Hochschule Offenburg
University of Applied Sciences

Headless Android Systems for Industrial Automation and Control

Aneeque Hassan, Axel Sikora, Dominique-Stephan Kunz, David Eberlein

***Abstract*—Android is an operating system which was developed for use in smart mobile phones and is the current leader in this market. A lot of efforts are being spent to make Android available to the embedded world, as well. Many embedded systems do not have a local GUI and are therefore called headless devices. This paper presents the results of an analysis of the general suitability of Android in headless embedded systems and ponders the advantages and disadvantages. It focuses on the hardware related issues, i.e. to what extent Android supports hardware peripherals normally used in embedded systems.**

***Index Terms*—Android, Accessory Development Kit (ADK), Android Open Source Project (AOSP), Asynchronous Shared Memory (Ashmem), Microcontrollers (MCUs).**

I. Introduction

Android is an operating system for mobile devices developed by the Open Handset Alliance (OHA). Android enjoys being the market leader in the field of smart phones with a market share of more than 70 % [1]. Android is based on the Linux kernel. So the question arises if there are any additional benefits in using Android instead of Linux for applications outside the smart phone or tablet world, i.e. for applications from industrial, process, or building automation.

The objective of this project was to discuss the suitability, i.e. the advantages and disadvantages of Android for headless systems in industrial automation. In the projected case, the anticipated end product was a system without a local user interface, but with the option to add a GUI for a certain percentage of the applications. This paper will only cover the use case that all applications are Android based and there are no Linux applications. A heterogeneous system (Android and Linux applications) would need further investigations.

Aneeque Hassan, ahassan@stud.hs-offenburg.de, and Axel Sikora, axel.sikora@hs-offenburg.de, are with Hochschule Offenburg, Badstraße 24, D77652 Offenburg.

Dominique-Stephan Kunz, dominique.kunz@ch.sauter-bc.com, and David Eberlein, david.eberlein@ch.sauter-bc.com, are with Fr. Sauter AG, Im Surinam 55, CH4058 Basel.

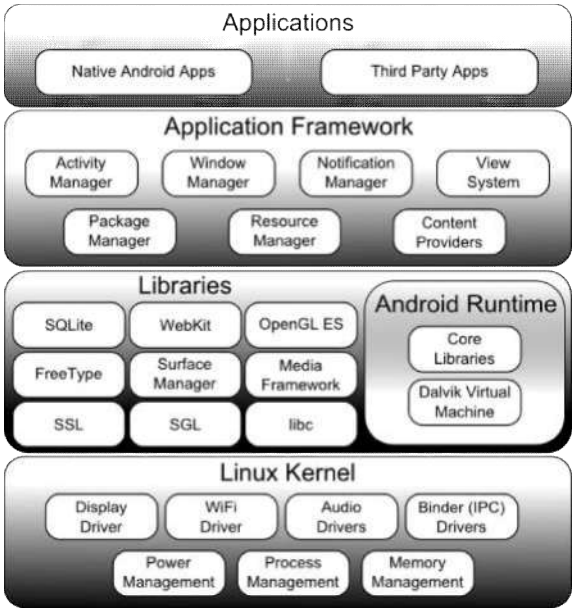

Figure 1: Architecture of Android [2].

A basic question in this context is the support for external hardware peripherals. Specifically built for smart phones Android does not need to provide generic support for external interfaces like SPI, UART, GPIO or similar peripherals as opposed to "normal" embedded systems.

In this paper first the architecture of Android is presented in ch. II and the concept of headless Android is introduced in ch. III. Then the used baseboard is shortly presented in ch. IV. The extension of Android based systems with additional local hardware interfaces was the most important part of the project. The mechanism of hardware support in Android is discussed in ch. V. Then the timing results of the test setup are presented in detail in ch. VII and ch. VIII. The report is concluded by giving an opinion based on the findings during the project.

II. Android Architecture

Android is based on the Linux kernel, a number of software libraries written in 'C', a Java-like virtual machine for the execution of application code, and a full suite of mobile phone applications [3]. Fig. 1 shows an overview of the Android architecture.

