Investigation of the deployment of Android as a user interface for ovens
Master's Thesis at Ulm University

Author:

Patryk Boczon

patryk.boczon@uni-ulm.de

Reviewers:

Professor Doctor Manfred Reichert

Professor Doctor Martin Theobald

Supervisors:

Marc Schickler

Michael Lamers

Year:

2015

Version October 20, 2015

Abstract

This theses is conducted in cooperation with BSH Hausgeräte GmbH. The target is the investigation of the applicability of *Android* as an operating system for home appliances, specifically for ovens. In order to draw conclusions in regard of applicability, three major topics will be investigated.

For a start, the performance of *Android* running on moderate target hardware will be analysed. The focus lies on graphics, since a high quality graphical user interface is most likely to be the crucial point in terms of performance. Providing a smooth and responsive graphical user interface is decisive for a satisfying user experience.

Furthermore, originating from the mobile domain, *Android* requires an array of modifications prior to being embedded into an oven. The goal is to identify these potential aspects that need modification and give appropriate solutions, thus also providing an estimate of the required effort for the embedding process.

Finally, potential inter-process communication mechanisms will be investigated with the objective to identify the most eligible method(s) for the communication between an *Android* application and the underlying oven hardware.

Gratitude

This thesis was conducted at Ulm University in cooperation with BSH Hausgeräte GmbH. Acknowledgements go to both facilities for having made this thesis possible in the first place. Special thanks go to Michael Lamers from BSH Hausgeräte GmbH and Marc Schickler from Ulm University for having provided competent supervision alongside the development of this thesis. Further acknowledgements go to Professor Doctor Manfred Reichert and Professor Doctor Martin Theobald from Ulm University for reviewing this thesis.

Contents

1

Introduction

With a constant rise in functionality and intercompatibility, even everyday objects such as home appliances will become more and more intricate. As the complexity of the software of such home appliances raises, it is desirable to be able to code in a high level programming language such as Java. Software modules should not only cover one particular type of home appliances, such as ovens, but also hobs, hoods coffee machines etc. The use of such generic software components proves more expandable for prospective projects and increases intercompatibility. As a result, a more abstract solution to common problems will probably be used in the future which in turn implies a further raise in complexity.

With the increasing number of rich user interface projects, the outlook to reuse code is a huge benefit. This is especially true for software developed for one particular brand where, due to consistency, user interface components will probably feature similar visuals and similar behavior. However, software for particular devices providing a rich user interface had often been developed from scratch in the past.

A well-conceived graphical user interface (GUI) is essential in order to render the rich functionality of upcoming devices easily accessible for the user. With the prevalence of touch screens it seems reasonable to provide a touch based modality for objects with rich functionality, not least due to consistency. A (touch screen based) GUI is significantly more dynamic than a physical interface and thus is able to provide access to diverse functionalities in an optimized manner. The question then arises as to why not embed *Android* into home appliances. Besides the touch interface, the *Android* software development kit (SDK) provides (means to build) generic software solutions for common problems.

This thesis investigates the applicability of *Android* to be embedded into an oven by *Bosch Hausgeräte GmbH* (BSH). Subject of this thesis is an oven model from the series 8 by BSH (see figure 1.1) which offers a large variety of capabilities and options.

Figure 1.1: The user interface of a series 8 oven by BSH. All three display sections as well as the ring in the center are touch sensitive [7].

1.1 Motivation for Android

Android as an operating system (OS) brings along several benefits.

For one thing, due to its **popularity**, *Android* will most likely sustain yet for a long time on the market. This ensures constant updates. With its popularity, *Android* provides a very well supported development environment and the chance to come across solutions to problems in literature or the internet is fairly high. Plus, skilled developers are rather easy to find.

Considering System on a Chip (SoC), there is strong **vendor support** for *Android*-supported hardware.

For another thing, the *Android* SDK utilizes Java which is a **high level programming language** and thus reduces development costs over low level programming languages [4].

Additionally, when developing applications for smartphones/tablets which are designed to communicate with specific home appliances, **code** can be **reused** (e.g. GUI code). In doing so, effort can be saved while consistency is ensured.

Furthermore, *Android* is **optimized for restricted hardware**. This comprises for instance the GUI framework and runtime which are optimized for speed and automatic usage of the graphics processing unit (GPU) [14]. The fact that Java can be used for development comes at the cost of a virtual machine, which implies losses in performance. The *DalvikVM* was explicitly designed for *Android* with performance in mind [3].

Moreover, *Android* features **intercommunicational abilities**. The *Android* SDK entails for instance application programming interfaces (APIs) for WLAN, Ethernet, Bluetooth and USB which could notably simplify interconnections between different home appliances and/or (mobile) devices.

Further on, *Android* supports **over-the-air updates** for both system and application software. This feature is very likely to be used in the home appliances context, as well. The *Android* app model could serve as a package management foundation. In the future, a requirement for allowing functional packages to be installed optionally might come up. The app installation and communication model in *Android* could be used to implement this requirement.

Finally, besides the *Android* SDK, there is a vast number of **open source libraries** which facilitate development for *Android*, e.g. in terms of intercommunication, GUIs, etc.

1.2 Subject Hardware

The Hardware used in this thesis is the AM335x Evaluation Module (TMDXEVM3358) by *Texas Instruments* (see figure 1.2). As illustrated in table 1.1, the evaluation module has rather limited resources. For that matter, the performance analysis conducted in chapter 2 is a crucial aspect for the determination of the suitability of *Android* as an operating system in the first place. The latest *Android* version that is available on the evaluation module is *Android Jelly Bean* 4.1.2.

Figure 1.2: TMDXEVM3358 - AM335x Evaluation Module [20]

Hardware	Software	Connectivity
AM3358 ARM Processor 1GB DDR3 TPS65910 Powermanagement IC 7" touch screen LCD	Linux EZ SDK Android	10/100 Ethernet UART SD/MMC USB2.0 OTG/HOST Audio in/out JTAG CAN

Table 1.1: An overview of specifications of the AM335x Evaluation Module [20].

1.3 Thesis Objective

The purpose of this thesis is to describe a conceptual approach on diverse topics that are relevant for embedding *Android* as an OS into an oven by *BSH*. The results should provide implications on the practicability and effort of such an embedding process.

As *Android* originates from the mobile domain [38], there are several aspects that require modification prior to embedding *Android* into a stationary oven. Besides these modifications concerning the behavior of *Android*, adding support for further hardware is an essential task for such an embedding project. This thesis examines the feasibility of such modifications and also provides approaches on how to achieve the required

objectives. Consequently, the effort for the required work of such an embedding process can be assessed.

1.4 Thesis Structure

This thesis is composed of three main chapters.

The focus of chapter 2 is to determine the applicability of *Android* as on OS in regard of performance. As the potential performance bottleneck is most likely to be graphical, a performance analysis is conducted in this chapter. In order to be able to draw realistic conclusions from the results, this performance analysis was executed on the given evaluation module (see chapter 1.2) with realistic user interfaces.

Chapter 3 discusses the necessary modification of the behavior and diverse features that are immanent in the *Android* OS in order to be applicable for an embedded scenario. The purpose of this chapter is to draw the attention to potential features that require modification, give solutions to these features and provide a general overview of the required work that is necessary for this step of the embedding process.

