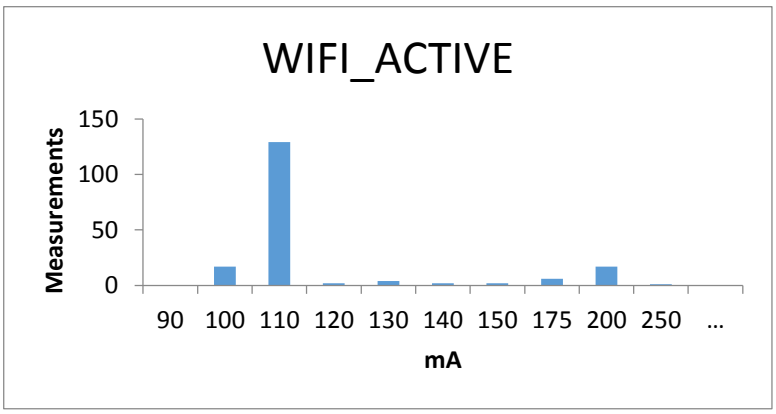
histogram (Histogram 9. WIFI_SCAN) give an overview of the power consumption character.



Graph 11. WIFI_SCAN

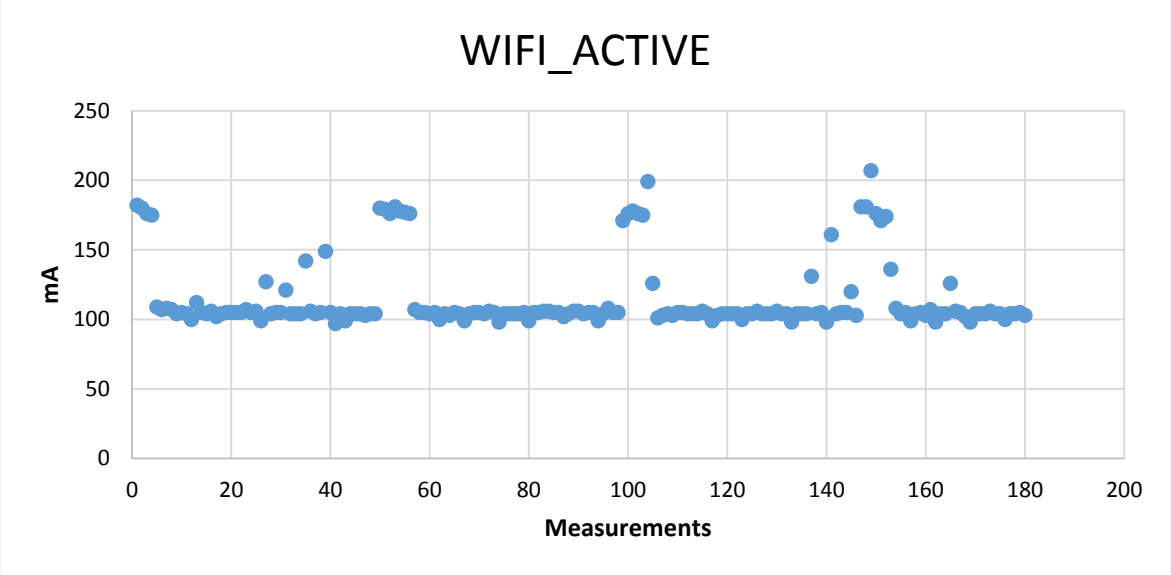
Energy consumed in the Wi-Fi scanning scenario is relatively stable. There are few outstanding values; however, they have not significant influence on the average value.

Third Wi-Fi test scenario is *WIFI_ACTIVE*. It refers to the state, in which files are constantly downloaded. Therefore, it depends on the *WIFI_ON* and *CPU_AWAKE* Power Profile entries (is acceptable to subtract "raw" *WIFI_ON* measured value). During 1-minute test scenario, 180 instant amperage values were measured. The average



Histogram 10. WIFI_ACTIVE

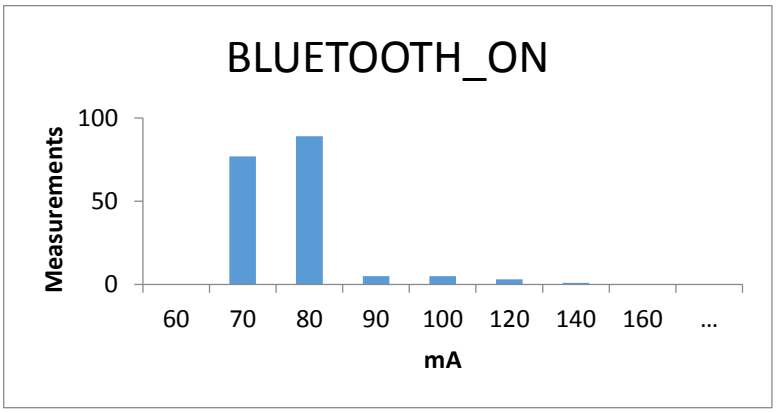
value is 115 mA and the standard deviation is 25 mA. Graph and histogram give an overview of the power consumption character.



Graph 12. WIFI_ACTIVE

Same outstanding activity, as in *WIFI_ON* test, is observed each 20 seconds during the test. However, in "pure" Power Profile *WIFI_ON* "raw" values will be subtracted, so this entry is not affected by this effect.