The Linux kernel in the Android is a modified version of the kernels available from the Linux kernel archives, often also called vanilla Linux kernel [4]. These modifications include Alarm, Asynchronous shared memory (Ashmem), Binder, Power Management, Low Memory Killer, Kernel Debugger and Logger [5]. Android uses the Linux kernel to control the underlying hardware.

III. Android for Headless Embedded Systems

Headless embedded devices do not have a local user interface like mobile devices, which Android was originally created for. Android for headless devices could therefore omit those components that are used to draw and control the user interface. The components, which are not required for the headless-only use case, are SurfaceFlinger, Window-Manager, Wallpaper-Service and InputMethodManager [6].

IV. The Base Board

The selection of the base board is an important step, as there are a lot of aspects to be considered in this process, especially in the case of porting Android to a new device. The parameters to be taken into account are SoC features, community, cost, features, expandability, availability, licensing, catalogue part, and software support [7].

In the given project the focus was not on the porting of Android, but on the features it offers. The project work was started by testing requirements on the Raspberry Pi [8] and then continued on the "Tsunami Pack" by TechNexion [9]. The Raspberry Pi was not used further because the Android builds available for it are still not stable. Figure 2 shows the board used in the setup of the test environment.

V. Hardware Support Mechanisms

A. General Architecture

Many embedded devices need to collect data from local sensors, analyze and process the data, and control local actuators accordingly. In some cases, they are connected to backend control and monitoring systems for proper operation. All these embedded scenarios require different hardware interfaces like GPIO, SPI, UART PWM or alike. This connectivity between sensors and actuator devices through local hardware interfaces is normally not required in smart phones. The Android architecture has been optimized for smart phones, and thus, only includes a minimum of hardware drivers required for smart phones [10].

Figure 2: TechNexionTsunami Board running Android.

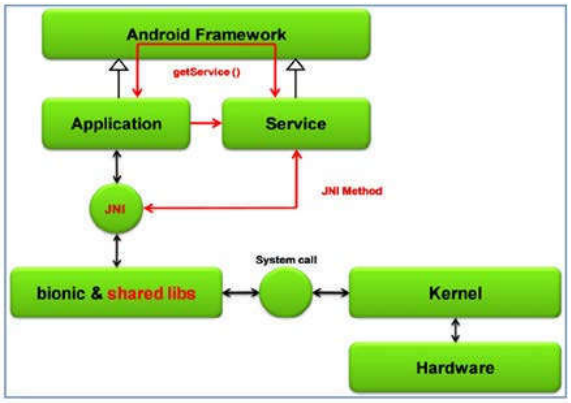

Figure 3: Android hardware access mechanisms [11].

In the changes made to the standard Linux kernel for Android, the approach of hardware support has also been changed. Standard Linux provides access to hardware interfaces via device drivers. In Android hardware access is typically supported through shared libraries provided by manufacturers. Most software stacks typically found in Linux distributions that enable interaction with local hardware are not found on Android [11]. Figure 3 shows how hardware is accessed on Android.

Unlike native Linux applications, Android applications communicate with the underlying hardware through Java APIs, direct system calls are not possible. Android defines APIs which can be utilized by applications to communicate with the underlying hardware. A system service is used to load and interface the shared libraries and provide the API functionality to the application. These system services consist of two parts. One is written in Java and the other one in C. The Java part implements the Android side of the service and the C part communicates with the shared library.

Consequently, there exist multiple ways to add local hardware interfaces to an Android run-time system:

- Add shared libraries to the source code and make a custom port to Android (cf. ch. B.1)).
- Use the Android Open Accessory Mode (cf. ch. B.2)).
- Use external MCUs connected over USB or Bluetooth (cf. ch. B.3)).

B. Accessing local interfaces from Android

1) Android Porting

This solution requires adding the needed hardware drivers to the Linux kernel and applying the Android patch to the kernel as a first step. As a second step, the shared libraries should be made available in the source code of Android. This custom version must then be ported to the target device. The best part of this approach is that after the accomplishment of this task only application level programming would be required to make use of these peripherals like the USB and WiFi APIs are used in Android normally. The downturn of this approach is that detailed knowledge of the hardware would be required, which mostly calls for support from the hardware vendor.