Chapter 4 focuses on diverse, potentially eligible inter-process communication (IPC) mechanisms to establish a communication between the *Android* application and an oven hardware module/driver. The *Android* application is supposed to be the oven's user interface and thus should be able to control the oven hardware and reflect its status.

Finally, a conclusion is given in chapter 5 which recapitulates the most relevant topics and results.

2

Performance Analysis

A smooth and responsive GUI is essential for a high quality user interface. This is especially crucial for touch-based GUIs since users might compare the user experience with what they already came to know from their smartphones. As a result, users will immediately register losses in performance.

However, in the context of an oven, touch responsiveness might be hampered due to specific requirements such as the usage of components that meet the heat criteria, appliance design restrictions (e.g. a thick glass front) or simply cost reduction plans.

Further on, most home appliances, such as ovens, will not be replaced as frequently as mobile devices. Consequently, home appliances will not feature cutting edge hardware in the long run.

Despite these drawbacks, modern home appliances feature rich capabilities and their GUIs have to render this functionality in an accessible manner and at the same time meet the level of quality of the product. Besides the aesthetic aspect, images and animations provide visual clues and feedback on actions.

2.1 Graphics in Android

Before starting with the actual implementation of the performance analysis, looking into the drawing/rendering process of *Android* seems worthwhile [33].

In *Android*, in order to draw content on the screen, for instance in case an application comes in focus, the *WindowManager* invokes the *SurfaceFlinger*. The *SurfaceFlinger* accepts and composites buffers of graphical data from multiple sources and forwards these

to the display. Since *Android* version 3.0 the *SurfaceFlinger* delegates the composition of the buffers to the *Hardware Composer*. The *Hardware Composer* is device-specific and determines the most efficient way that buffers of graphical data can be composited on the given hardware.

In the terminology of the *SurfaceFlinger*, a layer is for instance the status bar at the top of the screen (see figure 3.4), the navigation bar that holds the virtual buttons at the bottom of the screen (see figure 3.3) and the UI of the application. While the status and navigation bars are rendered by the system, the application renders its own content. Furthermore, layers can be updated independently.

In order to prevent screen tearing, vertical synchronization (VSYNC) is considered. This implies that the screen will only be updated during the period between the drawing of two frames. When it is safe to update the screen (meaning VSYNC), the *SurfaceFlinger* iterates through the layers and checks for new buffers. If there is no new buffer for a layer, the previous buffer will be used. Figures 2.1 and 2.2 illustrate the flow of an application's buffer data.
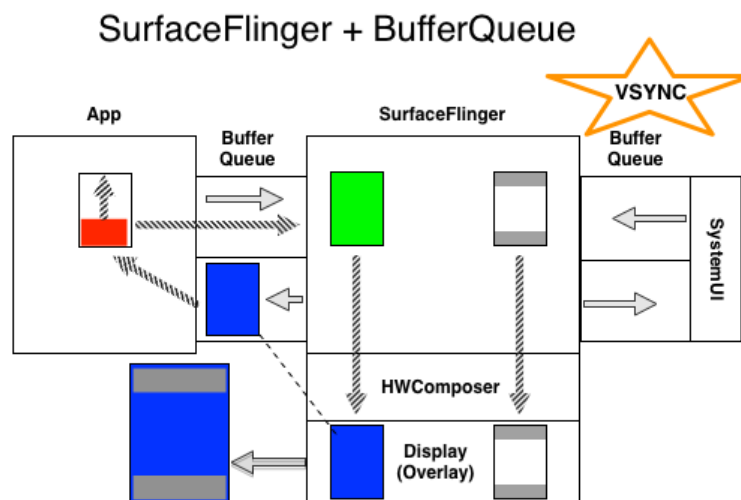

Figure 2.1: A diagram of the flow of buffer data between an application, the *SurfaceFlinger*, the *Hardware Composer* and the display [33].

The red buffer in figure 2.1 fills up and is transmitted to the *BufferQueue*. The blue buffer within the *BufferQueue* represents at that time the previous frame of the application

and takes the place of the next potential frame within the app. This ensures that as long as the application does not intend to display anything new, the previous content will be rendered. Once the VSYNC signal is dispatched, the *SurfaceFlinger* receives the red buffer from the *BufferQueue* and delegates the green buffer to the display. The green buffer was created by the application prior to the red buffer. At the same time, the *BufferQueue* receives the green buffer as a potential next buffer. Figure 2.2 illustrates the next frame where the app is about to draw a purple screen.
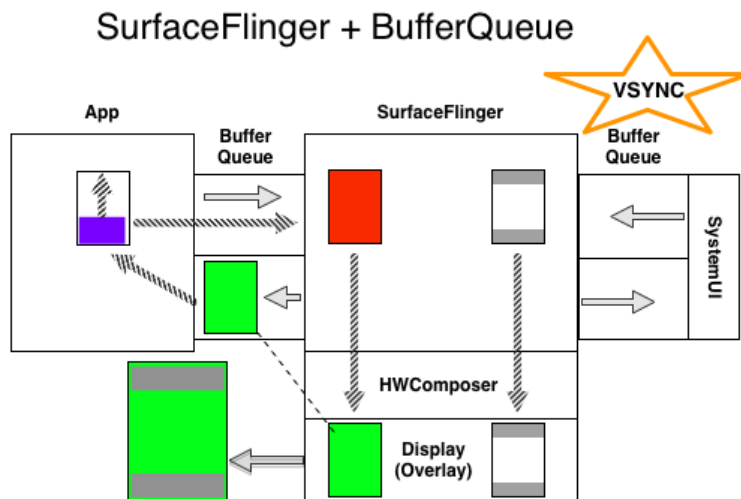
Figure 2.2: The state of the diagram of figure 2.1 after one frame (according to [33]).

The SystemUI's part is simplified in these diagrams. In reality the SystemUI would have two *BufferQueues*, one for the status bar and one for the navigation bar, each with a respective size.

In *Android*, the UI is composed of elements that are ultimately derived from *Views*. The application's UI thread is responsible for the layout and renders the content on a *Surface* that was created by the *SurfaceFlinger*. Such a *View* based implementation will be the first variant of the upcoming performance test.

When utilizing *SurfaceView* which is a specific implementation of *View*, the *Surface-Flinger* creates a new distinct *Surface* for it. The *SurfaceView* itself is completely transparent and its contents will not be composited by the application but rather by the

SurfaceFlinger directly. Consequently, this new *Surface* can be rendered in a separate thread and can be updated via different mechanisms, e.g. using a video decoder, the OpenGL API etc. This approach is more direct and will be implemented in the second variant of the performance test by utilizing the *LibGDX* framework that makes use of a *(GL)SurfaceView* implementation [33].

2.2 Implementation

In the following, a section of the user interface specification, provided by BSH, will be implemented on *Android* and executed on the evaluation module (see 1.2). The selected section will be implemented via *Android Views*. In the *Android* SDK, *View* represents the base class for user interface components (widgets). The *ViewGroup* class is a subclass of *View* and can contain other *Views*. Consequently, the *ViewGroup* is the base class for layouts in *Android* [19]. Since this performance analysis is conducted on *Android Jelly Bean* 4.1.2 (API level 16) and hardware acceleration for the *Android* 2D rendering pipeline is enabled by default since API level 14, there is no need to activate it manually [14].