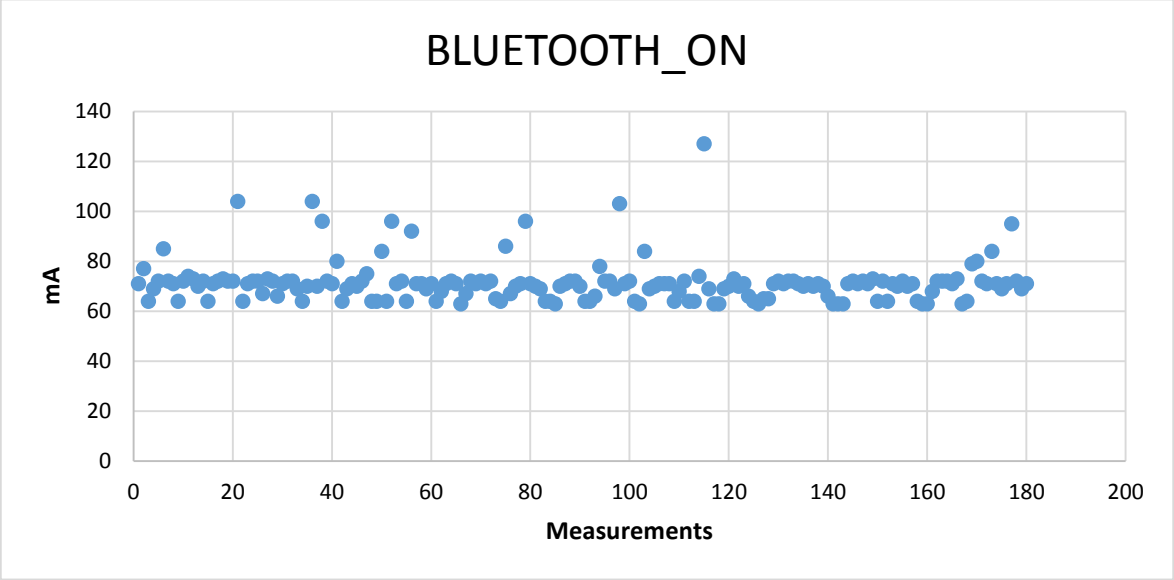
First Bluetooth test scenario, *BLUETOOTH_ON*, refers to the state, in which Bluetooth is powered on and paired with another device and no activity is done. This test depends only on the *CPU_AWAKE* entry. During standard 1-minute test, 180 instant amperage values were measured. The average value is 71.4 mA (in this case rounding contains



Histogram 11. BLUETOOTH_ON

decimal value after point) and standard deviation is 8 mA. The graph

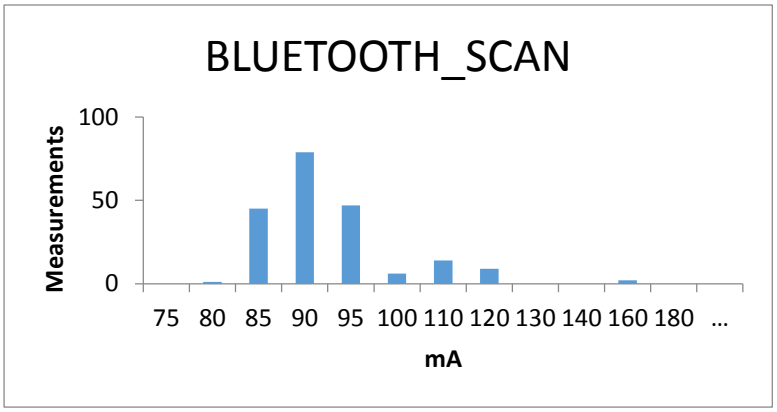
(Graph 13. BLUETOOTH_ON) and the histogram (Histogram 11. BLUETOOTH_ON) give an overview of the power consumption character.



Graph 13. BLUETOOTH_ON

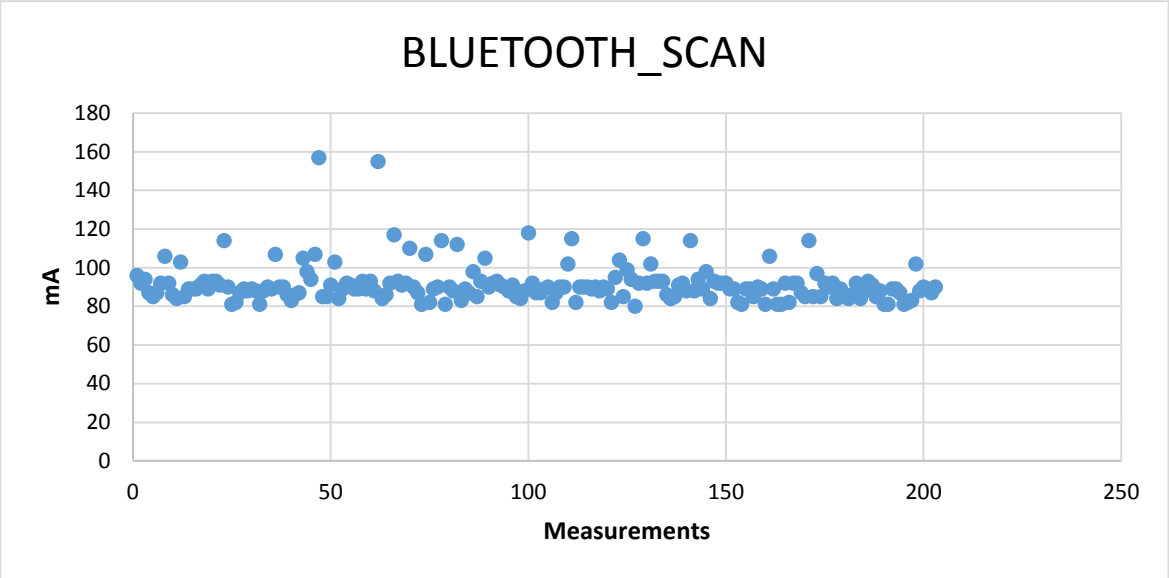
In this scenario, there is no any significant deviations in the time series values.

Second Bluetooth test case scenario, *BLUETOOTH_SCAN* is not part of the standard Android Power Profile. However, it is good idea to add such value to the Power Profiles to unify the model across entries set in Power Profile of different radio modules hardware. This value refers to the discovering of another Bluetooth devices. It depends on the *CPU_AWAKE* and *BLUETOOTH_ON* Power Profile values. During



Histogram 12. BLUETOOTH_SCAN

standard 1-minute test, 203 instant amperage values were measured. The average value is 91 mA and the standard deviation is 10 mA. The



Graph 14. BLUETOOTH_SCAN

graph (Graph 14. BLUETOOTH_SCAN) and the histogram (Histogram 12. BLUETOOTH_SCAN) give an overview of the power consumption character.