Generally speaking, this approach fundamentally contradicts the overall Android approach, which strives to avoid complex Linux programming and tries to replace it with comparatively easier Java programming. In the end this approach would double the work, because first Linux programmers would build their own build of Android to include the libraries in the OS and then the Java developers would use these libraries in the application development phase. This would also increase the initial development cycle of the product but with a much more stable result.

2) Android Open Accessory Mode

The second option to add hardware interfaces comes from the Android Open Accessory mode, which allows the connection of peripherals to an Android platform where the Android is the USB device and the peripheral (accessory) is the USB host [6]. This avoids the necessity of new hardware drivers.

Google implemented its own hardware by the name Android Accessory Development Kit (ADK) as a reference implementation to build hardware accessories for Android. The ADK is based on the Arduino open source electronics prototyping platform [12]. Android Open Accessory [13] protocol is used to communicate with the Android device over a USB or a Bluetooth connection. This mode is not supported on a lot of devices and as reported by various senior developers on communities it is not so robust [14].

3) External Microcontrollers

The third possible solution is very similar to the one used by Google for their Google ADK. It uses the USB or Bluetooth interfaces of the Android device to attach an external MCU, where the MCU implements and provides the actual connectivity to the local interfaces. There are external boards already available on the market, like the FT311D from FTDI [15] or the IOIO from Sparkfun [16]. These boards are delivered with a custom API library which may be included into an Android application and provides access to the boards peripherals. The good side is that for these boards the development time decreases considerably, compared to integrating Linux drivers.

C. Evaluation of methods to access local interfaces

If a company develops their own board and ports Android on their own, then they will write their own drivers for their system. This is the best way to offer these interfaces to the application. If a company wants to use an off the shelf board with Android already running on it but the required interfaces are missing, the Android Open Accessory Mode or the external MCU approach will be the approach with less effort.

The latter use case is interesting for this investigation because the TechNexion board does not support the drivers needed for the hardware. Furthermore the ADK is not orderable. Therefore, it was decided to go with the external MCU approach as a solution to add peripheral support to Android. The IOIO board was used for this work.

The board is created by a day-time Google employee as a hobby project and is sold by Sparkfun. The board connects to the Android device in debug mode using USB OTG or Bluetooth. An Android application can utilize the IOIO libraries to access functionalities on the board. No embedded programming is required. Another good thing is that it works with all Android versions above 1.5.

VI. HARDWARE FOR THE TARGET SYSTEM

The target platform requires a lot of hardware interfaces to work. In this section each interface will be discussed one after the other.

A. Tsunami Mainboard

1) SD-Card

As large memory storage is required to store logged data, an SD-Card could easily solve this requirement. The SD-Card support is provided by default in Android and is fully functional on the Tsunami board.

2) UART

A serial UART-interface is still an important way to access hardware peripherals on embedded systems. There are no standard APIs present in Android to read and write to Linux serial ports. However, there is an open source project which makes it possible to read and write data through the serial ports of Linux. The

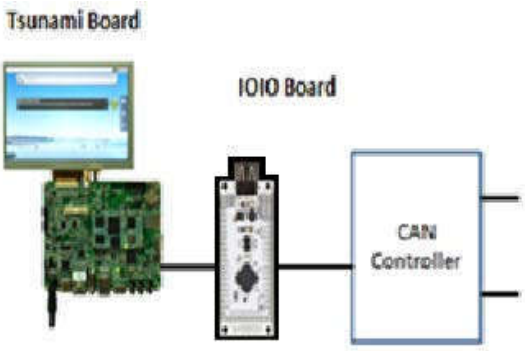

Figure 4: Block diagram of CAN Controller connected to the Android device using IOIO.

project name is "android-serialport-api“ [17]. The serial ports on the Tsunami board can be used with this api. Moreover, the Modbus [18] protocol can be used. The application level protocol operates directly on top of a serial interface and serial communication standards [19].