Aside from that, an *OpenGL ES 2.0* based variant of the same content will be implemented using the open source framework *LibGDX*. In doing so, both variants can be compared afterwards in terms of performance. Such an investigation will lead to conclusions about deviations in performance between the two implementations. Furthermore, assumptions about the applicability of the evaluation module (see chapter 1.2) in terms of performance can be derived from the resulting data.

Both GUI implementations feature similar principles in terms of hierarchy. While the *Android View* based implementation utilizes *ViewGroups*, such as *Layouts* to hold further *Views*, the *LibGDX* based variant is implemented in a similar manner, utilizing *Groups*, such as *Tables*, to encapsulate other widgets.

Due to the implications from chapter 2.1, it is expected that the *LibGDX* based implementation will result in a smoother user interface, meaning more frames per second (fps). In general a desirable outcome would comprise the plain *Android View* based

implementation to reach "the magic number of 30 fps for smooth motion" [32]. This would make the need for an additional framework, such as *LibGDX*, obsolete. Consequently, regarding the model-view-controller paradigm, no additional interfaces would be required for communications between the data model of the device (e.g. current oven data) and the view/controller provided by the framework. Lastly, no further learning sessions for *Android* developers would be necessary.

In order to measure performance, three screens and various animations were implemented. Utilized animations comprise fading, rotating, scaling, translating and color transitions.

The following screens, alongside with their animations, were implemented with *Android Views*. Afterwards, another application with the same content was developed using the *LibGDX* framework.

The first screen is the splash screen with the *Bosch* logo and includes up to four animations in parallel (see figure 2.3).
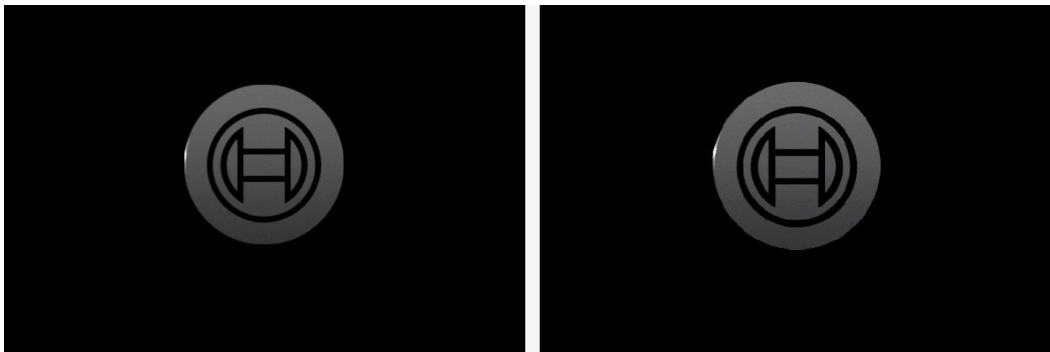
Figure 2.3: A screenshot of the *Android View* based (left) and *LibGDX* based (right) animated splash screen. This screen is assumed to generate the least workload within this performance analysis.

The second screen features two lists of clickable entries/buttons which can be scrolled simultaneously (see figure 2.4).

The third screen comprises a toggle animation between two options which entails up to 18 simultaneous animations (see figure 2.5).
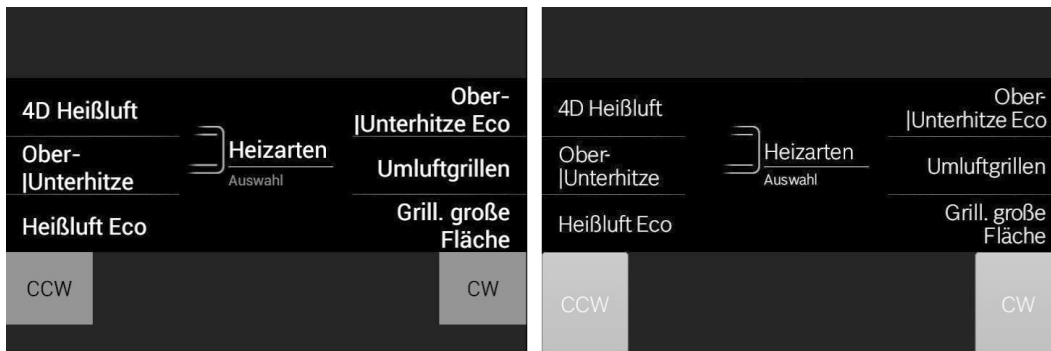
Figure 2.4: A screenshot of the *Android View* based (left) and *LibGDX* based (right) selection screen. The CW (clockwise) and CCW (counterclockwise) buttons simulate the respective swipe interaction along the ring of the oven's user interface (see figure 1.1). Such an interaction will cause a scroll animation of each of the two lists within this screen.

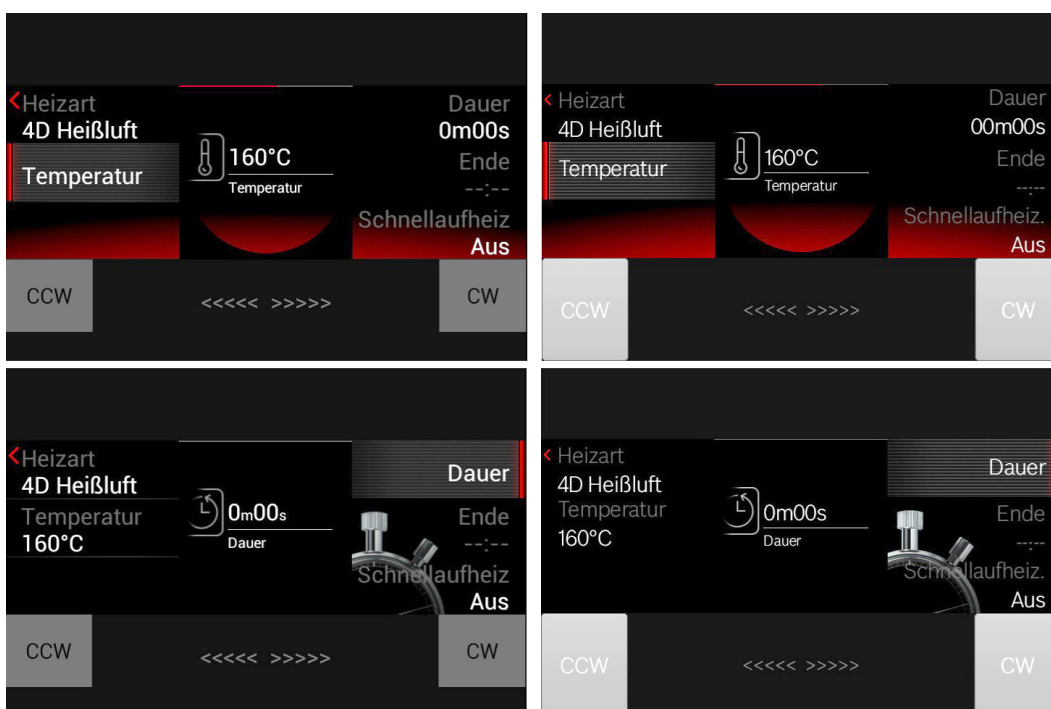**Figure 2.5:** Screens of the *Android View* based (left) and *LibGDX* based (right) *Heizart* settings. The toggle between the *Temperatur* (upper) and *Dauer* (lower) setting entails a total of 26 animations, 18 of which run in parallel. This screen is assumed to generate the most workload (in terms of animations) among the three screen which are under examination.