The resulting Power Profile for the radio modules hardware (after subtracting dependencies from the "raw" values) presented in the table (see Table 8. Android Power Profile – HTC Desire [Wi-Fi & Bluetooth]).

WIFI_ON	*WIFI_SCAN*	*WIFI_ACTIVE*
7 mA	**66 mA**	**37 mA**
BLUETOOTH_ON	*BLUETOOTH_SCAN*	*BLUETOOTH_ACTIVE*
0.3 mA	**20 mA**	**N/A**
RADIO_ON	*RADIO_SCAN*	*RADIO_ACTIVE*
N/A	**N/A**	**N/A**

Table 8. Android Power Profile – HTC Desire [Wi-Fi & Bluetooth]

4.5 Comparison with the Original Power Profile

To summarize the measurement session, the quick overview of the discrepancies between the original Power Profile and derived Power

Profile is given. There is table (see) that have both original Power Profile of the test phone, HTC Desire, and the Power Profile, which was updated using the values derived from the measurement session (see sections 4.1, 4.2, 0, 4.4).

	NONE	*CPU_IDLE*	*CPU_AWAKE*	*CPU_ACTIVE*
1	0	2,8	0	66,6
2	**0**	**6**	**65**	**34**
	WIFI_SCAN	*WIFI_ON*	*WIFI_ACTIVE*	*GPS_ON*
1	100	2,9	120	170
2	**66**	**7**	**37**	**65**
	BT_ON	*BT_ACTIVE*	*SCREEN_ON*	*SCREEN_FULL*
1	0,3	142	100	160
2	**0,3**	**N/A**	**72**	**225**
	RADIO_ON	*RADIO_SCAN*	*RADIO_ACTIVE*	*BATT_CAPACITY*
1	3	70	300	1390
2	**N/A**	**N/A**	**N/A**	**1220**

Table 9. Comparsion of Power Profiles - HTC Desire

The original values in regular font and measured values are marked in bold.

The significant difference in CPU Power Profile may be explained by different handling of Android Power Profiles. Because, the Android Power Profiles documentation [12] is poor quality, contains mistakes and lacks an examples the handling of Android Power Profiles may vary in very different ways. In our case, it looks like the *CPU_AWAKE* value (in interpretation of this thesis) was moved as the first value of the *CPU_ACTIVE* vector. The details on the *CPU_IDLE* value measuring and high influence of the other activities is discussed in section 4.1. The non-filtered value is taken because it mirrors better the real-world usage scenario and, therefore, is more suitable for battery life estimation (see next section).

However, the radio interfaces and screen values were improved a lot. There is only one coincidence – *BLUETOOTH_ON*, all other values are have completely different values. It might be caused by different measurement scenarios or mistakes during measurement. There is also assumption, that these values were not actually measured, but taken from the components manufacturers' specifications and refer to power consumption in completely different environments.

4.6 Energy Profiles Validation

In this section, the battery life estimation using the updated Power Profile and original Power Profile will be done. For this, two scenarios were chosen – GPS navigation application usage (Google Maps) and online video streaming application (YouTube).

The running time is considered to be calculated using following formula:

$$T_b = C \div \left(\sum_{i=1}^{n} E_i \right) \qquad (3)$$

T_b- is the time battery supposed to run (in hours), C – the battery capacity (see section 3.3.2 for details on measuring this value) and E_i is the energy consumption from the Power Profile of the corresponding component.

First scenario, using GPS navigation application, is done under following conditions: CPU may be considered being in the *CPU_ACTIVE_2* state in average. Actually, the CPU uses aggressive policy in this scenario; it runs at maximum state to process GPS response and goes to sleep as soon as possible (this behavior was observed querying `/proc/cpuinfo`, see details in section 3.1.1). Display is tuned to maximum brightness, GPS and Wi-Fi devices are powered on

and GPS is actively using. The rated battery life for vendor's Power Profile is $1220 \div (2.8 + 0 + 90 + 100 + 160 + 2.9 + 170) = 2{,}32\ h$ (the power consumption should be 525 mA in average). For the updated Power Profile it is $1220 \div (6 + 65 + 71 + 72 + 225 + 7 + 65) = 2{,}38\ h$ (the power consumption should be 511 mA in average). The average power consumption also were measured directly with Yocto-Amp (see section 2.3.2) hardware and the measured average during battery

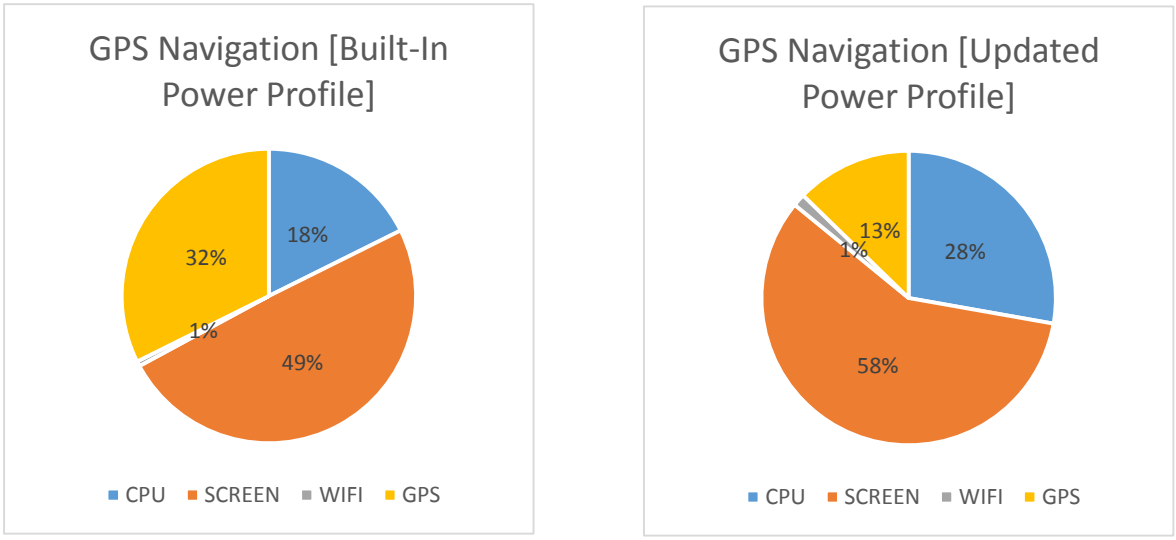


Figure 16. GPS Navigation Power Usage

discharge was 514 mA. The battery was fully discharged in 2,37 h. The fault of vendor's Power Profile is 2.2% and the fault of updated Power Profile is 0.4% (relatively to real discharge time).