3) GPIO, SPI and CAN

The support for other peripherals is present on the Tsunami board. However, no out of the box solutions or precompiled libraries exist for them. The IOIO board was used to provide the other peripherals required for the system.

B. Using IOIO Board

1) GPIO

All 48 pins on the IOIO board can be configured and used as GPIO's. For this both digital and analogue input/output tests were performed. For the digital output an LED was successfully controlled with the IOIO board. Using the IOIO board implementing the GPIO functionality is quite straight forward.

2) SPI

There is a special class in the IOIO library called "SpiMaster.java". Controlling the IOIO SPI interface is done via the SpiMaster class. An instance of this interface corresponds to a single SPI module on the board, as well as to the pins it uses for CLK, MOSI, MISO and SS (one or more) [20].

In the test setup it was tested to make the SPI work in the Master mode using a fixed buffer of 0x55 and it transmitted successfully. During the tests the transmitted signal was good up to 1 Mbps but it started distorting after that and it worked till 3.2 Mbps. At higher speeds, the output signal degraded significantly.

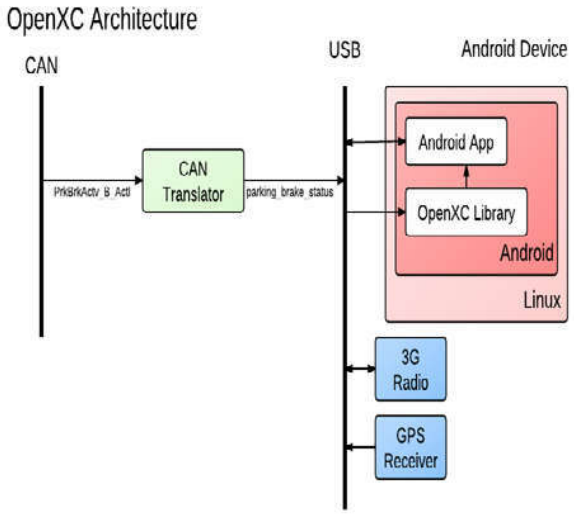

Figure 5: OpenXC architecture [22].

3) CANbus

CAN Bus [21] is also not natively supported on Android. One possible solution is to use the SPI of IOIO board to control a CAN controller from the Android device. Such a CAN controller is provided by Microchip, MCP2515 [23]. Figure 4 shows the block diagram of the proposed solution.

Another interesting solution is the OpenXC architecture [22] developed by Ford and Bug labs. It connects the CAN bus to the Android device over USB using a CAN translator in between. Figure 5 shows the overall architecture. The Android libraries of the project are available on the projects webpage.

4) UART

The IOIO Board also offers a UART interface, which can be used in a similar way as the Tsunami boards UART.

VII. IOIO TIMING TESTS

The purpose of the tests was to test the round trip time of communication from Android to the IOIO board and back. The idea was to measure the delays that incurred during this communication process. These tests were performed to assess the setup's feasibility for target systems communication timing requirements, which are very strict for automation control systems. The goal was to achieve a signal cycle time of less than 12.5 ms because it sufficed the timing requirements of the target automation systems.

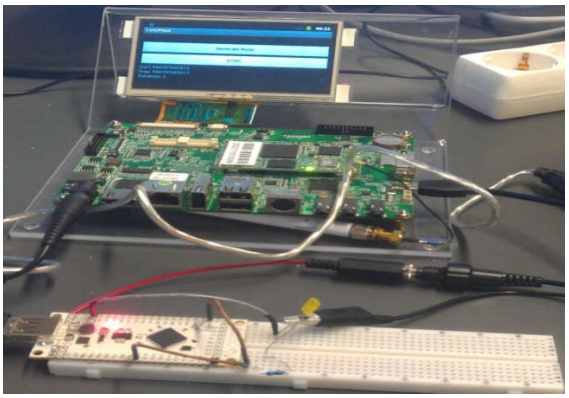

Figure 6: The hardware setup for the tests.