These screens with their respective animations were chosen as a test GUI in order to conduct the performance analysis in the scope of a realistic setting. The test GUI was implemented in regard of the user interface specification provided by BSH. Furthermore, an increasing workload in terms of animations was implemented by the three screens in order to provide potential correlations between workload and framerate.

2.3 Measurement

For each of the screens, the respective set of animations was tracked by recording the timestamp when the *onDraw* method (regarding *Android View*) or the *render* method (regarding *LibGDX*) was called. In doing so, the interval between two frames as well as the fps can be calculated as follows:

$$\Delta t = t_2 - t_1 \qquad (1)$$

$$fps = \frac{1000ms}{\Delta t} \qquad (2)$$

Considering *Android Views*, drawing is performed in regard of the *View* hierarchy, walking the tree breadth first in order. This default drawing order can be overridden, for instance when applying a Z value to a *View* (via *setZ(float)*). In order to override the *onDraw* function of the *Android View* based implementation, a custom layout class was implemented and applied to the top node of the layout.xml file of each of the three screens.

The *FrameInspector* class was introduced in both implementations with close to identical code. As a consequence, rendering will be equally influenced by the performance tracking process and a valid comparison can still be performed since both variants suffer from equal drawbacks in performance caused by the *FrameInspector*.
When an animation starts, the *FrameInspector* will be triggered, storing a timestamp in

the heap space each time the *onDraw* or *render* function is called. Code 2.1 shows the *FrameInspector* usage within the *render* function of the *LibGDX* implementation.

Code 2.1: The *FrameInspector* implementation within the *LibGDX render* function

```java
@Override
public void render(float delta) {
  ...
  if(frameInspector.doCount()){
    frameInspector.increment();
  }
}
```

The *increment* function of the *FrameInspector* class is shown in code 2.2.

Code 2.2: The *increment* function of the *FrameInspector*

```java
public void increment(){
  frame_count++;
  timeStamps.add(System.currentTimeMillis());
}
```

When the animation is finished, the *FrameInspector* will be notified and the intervals between the timestamps will be calculated and logged into a text file. The expensive procedure of writing the log file is executed after the animation has finished, therefore performance tracking, while the animation is running, is reduced to a minimum by merely gathering timestamps. The calculated intervals between the timestamps can be converted into fps as mentioned earlier.

2.4 Results

The following results were calculated as an average of 10 iterations of each animation. The results meet the previous expectations and clarify that the *LibGDX* based application exceeds the *Android View* based implementation in fps. The following diagrams (2.6,

14

2.7 and 2.8) depict the total amount of frames, as well as the derived fps as calculated via timestamps gathered in the *onDraw*/*render* functions.
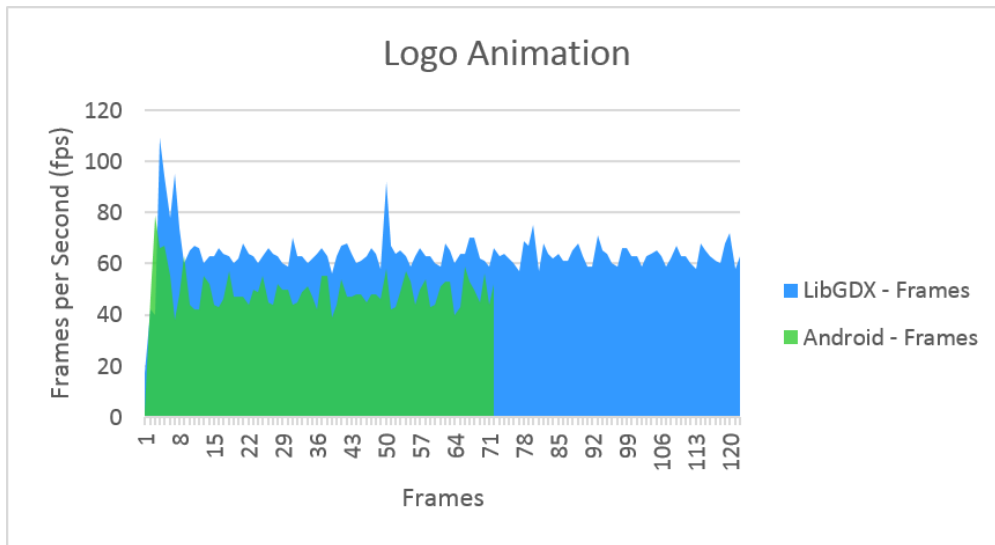

Figure 2.6: The results of the logo animation show significantly more fps of the *LibGDX* variant compared to the *Android View* based implementation. This also becomes apparent when comparing the amount of rendered frames throughout the animation.
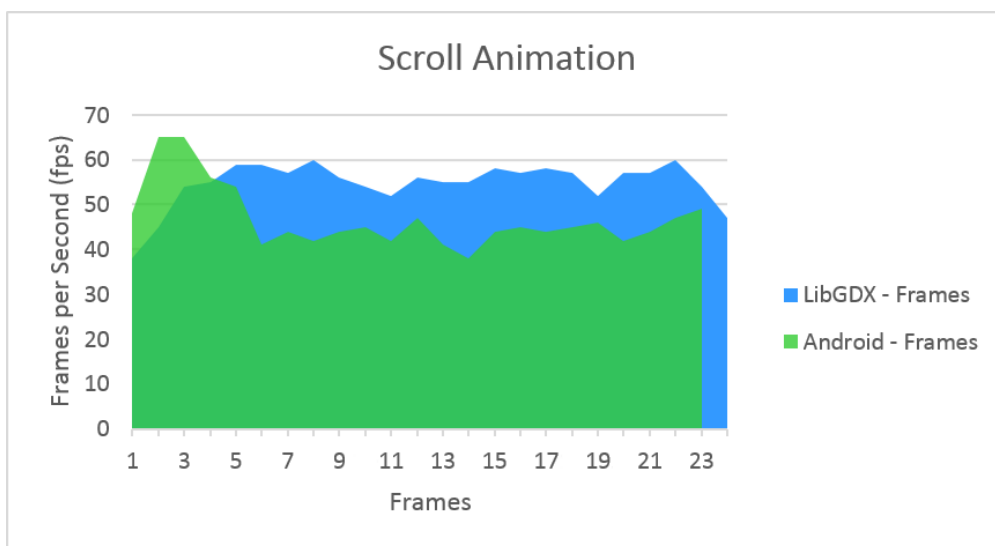