Second scenario, playing the video online, involved the following components: CPU may be considered being in the *CPU_ACTIVE_5* state while decoding video, display is tuned to maximum brightness and Wi-Fi is up and downloads the video stream (therefore, it is viewed as *WIFI_ACTIVE*). The rated battery life for vendor's Power Profile is $1220 \div (2.8 + 111 + 100 + 160 + 2.9 + 120) = 2{,}46\ h$ (the power consumption should be 496 mA in average). For the updated Power Profile it is $1220 \div (6 + 65 + 92 + 72 + 225 + 7 + 37) = 2{,}42\ h$ (the power consumption should be 504 mA in average). The average power

consumption also were measured directly with Yocto-Amp (see section 2.3.2) hardware and the measured average during battery discharge was

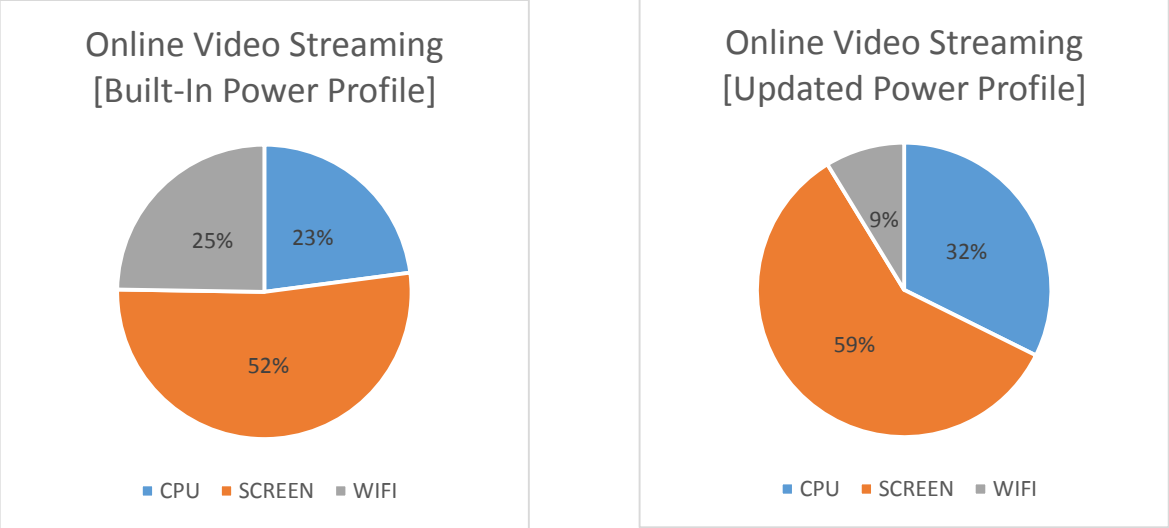


Figure 17. Online Video Streaming Power Usage

503 mA. The battery was fully discharged in 2,43 *h*. The fault of vendor's Power Profile is 1.2% and the fault of updated Power Profile is 0.4% (relatively to real discharge time).

Both Power Profiles give reasonable estimation of the battery life time. However, the updated Power Profile gives better picture of each component power usage.

4.7 HOW-TO: Derive Energy Profiles

In this section the example of the measurement application extension is given. For example, all appropriate things for measuring Energy Profile for the Vibration hardware will be added and then step-by-step the Energy Profile for Vibration hardware will be measured.

First, the new scenario should be added to Android benchmark application (see section 3.2.2). For this, the following tools are needed: Android Studio (version >=0.4.4) (see section 2.7.1) and Gradle build tool (version 1.10) (see section 2.8.3). Open the `powereichel-gradle` project in Android Studio. Create new class `VibrationManager` in `org.powereichel.android.benchmark.util` package. This class will be

responsible for switching vibration hardware states. To access any system service of Android OS the instance of `Context` class are needed. Access to the global application context implemented via additional fields on Application class (it looks like singleton, however things are more complicated under the hood – this class holds a reference to an *actual* application context, refreshing this reference when `Application` object is recreated by Android OS). Add the following line to the `PowerEichel` class:

```
1. private Vibrator mVibrator;
```

Then initialize this variable in `onCreate()` method:

```
1. mVibrator = (Vibrator)
   getSystemService(Context.VIBRATOR_SERVICE);
```

Do not forget to add getter to provide access to this service from other application classes:

```
1. public static Vibrator getVibrator() {
2.   return sApplication.mVibrator;
3. }
```

Go back to the `VibrationManager` class:

```
1. public class VibrationManager {
2.
3.   private static final String TAG = "VibrationManager";
4.   private static final Logger sLogger =
   LoggerFactory.getLogger(TAG);
5.
6.   private static volatile boolean isVibrating = false;
7.   private static final Object monitor = new Object();
8.
9.   public static void startVibration() {
10.    Vibrator vibrator = PowerEichel.getVibrator();
11.    long[] pattern = {0, 1000};
12.    synchronized (monitor) {
13.      isVibrating = true;
14.      vibrator.vibrate(pattern, 0);
15.    }
16.  }
17.
18.  public static void stopVibration() {
```

```
19.     Vibrator vibrator = PowerEichel.getVibrator();
20.     synchronized (monitor) {
21.       isVibrating = false;
22.       vibrator.cancel();
23.     }
24.   }
25.
26.   public static boolean isVibrating() {
27.     return isVibrating;
28.   }
29. }
```