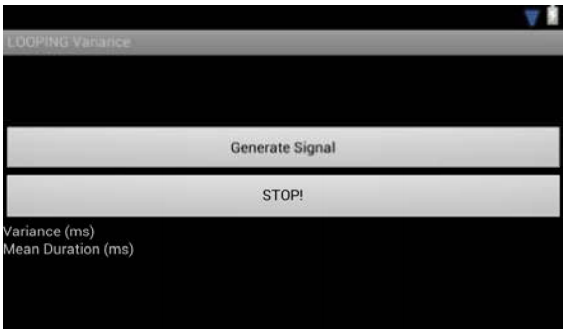

Figure 7: The screen shot of the application.

A. Hardware Setup

The set-up includes the Tsunami board running Android connected to the IOIO board through USB in Android Debug Bridge (ADB) mode. The other possibility would have been a connection over Bluetooth which would additionally include its own delays. Therefore the direct physical connection via USB was used instead.

For the hardware, one pin (#10) of the IOIO board was set as digital output in the application, which turns on an LED, when set to true (on), and turns it off, when set to false (off). The output of this pin (#10) is given as input to another pin on the board (pin #33) which is configured as digital input. With this setup a hardware loop is created on the IOIO board, which forms a cycle of on and off signals like a ring oscillator. An oscilloscope was used to monitor the signal across the LED, which was the desired repeating digital signal. Fig. 6 shows the hardware setup for the tests.

B. Software Application

The software uses the IOIO library and turns on the digital output pin when the "Generate Signal" button is pressed in the application. If the logical state of the input pin is changed, then this state is inverted and set to the output pin. The process of changing the logical state of the output pin until the input pin represents this logical state is one cycle. By pressing the “Generate signal” button this process runs in a loop until 1000 if these cycles are run through and subsequently the mean value of the readings and the variance of the read values are displayed using the TextView widget available in the Android API. Fig. 7 shows a screen shot of the application that was developed to perform the tests.

C. Measurements

Two types of measurements were taken for the tests. One was software timing in the application itself for one complete cycle off on and off (which means two cycles of the IOIO board). The other one was using an oscilloscope across the LED in the hardware setup to measure the time period of one signal cycle.

The software measurements could be halved and compared to the duration of one signal cycle. Alternatively two signal cycles (on and off) could be considered as one software cycle duration and the timings could be compared. The second option of taking two consecutive on and off signals was chosen. The results are discussed below.

D. Test Cases

1) Different Load Conditions

One important aspect is the dependency on the load conditions of the CPU, i.e. from other processes running quasi-simultaneously in the Android device. Three load conditions were tested to generate a set of results. The first one was to run the test application by itself, with no other applications running. The second one was with a medium load condition with some other applications running in the background. The third one was a high load condition with even more applications running in the background.

2) Sleep Function

For another test scenario the sleep function was introduced in the loop test. In each operation the looping thread of the IOIO test application is put to sleep for different amounts of milliseconds. After the time expires the thread is available again for scheduling and is executed [24]. This was done to get an idea of the delays that are incorporated by releasing and reacquiring the processor.

3) Run as Service

Another test scenario was to do the timing tests by communicating with the IOIO board in an Android service. A service is just like a foreground application (activity). The only difference is that it does not have a user interface and runs only in the background without any user interaction. The looping test was the same as before, this time only the application was run as a

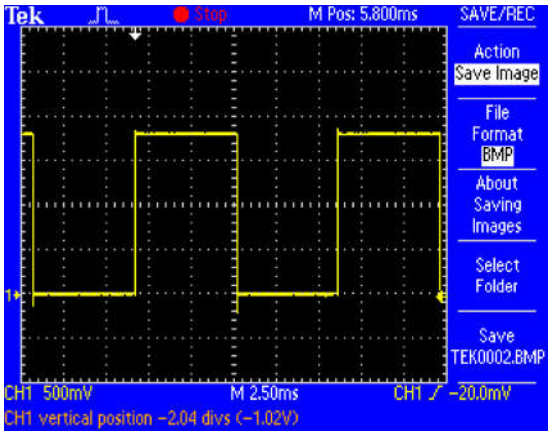

Figure 8: Oscilloscope measurements for the low load scenario.