Figure 2.7: Throughout the scroll animation, the *Android View* based variant keeps up a constantly high framerate above 30fps.
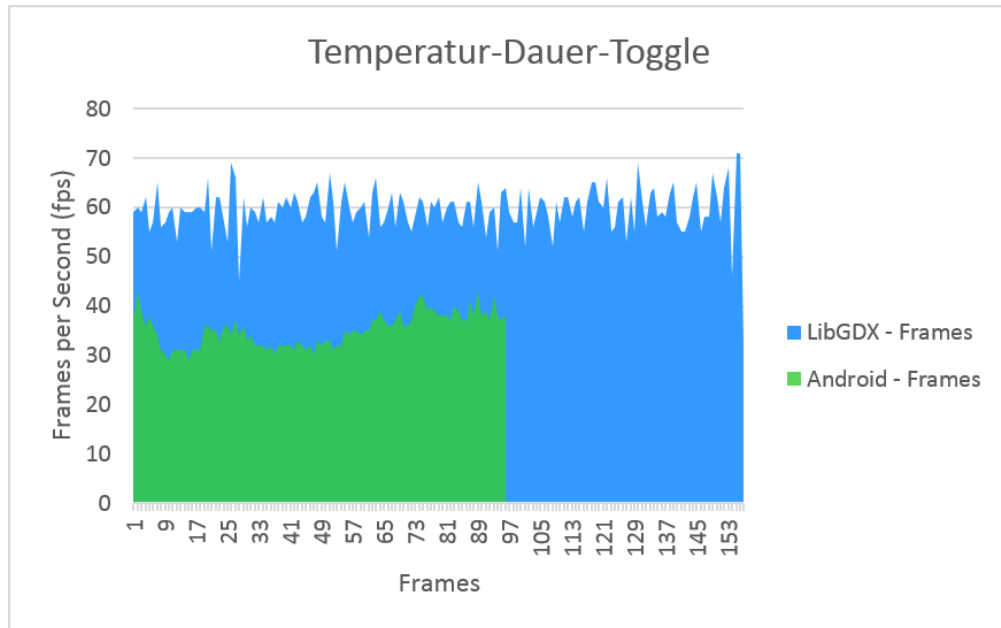
Figure 2.8: The measured data of the *Temperatur-Dauer*-toggle animation shows an even greater gap between the two implementations when compared to the results of the logo animation 2.6. The *Android View* based variant suffers from significant drops in framerate as the workload increases.

A notable implication can be drawn from the three diagrams: While the *Android View* based variant features an explicit drop in fps as the animation workload increases, the *LibGDX* based implementation appears to render even more complex animations (see diagram 2.8) as smoothly as rather simple scenarios (see diagram 2.6), with close to 60 fps. The *Android View* based implementation seems to suffer from a considerable loss in framerate as the amount of simultaneous animations increases. As a consequence, when further increasing the workload, the framerate would probably drop even more often below 30 fps as it already did while rendering the *Temperatur-Dauer*-toggle animation (see diagram 2.8).

Nonetheless, the *Android View* implementation only dropped to a minimum of 28 or 29 fps and thus still reached an average target framerate of well above 30 fps in each animation on the evaluation module (see chapter 1.2).

In conclusion, *Android* is clearly capable of rendering the tested scenarios smoothly on hardware with limited resources, such as the evaluation module introduced in chapter 1.2. Furthermore, the results clearly show that the *LibGDX* based implementation delivers a considerably higher framerate as opposed to an *Android View* based implementation.

3

Embedding Android

The goal of this chapter is to describe how to embed an application similar to a kiosk-mode on *Android* but to an even more thorough extent. In other words, the application should imitate a native system. Consequently, the application in focus should be the only accessible application to the user and should be running in foreground permanently. Redundant functionalities, enabled through both hardware and software modules, should be disabled.

Due to the active nature of the *Android Open Source Project* (AOSP), different versions may vary in terms of conventions, structure, modules etc. The main objective of this chapter is to collect all potential aspects that are relevant in the process of embedding an application alongside with the *Android* OS. These aspects will be described, a specific implementation to achieve the desired behavior will be given and potential alternatives will be compared.

3.1 Target Specification

Since *Android* comes from the mobile domain [35], it requires certain modifications to be suitable for being embedded into an oven. Before beginning with the modifications, it is necessary to specify the desired behavior.

For a start, the application in focus should act as the **launcher application** on the *Android* OS. The application should start immediately after booting the device (see chapter 3.4.1).

Trimming redundant packages will not only improve performance of the device but

also contribute to the stability of the embedded system (see chapter 3.4.2).

Furthermore, the **button functionality** on Android (such as the home button, volume buttons, menu/recent apps button and back button) should be disabled (see chapter 3.4.3). Navigation through the GUI should solely be conducted via itself, meaning the functionality of for instance the back button will be delegated to the respective widget within the GUI.

Besides, **maintaining the focus** of the application is crucial for such an embedded scenario (see chapter 3.4.4). Neither should the application go into background nor should any other application obtain focus. Uncaught exceptions for instance pose a threat to the continuity of the application in focus and have to be handled.

Additionally, a **power management** plan should be developed by looking into the standby mode as well as the regulation for the display brightness. Screen dimming for instance is usually engaged in case the user interface remains in an idle state for a specific duration (see chapter 3.4.5).

Finally, alterations of the *Android* OS should be conducted to meet the **corporate design** of BSH (see chapter 3.4.6). This comprises for instance customizations of the boot animation (and respective sound) when powering up the device.

3.2 Kiosk Software

In order to consider all potential aspects relevant to the embedding process, it seems worth considering (commercial) kiosk software. Such software is often used for exhibitions, studies, etc. where the access of a device is restricted to a certain website or application. In contrast to the embedding process that is in focus of this thesis, such kiosk software is often applied only temporarily to a device and primarily blocks several features for a certain period of time rather than making persistent changes to the system. However, the aspects that are considered in such software (rather than their implementation) should be taken into account to provide an embedding process that is as complete as possible.

Among the regarded kiosk software was *KioWare* [25] and *SureLock* [29]. These products enable restrictions to certain applications so that only specified applications are accessible. *SureLock* for instance features a custom home screen which provides the exhibited applications. Furthermore, *SureLock* enables the designation of a launcher application that executes on startup. A permanent setting of these features is illustrated in chapters 3.4.1 and 3.4.2.

Furthermore, *SureLock* can hide the virtual buttons on *Android* 3.x and higher. Chapter 3.4.3 covers button handling of an *Android* device. This includes virtual buttons as well as physical buttons.

The examined kiosk software is also able to block the system settings and lock specific features, for instance sound, bluetooth etc. Such restrictions can be achieved permanently by removing the respective applications/packages such as the status bar, bluetooth and the settings application (see chapter 3.4.2).

The fact that such kiosk software is able to achieve these objectives proofs that a potential oven application could implement these features, as well.

3.3 Android Architecture

Android is an open-source project that was released in October 2008 [11]. It is an operating system, initially designed as a mobile software platform [6], that features a *Linux* kernel-based architecture. The *Android* architecture, as depicted in Figure 3.1, consists of four main layers and five sections [13]:

Applications

The top layer is composed of the default/initial applications such as the home launcher application or the contacts application that come with a smartphone. Consequently, any application that will be installed goes to this layer.

Application Framework

The second layer is the Application Framework which provides APIs to be used by the Application layer. This comprises for instance the View System, which provides a framework to create GUIs.

Libraries

The Libraries layer enables applications to access core features, e.g. a custom system C library (libc) for embedded Linux-based devices, a SQLite database or the *OpenGL ES* library.

Android Runtime

The Android Runtime features an adaption of a *Java* virtual machine (VM) named *Dalvik* which is specifically designed for memory- and CPU-constrained devices. The core libraries are designed to interact directly with an instance of the *DalvikVM*.