The only thing left is the implementation of the scenario itself. For this, open the class `EnergyBenchmarkService` in the `org.powereichel.android.benchmark.service` package. Create method `runVibrationActiveBenchmark()`:

```
1.  private void runVibrationActiveBenchmark(String
    benchmarkGUID) {
2.   EnergyBenchmark benchmark = new CPUAwakeEnergyBenchmark();
3.   List<EnergyBenchmarkTimestamp> timestamps = new
    ArrayList<>();
4.   // === Setup Benchmark ===
5.   waitForScreenOff();
6.   disableWiFi();
7.   disableBluetooth();
8.   // === Start Benchmark ===
9.   String benchmarkName = "VIBRATION_ACTIVE";
10.  VibrationManager.startVibration();
11.  CPUManager.lockFrequencyAt(CPUMonitor.getMinimumFreq());
12.  doWork(benchmark, benchmarkGUID, benchmarkName,
    timestamps);
13.
14.  // === Finish Benchmark ===
15.  VibrationManager.stopVibration();
16.  processBenchmarkResults(timestamps);
17. }
```

Add the call of this method to the `run()` method of closure `Runnable` inside the `onStartCommand()` method. You are done with new scenario implementation.

To measure the energy consumption, connect the Yocto-Amp ammeter to the PC that running YAmpy measurement tool (see section 3.2.1) and start the benchmark (you need an SD card to be available on

the phone to get the CSV report). After benchmark is finished, copy the CSV report from the device to the computer, which runs the YAmpy tool and use `analyze` command to export the measured data in XLSX (Microsoft Excel) format.

4.8 Generalization

The approach used in the thesis, has some limitations and works only with described assumptions. This section gives an overview of how to apply this approach to another devices and platforms.

The hardware part of this work includes taking a look at alternatives. However, the device was chosen mostly because of its availability and low price. If you will consider choosing another device, keep in mind the easy-to-use API and compatibility. It also will be good, if the measurement device will have higher refresh rate because modern phones tries to go to lower power state as soon as possible and sometimes the refresh rate of Yocto-Amp (see section 2.3.2) might be not sufficient. It also need to be considered that many modern phones have non-removable batteries and it is not possible to connect ammeter to them without cracking the case of the phone.

From the software prospective, it need to keep in mind that delivered Android app (see section 3.2.2) was tested using only one single phone running specific HTC's build of Android 2.3.3. The application full of workarounds for Android API limitations and therefore, the code might need to be ported to other phones. The CPU tests were implemented in assumption that we have only one CPU core, which is not true for modern phones. It is easy to overcome this limitation, but it requires additional testing. In addition, the paths to the Linux kernel system files in application are specific for this particular model of the phone and it is needed to implement dynamical choosing

of the correct path depending on the phone model to adapt the app to the other phones.

From the methodology prospective, the approach may be applied to other platforms like iOS or Windows Phone. However, before applying it, the API of those systems should be carefully revised and test scenarios (see examples in pseudo-code in subsections of section 3.1) need to be implemented regarding native system API. In addition, it need to be checked if there are way on those platforms to count the component usage time. Other platforms than Android are known to lack the multi-tasking implementation, due to this limitation the approach should be revised. For example, it might be possible for applications on these platforms to have some library built-in to keep tracking the components usage time. In this case, at least the access to the state of components (the ability to query system is certain component in particular state at least) is required.

4.9 Direction of the Future Research

This thesis is mostly focused on the technical side of measuring Energy Profiles for Android platform. However, there are many related questions, which need to be solved before the Energy Profiles become a usable technique.

First, as it was seen sometimes during the measuring scenario some deviations occurs. The assumption, that some other applications are triggered by alarm managers or broadcast receivers. It might be good idea to implement filtering of measured values. However, it is need to be determined which activity is belongs to the test scenario and which period of time was affected by external influence. For this, the precise system of component usage, which is able to exactly determine which components were active in certain moment of time. If such information

will be available, it will increase the precision of the measured values a lot.

Second, the main goal of the Energy Profiling process is to determine energy inefficient applications. For this, it is needed to map instant energy of consumption of each component to the application that use the component in particular time moment. It is even more challenging task than in previous paragraph, because it is needed to know not only which components were active, but also which applications triggered the component activation respectively. This information will allow calculating of how much each application consumed the power and detect battery drains efficiently.

Third, the Android Power Profile model lacks many components. For accurate calculation, it is important to keep tracking of as many components as device may have. For this the custom Android power model should be developed, which will include all components and probably more convenient combination of components usage. For example, it is hard to measure separately how much energy spent by CPU and DSP co-processor while decoding the video. However, it is more convenient to know how the energy consumption increases in total while decoding particular format by particular implementation (i.e. library). This requires classifying the common application's work patterns (video decoding, audio decoding, etc.) and providing and well-documented scenario (probably, also the tool) to measure energy consumption for this scenario of that particular implementation. Another example of the Android Power Profiles inadequacy is that they do not provide a way of providing hardware-specific information. For example, for displays the energy consumption also depends from the colors on the screen, not only the screen brightness. However, the Android Power Profiles have no colors section for display states. Other

components may have states that are not covered by Android Power Profiles. All this limitations significantly affects the energy consumption estimations.

5 *Conclusion*

In this thesis, it was shown how to calculate Energy Profiles for Android Platform using the reference Android power model – Android Power Profiles. The extendable software tools were developed in order to provide an example of general methodology of Energy Profiles calculation. The approach was evaluated using the HTC Desire test phone and updated Power Profiles for this Android mobile device was delivered. In addition, it was shown that the updated Power Profiles give more precise results for estimation of the mobile device battery life time.

6 *References*

[1] Intel Developer Zone, "Wakelocks: Detect No-Sleep Issues in Android* Applications," [Online]. Available: http://software.intel.com/en-us/articles/wakelocks-detect-no-sleep-issues-in-android-applications. [Accessed 23 December 2013].

[2] Google Inc., "Android, the world's most popular mobile platform," [Online]. Available: http://developer.android.com/about/index.html. [Accessed 10 November 2013].

[3] Android Developers, "Best Practices for Performance - Optimizing Battery Life," [Online]. Available: http://developer.android.com/training/monitoring-device-state/index.html. [Accessed 17 November 2013].