Figure 9: Software application readings for the low load scenario test.

service, everything else was the same. Further three sub scenarios for different load conditions were tested which included the following:

- Run only the IOIO service.
- Run the service plus the music player application continuously running in background as a service.
- Use the service, the music player playing songs plus playing a game at the same time

VIII. Results

A. Tests without sleep

On average a delay of 12 ms was experienced for the two signal cycles. From the measurements it was seen that adding the load didn't really affect the average duration a lot (at least in milliseconds) but the variance did increase when increasing the number of applications running. Using a web server during the IOIO operation increased the mean duration as well, which was obvious due to processor scheduling. But apart from only those specific times the oscilloscope readings were the same. This explains Android scheduling as well. The foreground activities have most (~90 %) of the CPU time whereas background activities share only the remaining resources (~10 %) [25]. Fig. 8 shows the oscilloscope image of one of the tests being carried out for the low load scenario.

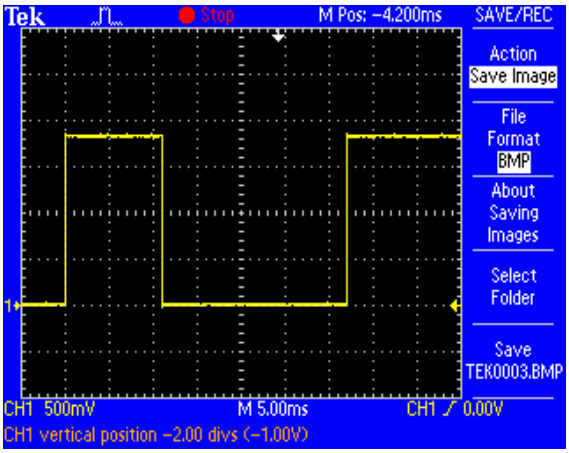

Figure 10: Oscilloscope measurements for the high load scenario and 10 ms sleep time.

Table 1. Repeated readings for the low load test scenario.

Application with no sleep (low load)			
test #	mean duration (ms)	variance (ms)	oscilloscope reading (ms)
1	12.09	0.28	12
2	12.21	0.49	12
3	12.23	0.54	12

Fig. 9 shows the software results of the timing test for the same scenario presented above. Table 1 shows the timing measurements for three repeated tests for the same scenario, which are consistent enough to deduce results. Readings were noted for the other test scenarios as well, but are omitted here due to limited space.

B. Tests with sleep

For this set of measurements a huge variation between the oscilloscope readings was observed. Also when compared to the readings of the software application the readings don't match as for the test measurements in the "no sleep scenarios". However, the mean values are consistent between different tests. This indicates that the oscilloscope reading depends a lot on the point in time when the oscilloscope was stopped to take a measurement. Whereas the difference between the mean value in the software application and the measured oscilloscope values presumably are caused by the increased variance. The mean values didn't change a lot in the computed results but the variance increased a lot.

The results also depict that the delays added by adding more load to the processor are small enough to remain in the given timing requirements. Adding the sleep functionality increases the timing of the whole setup by almost the same amount as the sleep function puts the thread on a hold.

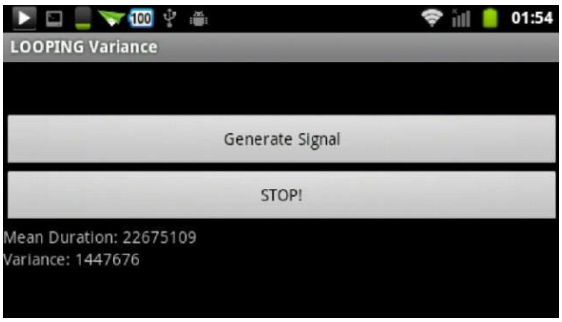

Fig 11: Software application readings for the high load and 10 ms sleep scenario test.

Table 2: Repeated readings for the high load test scenario.

Application with 10 ms sleep (high load)			
test #	mean duration (ms)	variance (ms)	oscilloscope reading (ms)
1	22.67	1.45	32 (varying)
2	22.55	1.28	32 (varying)
3	22.28	0.91	32 (varying)

Fig. 10 shows the oscilloscope image of one of the tests being carried out for the high load scenario and a sleep time of 10 ms. Fig. 11 shows the software results of the timing test for the same scenario presented above. Table 2 shows the timing measurements for three repeated tests for the same scenario.