Linux Kernel

The Linux Kernel is the base layer of the *Android* architecture. It contains all hardware drivers, handles power and memory management as well as resource access.

Figure 3.1: The *Android* architecture is composed of four main layers and five sections [40].

A more system oriented view with regards to the AOSP is given in the depiction of the *Android* architecture in figure 3.2.
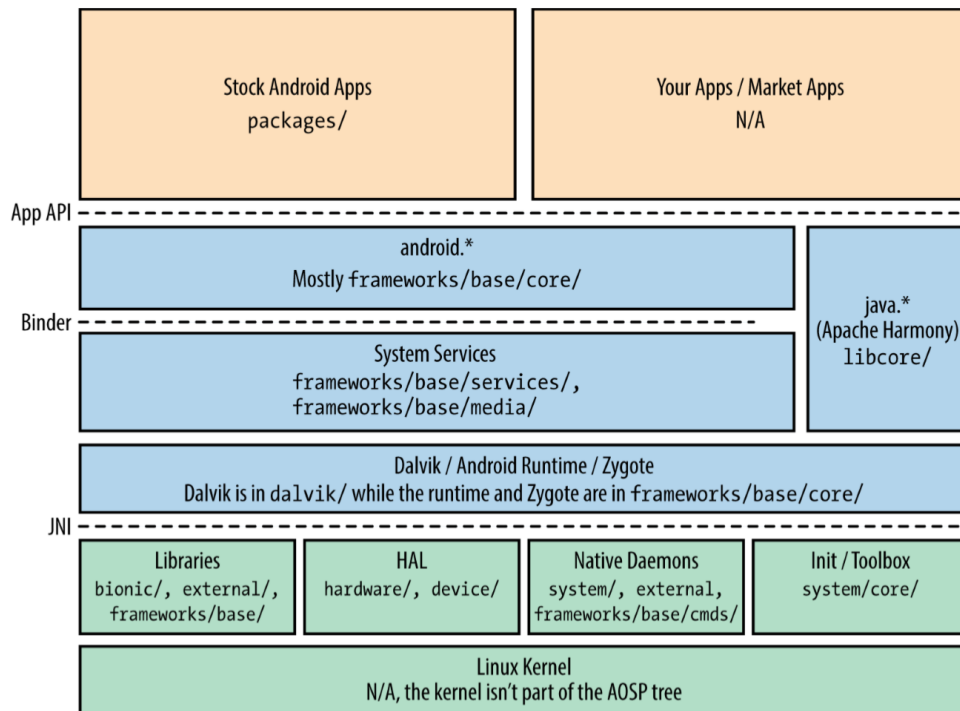
Figure 3.2: The *Android* architecture with respect to the AOSP. The directories indicate the location of the respective component within the AOSP [44].

In order to meet the target specification as defined in chapter 3.1, the init component (see figure 3.2) will be modified to customize the boot process and set various properties to disable/enable certain features. Another important component for the embedding process is the hardware abstraction layer (HAL). It defines APIs for hardware components, such as Bluetooth, NFC, WLAN, camera, audio etc. By modifying the HAL modules, the functionality of certain hardware components can be disabled. Additionally, the respective drivers and services can be removed for a lightweight *Android* OS.

3.4 Embedding Strategies

This chapter describes a conceptual approach on how to embed an application alongside with the *Android* OS into an oven by BSH. Necessary steps, as well as alternatives, will be explained and accompanied by supplementary code snippets where applicable. Most adaptations will be undertaken either in the application itself (application layer) or directly within components of the AOSP. Regarding the latter, modifications can often be made by editing the */system/build.prop* file. The *build.prop* file is a system file that contains properties such as flags and values which are requested by various modules during the device's boot process [44]. Adjustments to the *build.prop* file require root privileges. Besides, due to the open source nature of the *Android* OS, a custom AOSP can be compiled.

Several required modifications can be done by editing properties within the *build.prop* or the *init.rc* file. This will result in global changes throughout all devices. Such properties can be overridden by a specific device by editing its *device.mk* file within the */devices/<vendor>/<product-name>/* folder. This principle of global changes as opposed to device specific modifications of the AOSP can be applied to various aspects, such as the default set of pre-installed applications or support for specific hardware.

3.4.1 Launcher Application

The application at hand, meaning the application to control the oven, is supposed to be the default application running on the *Android* OS. Adjustments withing the application itself are suffice in order for the *Activity* to start immediately after booting the device. For this purpose, the *AndroidManifest.xml* file can be augmented as illustrated in code 3.1.

The *RECEIVE_BOOT_COMPLETED* permission enables the application to listen for the *BOOT_COMPLETED* action that will be received by the BroadcastReceiver implementation as shown in code 3.2.

Code 3.1: Necessary modifications within the *AndroidManifest.xml* to launch an application directly after the boot process

```
<manifest ...>

  <uses-permission
     android:name="android.permission.RECEIVE_BOOT_COMPLETED" />

  <application ...>
    <receiver android:name=".BootReceiver">
      <intent-filter>
      <action android:name="android.intent.action.BOOT_COMPLETED"/>
      </intent-filter>
    </receiver>
  </application>
</manifest>
```

Code 3.2: Once the *BOOT_COMPLETED* action was invoked this *onReceive* function starts the desired *Activity*

```
public class BootReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    Intent bsh_start_activity_intent = new Intent(context,
        BSH_Start_Activity.class);
    context.startActivity(bsh_start_activity_intent);
  }
}
```

Once the *onReceive* function received the *BOOT_COMPLETED* action, the initial *Activity* of the application will be started, in this case named *BSH_Start_Activity*.

An alternative to configure a launcher application is to simply augment the intent-filters of the *BSH_Start_Activity* in the *AndroidManifest.xml* as illustrated in code 3.3.

Code 3.3: Defining intent-filters for a launcher application

```
<intent-filter>
  <category android:name="android.intent.category.LAUNCHER" />
  <category android:name="android.intent.category.HOME" />
</intent-filter>
```

In doing so, the *BSH_Start_Activity* will launch directly after booting the device and will be handled as a home application, meaning it can be started when triggering the home button [17]. Note that the *LAUNCHER* category does not set this *Activity* to be started after the booting process but to be the initial *Activity* that is launched when the application is started. Category *HOME* designates this *Activity* to be started after the booting process as well as in the event of the home button.

The first method might seem more cumbersome, simply due to the amount of code that is required when compared to the second implementation. However, utilizing a *BroadcastReceiver* which determines the *Activity* that should be started is more flexible. For instance in the event of an abrupt reboot or the like, the *BootReceiver* can resume the system's state by invoking the last *Activity* that was active prior to the reboot. Nonetheless, this variant also results in a higher delay between the finished boot process and the application to start.

These adaptations are undertaken within the application itself and thus, in the context of the *Android* architecture (chapter 3.3), can be considered as part of the applications layer.

3.4.2 Trimming Packages

Removing redundant packages from the custom AOSP will boost performance and stabilize the isolation of the application in focus. Packages that are designated to be installed initially are specified in several make files located in the */target/product* folder. As described by Karim Yaghmour [44], alterations within the *base.mk* will have crucial impacts on the system and are more likely to generate a broken

AOSP since there are no dependency checks considering packages. The *core.mk* and *generic.mk*/*generic_no_telephony.mk* files hold more self-contained packages and thus are safer to edit. The *generic_no_telephony.mk* holds, among others, the entries that are mentioned in code 3.4.