[4] C. Wilke, "JouleUnit - A generic framework for profiling ICT applications," [Online]. Available: https://code.google.com/p/jouleunit/. [Accessed 10 November 2013].

[5] M. Gottschalk, Energy Refactorings, Oldenburg: University of Oldenburg, 2013.

[6] Android Developers, "BatteryManager API (Javadoc)," 2013. [Online]. Available: http://developer.android.com/reference/android/os/BatteryManager.html. [Accessed 10 November 2013].

[7] P. Mochel, "The sysfs Filesystem," 2005. [Online]. Available: https://www.kernel.org/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf. [Accessed 10 November 2013].

[8] T. H. Bui, "Android* Power Measurement Techniques," Intel Corporation, 6 January 2012. [Online]. Available: http://software.intel.com/en-us/articles/android-power-measurement-techniques. [Accessed 10 November 2013].

[9] CurrentWidget Source Code, "com.manor.currentwidget.library.CurrentReaderFactory," [Online]. Available: https://code.google.com/p/currentwidget/source/browse/trunk/CurrentWidgetLibrary/src/com/manor/currentwidget/library/CurrentReaderFactory.java. [Accessed 15 November 2013].

[10] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1 ed., Addison-Wesley Professional, 1994, p. 416.

[11] Android Source Code (4.3_r2.1), "android.os.BatteryStats," [Online]. Available: http://grepcode.com/file/repository.grepcode.com/java/ext/com.google.android/android/4.3_r2.1/android/os/BatteryStats.java. [Accessed 15 November 2013].

[12] Android Developers, "Power Profiles for Android," 2013. [Online]. Available: https://source.android.com/devices/tech/power.html. [Accessed 10 November 2013].

[13] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba., "ADVANCED CONFIGURATION AND POWER INTERFACE

SPECIFICATION," 6 December 2013. [Online]. Available: http://www.acpi.info/spec50.htm. [Accessed 15 November 2013].

[14] T. Maturo, "Short ARM Holdings: Sensitivity Analysis On Market Share," 16 May 2013. [Online]. Available: http://www.nasdaq.com/article/short-arm-holdings-sensitivity-analysis-on-market-share-cm246545. [Accessed 25 November 2013].

[15] Sushu Zhang (Intel), "Intel® Power Monitoring Tool for Android* Devices – A power and performance related data profiling tool for Android Software Developers," 2 October 2012. [Online]. Available: http://software.intel.com/en-us/articles/intel-power-monitoring-tool-for-android-devices-a-power-and-performance-related-data. [Accessed 19 January 2014].

[16] ExtremeTech, "Medfield, two years in: What killed Intel's mobile phone ambitions?," 29 November 2013. [Online]. Available: http://www.extremetech.com/extreme/171682-medfield-two-years-in-what-killed-intels-mobile-phone-ambitions. [Accessed 19 January 2014].

[17] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, Refactoring: Improving the Design of Existing Code, 1 ed., Addison-Wesley Professional, 1999, p. 464.

[18] Qualcomm Developer Network, "Trepn Profiler," [Online]. Available: https://developer.qualcomm.com/mobile-development/performance-tools/trepn-profiler. [Accessed 9 December 2013].

[19] Android Developers, "Best Practices for Background Jobs," [Online]. Available: http://developer.android.com/training/run-background-service/create-service.html. [Accessed 17 November 2013].

[20] Google Play Store, "PowerTutor," [Online]. Available: https://play.google.com/store/apps/details?id=edu.umich.PowerTutor&hl=ru. [Accessed 28 January 2014].

[21] PowerTutor.org, "A Power Monitor for Android-Based Mobile Platforms," [Online]. Available: http://ziyang.eecs.umich.edu/projects/powertutor/. [Accessed 28 January 2014].

[22] GItHub, "PowerTutor Project," [Online]. Available: https://github.com/msg555/PowerTutor. [Accessed 28 January 2014].

[23] Little Eye Labs, "Performance Analysis and Monitoring Tools for Android Developers," [Online]. Available: http://www.littleeye.co/. [Accessed 29 January 2014].

[24] Little Eye Labs, "Understanding, debugging and fixing power bugs," [Online]. Available: http://www.littleeye.co/blog/2013/03/24/understanding-debugging-and-fixing-power-bugs/index.html. [Accessed 29 January 2014].

[25] Little Eye Labs, "How Little Eye Measures Power Consumption," [Online]. Available: http://www.littleeye.co/blog/2013/07/30/how-little-eye-measures-power-consumption/. [Accessed 30 January 2014].

[26] Little Eye Labs, "Little Eye Labs is joining Facebook!," [Online]. Available: http://www.littleeye.co/transition.php.html. [Accessed 29 January 2014].

[27] Meetup - The San Francisco Android User Group, "Learn about power consumption and battery life on Android devices," [Online]. Available: http://www.meetup.com/sfandroid/events/12803170/?action=detail&eventId=12803170. [Accessed 31 January 2014].

[28] YouTube, "Learn about energy consumption and battery life on Android devices," [Online]. Available: http://www.youtube.com/watch?v=TwVoxaO0f1A. [Accessed 31 January 2014].

[29] SlideShare by LinkedIn, "Mobile Energy Consumption," [Online]. Available: http://www.slideshare.net/marakana/learn-about-energy-consumption-and-battery-life-on-android-devices?from_search=4. [Accessed 31 January 2014].

[30] Responsible Energy Corporation, "Glossary of Battery Terms," 2013. [Online]. Available: http://www.greenbatteries.com/batteryterms.html. [Accessed 24 November 2013].

[31] SANYO Component Europe GmbH, "Capacity," 2013. [Online]. Available: http://www.eneloop.info/home/technology/capacity.html. [Accessed 2014 11 2013].

[32] Apple Inc., "Lithium-Ion batteries," [Online]. Available: http://www.apple.com/batteries/. [Accessed 25 November 2013].

[33] N. Riedel, Electric Circuits, 9 ed., Prentice Hall, 2010, p. 816.

[34] Yoctopuce, "Yocto-Amp," [Online]. Available: http://www.yoctopuce.com/EN/products/usb-electrical-sensors/yocto-amp. [Accessed 24 November 2013].