In summary if a process is released by the CPU and it comes back after some time, the delays incorporated are still bearable according to the timing requirements. Adding the load also doesn't affect the communication in an adverse manner. The sleep function only adds delays roughly of the passive time of the sleep function.

C. Test of Running as a Service

The same logic of testing was applied in another Android program which was run as service under different load conditions. From the oscilloscope images it was clear that the variance is so high at different times that the mean delay of communication has also increased to a high number. These high numbers will not meet the desired communication timing requirements so running the application in the foreground as an activity would be advisable to stay under the communication delay requirements. Fig. 12 shows the oscilloscope image of one of the tests being carried with the music service and game a being run simultaneously. Fig. 13 shows the software results of the timing test for this very scenario.

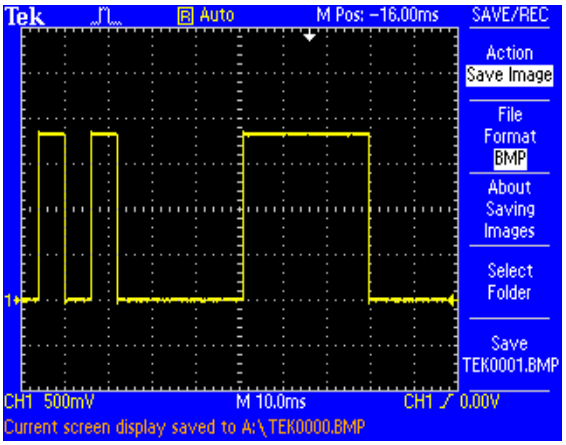

Fig 12: Oscilloscope measurements for the service, music service and game run together.

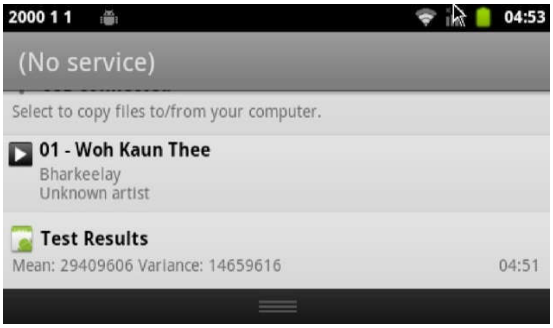

Fig 13: Software application readings for the service, music service and game run together.

It is interesting that when the service is running alone, it experiences significantly larger mean values and variances in comparison to the situation when other services are simultaneously executed on the Android system. For the moment, it can just be assumed that the power management has an impact also on the scheduler. Perhaps the scheduler calls the background services more often if there is more than one service active, which causes this unpredicted behavior.

IX. Conclusion

The target systems are headless systems with the option to add a local GUI to the system if required. These headless systems require different hardware interfaces than the ones used in Android smartphones. The standard procedure of integrating hardware interfaces in Android is a bit complex but there are workarounds available which were presented in this paper. The delays these elements add to the whole communication setup are still bearable in most scenarios as proved from the tests performed during this project.

From the work it was seen that Android is set to join the embedded world. If the target systems are headless with a small percentage being GUI enabled as well, it is recommendable to use Android as the OS. This will give the advantage of having a single platform with

slightly different features. This would reduce the development phase and costs. To summarize we think that using Android in headless devices in industrial automation is attractive, as soon as they have optional GUIs.

REFERENCES

[1] http://mashable.com/2012/11/14/android-72-percent/. Retrieved 3 April 2013.

[2] http://www.techotopia.com/index.php/. An_Overview_of_the_Kindle_Fire_Android_Architecture. Retrieved 3 April 2013.

[3] http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Porting-Android-to-a-new-device/. Retrieved 3 April 2013.

[4] http://www.kernel.org/. Retrieved 3 April 2013.