Code 3.4: An excerpt of the stock packages of a *generic_no_telephony.mk* file

```
PRODUCT_PACKAGES := \
DeskClock \
Bluetooth \
Calculator \
Calendar \
Email \
Exchange2 \
Gallery2 \
Launcher2 \
Music \
MusicFX \
Phone \
Settings \
```

When removing for instance the *Launcher2* entry, the device will be missing the default home screen application.

By merely removing an entry, the respective application will still be part of the AOSP, located in */packages/apps*. When not needed, it can be deleted from that location.

Consequently, adding an application to the default packages list is the reversed process. Either the *core.mk* or the *generic.mk*/*generic_no_telephony.mk* file needs to add the desired application to *PRODUCT_PACKAGES* and the application has to be placed into the */packages/apps* folder. In doing so, the application will be installed as a stock application on all potential devices which the custom *Android* version will be mounted on.

Besides this global approach, the application can be installed in respect to a particular device. For this example, the application will be labeled Oven-App. The device specific

configuration is located in the */devices/<vendor>/<product-name>/* folder. In order to limit the installation of the application as a stock package to specific devices, the application has to be placed inside this folder. Further on, an *Android.mk* file has to be created within the application folder with the content in code 3.5.

Code 3.5: Settings for the *Android.mk* file for building the *Oven-App* application

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES := $(call all-java-files-under, src)
LOCAL_PACKAGE_NAME := Oven-App
include $(BUILD_PACKAGE)
```

Finally, the application has to be added to *PRODUCT_PACKAGES* within the *full_product-name.mk* file [44].

3.4.3 Button Handling

Since navigation on the device should solely be conducted via the application's user interface and there is no need for other features such as the home application or a recent apps list, the regular *Android* buttons (home button, volume buttons, menu/recent apps button and back button) will be disabled. Ideally, both the buttons' default functionality and/or the firing of the button event in the first place should be prevented.

Preventing button firing

Firstly, the **navigation bar** (containing the back button, home button and menu/recent apps button in the form of virtual buttons) can be hidden through application code when using *Android* version 4.0 and higher [15]. This is achieved with an implementation as indicated in code 3.6. It is notable that this method will not hide the navigation bar throughout the entire system but merely within the scope of an *Activity* that implements code 3.6.

Code 3.6: Placing this code in the *onCreate* function of an *Activity* hides the navigation bar within the *Activity*

```
View decorView = getWindow().getDecorView();
int uiOptions = View.SYSTEM_UI_FLAG_HIDE_NAVIGATION
            | View.SYSTEM_UI_FLAG_FULLSCREEN;
decorView.setSystemUiVisibility(uiOptions);
```


Figure 3.3: The navigation bar with virtual buttons of an *Android* device [15].

A more thorough approach requires root privileges or a custom built of the AOSP. Rooting an *Android* device is another term for the process of gaining super user privileges [3]. The line of code 3.7 has to be added into the */system/build.prop* file to disable the navigation bar throughout the entire system [41]:

Code 3.7: Setting this property in the *build.prop* file hides the navigation bar

```
qemu.hw.mainkeys=1
```

This will hide the navigation bar on all devices. To target a specific device, the navigation bar can be hidden by overriding the value in the */devices/<vendor>/<product-name>/device.mk* file as shown in code 3.8.

Code 3.8: Hiding the navigation bar for specific devices

```
PRODUCT_PROPERTY_OVERRIDES += \
qemu.hw.mainkeys=1
```

Furthermore, key layout files (*.kl* files), which map *Linux* key codes to *Android* key codes [18], can be edited within the */devices/<vendor>/<product-name>/* folder. Code 3.9 shows the entries for the virtual buttons (home button, menu button and back button) and the physical buttons (volume control and power).

Code 3.9: Key code mapping that assigns keys of *Android* devices to key codes

```
key 102   HOME      VIRTUAL
key 114   VOLUME_DOWN  WAKE
key 115   VOLUME_UP  WAKE
key 116   POWER      WAKE
key 139   MENU      VIRTUAL
key 158   BACK      VIRTUAL
```

The *WAKE* keyword implies that the device will wake up if it is asleep in case the respective button was invoked. By commenting out these mappings, the buttons will be detached from any functionality.

Disabling button functionality

The following describes strategies on how to consume button events in case they have been fired. Although the possibility to trigger respective buttons is intended to be disabled in the first place, these approaches present a further level of immunity against unwanted behavior.

Handling the **back button** is common practice in *Android* development. It can be easily disabled by overriding the respective function in each *Activity* as shown in code 3.10.

Code 3.10: Overriding the back button behavior within an *Activity*

```
@Override
public void onBackPressed() {
   //do nothing
}
```

The **menu button** and **volume buttons** can be deactivated in a similar manner as the back button, by overriding the functionality in each Activity as illustrated in code 3.11.

Code 3.11: Overriding the menu and volume buttons within an *Activity*

```java
@Override
public boolean dispatchKeyEvent(KeyEvent event) {
  if(event.getKeyCode()==KeyEvent.KEYCODE_VOLUME_DOWN ||
      event.getKeyCode()==KeyEvent.KEYCODE_VOLUME_UP){
    return true;
  }else if(event==KeyEvent.KEYCODE_MENU){
    return true;
  }else{
    return super.dispatchKeyEvent(event);
  }
}
```

Long pressing the **power button** prompts a system dialog which provides options such as to restart the device or to turn it off. A simple way to prevent this dialog to be displayed is to immediately close it when it is about to show up [39]. Such behavior can be achieved by augmenting an *Activity* with code 3.12.

Code 3.12: Immediately closes any system dialog that is about to show up from within an *Activity*

```java
@Override
public void onWindowFocusChanged(boolean hasFocus) {
  super.onWindowFocusChanged(hasFocus);
  if(!hasFocus) {
    Intent closeSystemDialog = new
        Intent(Intent.ACTION_CLOSE_SYSTEM_DIALOGS);
    sendBroadcast(closeSystemDialog);
  }
}
```

Overriding the *onWindowFocusChanged* function also directly addresses the recent apps dialog since this dialog also qualifies as a system dialog. Depending on the *Android* version and the device, the recent apps dialog can be opened via a long press on the home button or a dedicated recent apps button. Consequently, by listening to a focus

change, the recent apps dialog will be closed no matter if it was opened via a long press on the home button or a recent apps button.

Further long press button events, such as long pressing the back button, can be handled by overriding the *onKeyLongPress(int keyCode, KeyEvent event)* function.

When configuring the application as a home application, as described in chapter 3.4.1, the launcher *Activity* (previously labeled *BSH_Start_Activity*) would be launched when the home button is pressed. However, when the user has navigated to another *Activity* and the home button is triggered, the launcher *Activity* will be invoked. One feasible approach to keep the current state (consisting of displayed *Activity*, selected values etc.) in case the home button was pressed, is to persist the state (e.g. via *SharedPreferences*) and load it when the start *Activity* is called.

Overriding the home button in *Android* is considered bad practice due to security and user experience reasons. However, these arguments do not apply in an embedded scenario and there are several workarounds on the application layer to consume the home button event. Some of these approaches, though, have been rendered impracticable as the *Android* versions advanced. This simple consumption of the home button event (see code 3.13), applied to each *Activity* in which to prevent the default home button functionality, has been disabled with *Android* version 4.0.

Code 3.13: Overriding the home button within an *Activity* below *Android* version 4.0