[35] Yoctopuce, "About Us," [Online]. Available: http://www.yoctopuce.com/EN/aboutus.php. [Accessed 24 November 2013].

[36] Yoctopuce, "VirtualHub," [Online]. Available: http://www.yoctopuce.com/EN/virtualhub.php. [Accessed 24 November 2013].

[37] Yoctopuce, "Libraries," [Online]. Available: http://www.yoctopuce.com/EN/libraries.php. [Accessed 24 November 2013].

[38] Yoctopuce, *Yocto-Amp, User's Guide,* p. 177.

[39] MakerBot, "MakerBot Replicator 2 Desktop 3D Printer," [Online]. Available: http://store.makerbot.com/replicator2. [Accessed 9 February 2014].

[40] Monsoon Solution Inc., "Power Monitor," [Online]. Available: http://www.msoon.com/LabEquipment/PowerMonitor/. [Accessed 24 November 2013].

[41] Monsoon Solutions Inc., "PowerTool Software," [Online]. Available: http://msoon.github.io/powermonitor/. [Accessed 24 November 2013].

[42] Gitorious (Replicant), "Samsung P3100 Power Profile XML," [Online]. Available: https://gitorious.org/replicant/device_samsung_p3100/source/0334722f0c1f854c68bee6b98913257411527b9b:common-overlay/frameworks/base/core/res/res/xml/power_profile.xml. [Accessed 20 January 2014].

[43] Replicant, "Replicant Wiki," [Online]. Available: http://redmine.replicant.us/projects/replicant/wiki#Introduction. [Accessed 20 January 2014].

[44] Oracle, "Java SE HotSpot at a Glance," [Online]. Available: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html. [Accessed 10 December 2013].

[45] The Android Source Code, "Dalvik Executable Format," [Online]. Available: http://source.android.com/devices/tech/dalvik/dex-format.html. [Accessed 10 December 2013].

[46] C. Mcmanis, "The basics of Java class loaders," 1 October 1996. [Online]. Available: http://www.javaworld.com/article/2077260/learn-java/the-basics-of-java-class-loaders.html. [Accessed 24 December 2013].

[47] Sun Microsystems, Inc., "Java Language Specification - Constant Expression," [Online]. Available:

http://docs.oracle.com/javase/specs/jls/se5.0/html/expressions.html#15.28. [Accessed 24 December 2013].

[48] Oracle Technology Network, "Trail: The Reflection API," [Online]. Available: http://docs.oracle.com/javase/tutorial/reflect/. [Accessed 25 December 2013].

[49] Open Handset Aliance, "FAQ," [Online]. Available: http://www.openhandsetalliance.com/oha_faq.html. [Accessed 10 December 2013].

[50] Android Developers, "Tools Help," [Online]. Available: http://developer.android.com/tools/help/index.html. [Accessed 25 December 2013].

[51] Oracle, "The try-with-resources Statement," [Online]. Available: http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html. [Accessed 26 January 2014].

[52] Android Developers, "Android Debug Bridge," [Online]. Available: http://developer.android.com/tools/help/adb.html. [Accessed 15 January 2014].

[53] CyanogenMod Inc., "All About Recovery Images," [Online]. Available: http://wiki.cyanogenmod.org/w/All_About_Recovery_Images. [Accessed 15 January 2014].

[54] Android Developers, "Factory Images for Nexus Devices," [Online]. Available: https://developers.google.com/android/nexus/images. [Accessed 15 January 2014].

[55] ZTE Corporation, "ZTE Firefox OS Update Website," [Online]. Available: http://firefox.ztems.com/. [Accessed 15 January 2014].

[56] ClockworkMod, "Recovery Builder," [Online]. Available: http://builder.clockworkmod.com/. [Accessed 16 January 2014].

[57] TeamWin, "TWRP 2.6," [Online]. Available: http://teamw.in/project/twrp2. [Accessed 16 January 2014].

[58] CyanogenMod Inc., "Fastboot Intro," [Online]. Available: http://wiki.cyanogenmod.org/w/Doc:_fastboot_intro. [Accessed 15 January 2014].

[59] Android Revolution HD | Mobile Device Technologies, "Do we really need S-OFF?," 14 June 2013. [Online]. Available: http://android-revolution-hd.blogspot.de/2013/06/do-we-really-need-s-off.html. [Accessed 2014 January 16].

[60] Android Developers, "Activity," [Online]. Available: http://developer.android.com/reference/android/app/Activity.html. [Accessed 31 January 2014].

[61] Android Developers, "Services," [Online]. Available: http://developer.android.com/guide/components/services.html. [Accessed 1 February 2014].

[62] Android Developers, "AlarmManager," [Online]. Available: http://developer.android.com/reference/android/app/AlarmManager.html. [Accessed 23 January 2014].

[63] Android Developers, "Intent," [Online]. Available: http://developer.android.com/reference/android/content/Intent.html. [Accessed 1 February 2014].

[64] SQLite Website, "SQLite Home Page," [Online]. Available: http://www.sqlite.org/. [Accessed 1 February 2014].

[65] Android Developers, "Compatibility Test Suite," [Online]. Available: http://source.android.com/compatibility/cts-intro.html#how-does-the-cts-work. [Accessed 27 December 2013].

[66] Android Developers, "Android Low-Level System Architecture," [Online]. Available: http://source.android.com/devices/. [Accessed 27 December 2013].

[67] SAMSUNG, "S Pen SDK," [Online]. Available: http://developer.samsung.com/s-pen-sdk. [Accessed 27 December 2013].

[68] A. Goldfeld, "Superuser," [Online]. Available: http://www.cs.bgu.ac.il/~arik/usail/concepts/basic-unix-know/superuser.html. [Accessed 28 December 2013].

[69] HTC Developer Center, "Android 2.3 Update for HTC Desire," [Online]. Available: http://www.htcdev.com/process/legal_download/152. [Accessed 28 December 2013].

[70] M. Odersky, "What is Scala?," [Online]. Available: http://www.scala-lang.org/what-is-scala.html. [Accessed 2 February 2014].