[5] F. Maker, Y.-H. Chan, *A Survey on Android vs. Linux,* http://handycodeworks.com/wp-content/uploads /2011/02/linux_versus_android.pdf. Retrieved 3 April 2013.

[6] http://www.opersys.com/blog/headless-android-1. Retrieved 3 April 2013.

[7] K. Yaghmour, *Embedded Android Porting, Extending, and Customizing,* O'Reilly Media, 2011.

[8] http://www.raspberrypi.org/quick-start-guide. Retrieved 3 April 2013.

[9] http://technexion.com/index.php/products/arm-cpu-modules/ti-omap3530/tsunami. Retrieved 3 April 2013.

[10] FTDI Whitepaper, *Connecting peripherals to an Android Platform*. Retrieved 3 April 2013.

[11] http://sujaiantony.wordpress.com/category/android-2/. Retrieved 3 April 2013.

[12] http://www.arduino.cc/. Retrieved 3 April 2013.

[13] http://source.android.com/tech/accessories/aoap/aoa2.html. Retrieved 3 April 2013.

[14] https://groups.google.com/forum/?pli=1#!searchin/ioio-users/frequency/ioio-users/EIEGF2h8Y8c/uAutFXUPvjYJ. Retrieved 3 April 2013.

[15] White paper, FTDI, USB Android Host Module.

[16] S. Monk, *Making Android Accessories with IOIO,* O'Reilly Media, 2012.

[17] http://code.google.com/p/android-serialport-api/. Retrieved 3 April 2013.

[18] http://www.modbus.org/. Retrieved 3 April 2013.

[19] http://jamod.sourceforge.net/kbase/protocol.html#sub_serial. Retrieved 3 April 2013.

[20] https://github.com/ytai/ioio/wiki/SPI. Retrieved 3 April 2013.

[21] http://www.can-cia.org/index.php?id=46. Retrieved 3 April 2013.

[22] http://openxcplatform.com/getting-started/index.html. Retrieved 3 April 2013.

[23] Data Sheet, MCP2515, Microchip.

[24] https://github.com/keesj/gomo/wiki/AndroidScheduling. Retrieved 3 April 2013.

[25] http://stackoverflow.com/questions/8081042/sleep-function. Retrieved 3 April 2013.

Aneeque Hassan was awarded the degree of Bachelors of Science from FAST-NUCES University in Lahore (Pakistan) in 2009. He is currently a student of M.Sc. in Communication and Media Engineering in the Hochschule Offenburg and works in the Embedded Communications Lab with Prof. Dr. Axel Sikora.

Axel Sikora holds a diploma of Electrical Engineering and a diploma of Business Administration, both from Aachen Technical University. He has done a Ph.D. in Electrical Engineering at Fraunhofer Institute of Microelectronics Circuits and Systems, Duisburg. After various positions in the telecommunications and semiconductor industry, he became a professor at the Baden-Wuerttemberg Cooperative State University Loerrach in 1999. In 2011, he joined Offenburg University of Applied Sciences, where he holds the professorship of Embedded Systems and Communication Electronics. His major interest is in the system development of efficient, energy- aware, autonomous, secure, and value-added algorithms and protocols for wired and wireless embedded communication. He is also founder and head of Steinbeis Transfer Center Embedded Design and Networking (stzedn).
Dr. Sikora is author, co-author, editor and co-editor of several textbooks and numerous papers, as well as conference committee member to various international conferences in the field of embedded design and wireless and wired networking.

Dominique Stephan Kunz studied electrical engineering (with a focus on control and automation engineering) at the University of Applied Sciences in Basel. Since 1999, he has been working for the company Fr. Sauter AG in Basel. He has over 10 years of experience in industrial product development and project management in the heating, ventilation and air conditioning market with respect to building automation.
Currently, he heads the Technologies department, which deals with research projects, and he is responsible for cooperation with universities.

David Eberlein graduated as Bachelor of Engineering in Information Technology at the Cooperative State University Lörrach in 2009, winning the prize of the city Lörrach for the best bachelor thesis of the year.
Since then, he has been working for Fr. Sauter AG as Software Engineer at the Technologies department. His main interests are web based graphical user interfaces, as well as mobile applications.