[71] Webopedia.com, "ETL - Extract, Transform, Load," [Online]. [Accessed 2 February 2014].

[72] M. Odersky and L. Spoon, "The Architecture of Scala Collections," [Online]. Available: http://docs.scala-lang.org/overviews/core/architecture-of-scala-collections.html. [Accessed 2 February 2014].

[73] Google Inc., "Guava: Google Core Libraries for Java 1.6+," [Online]. Available: https://code.google.com/p/guava-libraries/. [Accessed 2 February 2014].

[74] sbt, "sbt Documentation," [Online]. Available: http://www.scala-sbt.org/. [Accessed 2 February 2014].

[75] Graph Visualization Software, "Graphviz," [Online]. Available: http://www.graphviz.org/. [Accessed 2 February 2014].

[76] Scala Standard Library API (Scaladoc) 2.10.3, "process - Scala Standard Library API (Scaladoc) 2.10.3 - scala.sys.process," [Online]. Available: http://www.scala-lang.org/api/current/index.html#scala.sys.process.package. [Accessed 2 February 2014].

[77] The hsql Development Group, "HSQLDB - 100% Java Database," [Online]. Available: http://hsqldb.org/. [Accessed 2 February 2014].

[78] R. Savage, "BNF Grammar for ISO/IEC 9075:1992 - Database Language SQL (SQL-92)," [Online]. Available: http://savage.net.au/SQL/sql-92.bnf.html. [Accessed 2 February 2014].

[79] Apache Software Foundation, "Apache Ant," [Online]. Available: http://ant.apache.org/. [Accessed 2 February 2014].

[80] Gradleware Inc., "Gradle - Build Automation Evolved," [Online]. Available: http://www.gradle.org/. [Accessed 2 February 2014].

[81] Eclipse Foundation, "Eclipse Downloads," [Online]. Available: https://www.eclipse.org/downloads/. [Accessed 2 February 2014].

[82] Android Developers, "Getting Started with Android Studio," [Online]. Available: http://developer.android.com/sdk/installing/studio.html. [Accessed 2 February 2014].

[83] JetBrains Company Blog, "IntelliJ IDEA is the base for Android Studio, the new IDE for Android developers," 15 May 2013. [Online]. Available: http://blog.jetbrains.com/blog/2013/05/15/intellij-idea-is-the-base-for-android-studio-the-new-ide-for-android-developers/. [Accessed 2 February 2014].

[84] Apache Software Foundation, "Welcome to Apache Maven," [Online]. Available: http://maven.apache.org/. [Accessed 2 February 2014].

[85] Apache Software Foundation, "Apache Ivy," [Online]. Available: http://ant.apache.org/ivy/. [Accessed 2 February 2014].

[86] The Codehaus, "Groovy - Home," [Online]. Available: http://groovy.codehaus.org/. [Accessed 2 February 2014].

[87] QOS.ch, "Simple Logging Facade for Java (SLF4J)," [Online]. Available: http://www.slf4j.org/. [Accessed 2 February 2014].

[88] QOS.ch, "Logback," [Online]. Available: http://logback.qos.ch/. [Accessed 2 February 2014].

[89] QOS.ch, "SLF4J Android," [Online]. Available: http://www.slf4j.org/android/. [Accessed 2 February 2014].

[90] Typesafe Inc. - GitHub, "ScalaLogging," [Online]. Available: https://github.com/typesafehub/scalalogging. [Accessed 2 February 2014].

[91] Android Developers, "logcat," [Online]. Available: http://developer.android.com/tools/help/logcat.html. [Accessed 2 February 2014].

[92] OpenCSV, "Frequently Asked Questions," [Online]. Available: http://opencsv.sourceforge.net/#what-is-opencsv. [Accessed 2 February 2014].

[93] Apache Software Foundation, "Apache POI - the Java API for Microsoft Documents," [Online]. Available: http://poi.apache.org/. [Accessed 2 February 2014].

[94] Microsoft Inc., "MIcrosoft Excel," [Online]. Available: http://office.microsoft.com/en-us/excel/. [Accessed 2 February 2014].

[95] Scientific American, a Division of Nature America, Inc., "How does Bluetooth work?," 5 November 2007. [Online]. Available: http://www.scientificamerican.com/article.cfm?id=experts-how-does-bluetooth-work. [Accessed 17 January 2014].

[96] HTC, "Resetting HTC One (Hard Reset)," [Online]. Available: http://www.htc.com/www/support/htc-one/howto/365398.html. [Accessed 16 January 2014].

[97] Google Play Store, "Autostarts," [Online]. Available: https://play.google.com/store/apps/details?id=com.elsdoerfer.android.autostarts&hl=ru. [Accessed 29 January 2014].

[98] M. Bland and D. Altman, "Statistics Notes - Measurement Error," 29 June 1996. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2351401/pdf/bmj00548-0038.pdf. [Accessed 19 January 2014].

[99] Google Play Store, "Nova Battery Tester," [Online]. Available: https://play.google.com/store/apps/details?id=ru.NanoDinamics.NovaTester&hl=ru. [Accessed 30 January 2014].

[100] Nano Dynamics, "Nova Battery Tester - for Android," [Online]. Available: http://cyberdine.ru/~novatester~/. [Accessed 30 January 2014].

[101] H. Knaust and M. A. Khamsi, "Mean Value Theorems for Integrals," [Online]. Available: http://www.sosmath.com/calculus/integ/integ04/integ04.html. [Accessed 26 January 2014].

[102] Android Tools Project, "New Build System," [Online]. Available: http://tools.android.com/tech-docs/new-build-system. [Accessed 24 December 2013].

7 *Figures*

8 *Tables*

Appendix A – CD Contents

- *\Measurements*
  Raw measurements data (Microsoft Excel files)
- *\Models*
  Models of plastic battery adapters (OpenSCAD files)
- *\PowerEichel*
  Source code of Android benchmark application (Gradle project)
- *\Steps*
  Screenshots how to run Android benchmark application
- *\YAmpy*
  Source code of measurement tool for Yoctopuce Yocto-Amp device (SBT project)