```java
@Override
public void onAttachedToWindow(){
  this.getWindow().setType(WindowManager.LayoutParams.TYPE_KEYGUARD);
  super.onAttachedToWindow();
}
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
  if (keyCode == KeyEvent.KEYCODE_HOME) {
    return true;
  }
  return super.onKeyDown(keyCode, event);
}
```

Due to the fact that the home button handling has been changed with the different *Android* versions in the past, this might also be the case with future *Android* versions as well. Since there is no clean way of handling the home button event in *Android* versions 4.0 and above (at least within the application layer), modification of the key layout files will have to be sufficient.

3.4.4 Maintaining the Focus

Status Bar

In order to keep the application in the foreground, the status bar has to be disabled or rendered inaccessible. The status bar is an entry point for the settings and potentially any other application.

Figure 3.4: The status bar of an *Android* device [16].

Hiding the status bar can be achieved by creating a custom theme in the *themes.xml* file (see code 3.14) and applying it as the theme to the application in the *AndroidManifest.xml* file. This approach, however, might not be a definitive solution since a hidden status bar, depending on the *Android* version, can be fetched (or not) by swiping down from the top edge of the screen. Pulling down the status bar can be prevented by covering the status bar with a view that consumes touch events.

Code 3.14: A custom theme that hides the *Android* status bar

```
<resources>
   <style name="customTheme"
      parent="@android:style/android:Theme.Holo.Light">
      <item name="android:windowFullscreen">true</item>
      <item name="android:windowContentOverlay">@null</item>
   </style>
</resources>
```

The most exhaustive approach, however, would be to remove the status bar application completely (see chapter 3.4.2).

Uncaught Exceptions

A further potential way for the application to be left/closed is through an uncaught exception. To catch such exceptions, each *Activity* should set an *UncaughtExceptionHandler* as shown in code 3.15.

Code 3.15: (Re)Starting the *BSH_Start_Activity* by catching uncaught exceptions

```
Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  final Context ctx = this.getApplicationContext();

  Thread.setDefaultUncaughtExceptionHandler(new
     Thread.UncaughtExceptionHandler(){
    @Override
    public void uncaughtException(Thread thread, Throwable ex) {
      Intent bsh_start_activity_intent = new Intent(context,
         BSH_Start_Activity.class);
      ctx.startActivity(bsh_start_activity_intent);
      System.exit(2);
    }
  });
}
```

In this case, the *BSH_Start_Activity* is invoked after the exception was caught. However, it is also possible for each *Activity* to restart a specific *Activity*, for instance itself, in case they crash. Further on, the *Intent* to start the new *Activity* can be augmented with extra information about the error and currently set values. In doing so, a message about the error can be displayed and previously set values can be restored on restart.

3.4.5 Power Management

Conserving power might not be as crucial to (stationary) home appliances as to mobile devices that run on battery, but an optimized power consumption plan, with respect to responsiveness, is still very desirable since an oven is potentially turned on permanently. Due to the portability of conventional *Android* powered devices, the AOSP alongside with a respective *Linux* kernel implementation already feature certain power conservation strategies. For example the *Linux* kernel adaptations which *Android* is built upon utilize wake locks as a more thorough approach on power management compared to a non-*Android* targeted *Linux* built [13]. There are different types of wake locks that can maintain for instance CPU or screen occupation.

Aside from the power management that already comes with the *Android* OS, a specific scenario as embedding *Android* into an oven might leave room for unique power management strategies. One way to conserve power is to manage **screen dimming**. The screen might be dimmed when the device is in an idle state or during a phase when it is unlikely for the user to interact with the display. There are several parameters that can be edited for custom screen dimming management. For one thing, the minimum and default brightness level can be set by editing properties, for instance within the */system/build.prop* file as shown in code 3.16.

Code 3.16: The screen brightness settings within the *build.prop* file

```
ro.lcd_min_brightness=5
ro.lcd_brightness=160
```

The brightness values are in the range between 0 and 255.

In the *Application* layer, the actual dimming value can be set for each *Activity* individually as illustrated in code 3.17.

Code 3.17: Controlling the screen brightness within an *Activity*

```
WindowManager.LayoutParams layoutParams =
    getWindow().getAttributes();
layoutParams.screenBrightness = 0.2; //Value between 0.0 and 1.0
getWindow().setAttributes(LayoutParams);
```

Besides screen dimming the device can go into **standby mode** after a certain period of time, thus turning off the display. This can be prevented programmatically within an *Activity* via code 3.18.

Code 3.18: Preventing the screen from dimming within an *Activity*

```
getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
```

Alternatively, the attribute in code 3.19 can be added to the root layout in the layout *.xml* file of the respective Activity.

Code 3.19: Preventing the screen from dimming via a layout attribute

```
android:keepScreenOn="true"
```

Considering timing, the default screen dimming timeout is managed by the OS but can be altered for each application. However, changing the timeout throughout the entire oven application might not be the desired solution. The desired timeout might change depending on the current state of the application, for instance setting a shorter timeout when the oven is preheating. Screen dimming can be regulated manually via the *WindowManager* with full control over brightness and timing when using custom timers and brightness values in regard of specific *Activities* and application states.

In terms of *Android*'s sleep/stand-by mode, the actual power consumption highly depends on the specific services and applications which might run in background. For instance a service might or might not be operating while the system appears to be sleeping.

3.4.6 Corporate Design

Since the application directly starts after booting the device, the only runtime at which the application is not in focus is during boot time and shutdown.

During the boot process, the screen will potentially display three different stages [44]. The **kernel boot screen** is the first stage during the visible boot process in which the kernel might display a static image. However, an *Android* device will usually not display this screen. Afterwards, a static **init boot logo** (either text string or image) will be displayed on the screen. Naturally, to ignore this init boot logo phase, an empty string can be assigned to it. The string can be edited within the *console_init_action()* function within the *system/core/init/init.c* file. For an image to show during this stage, the screen dimensions in pixels must be known and a proper sized image has to be converted into the *.rle* type, titled *initlogo* and placed into the root directory of the *boot.img* image [44]. Finally, the AOSP has to be rebuilt.

After the init boot logo, the **boot animation** will be invoked. The boot and shutdown animations can be placed in uncompressed *bootanimation.zip* and *shutdownanimation.zip* files within the *system/media* or *data/local* folder [44]. Code 3.20 is a configuration that decreases the booting duration and removes the respective animations (and sounds). It has to be set in the *system/build.prop* file.

Code 3.20: Setting within the *build.prop* file that disables the boot animation and thus increases the boot process speed

```
debug.sf.nobootanimation=1
```

To customize the boot animation, the content of the *bootanimation.zip* archive needs to be edited. The content of the *bootanimation.zip* depends on the *Android* version and includes a description file, for example a *desc.txt* or *boot_animation.xml* file. The *desc.txt* for instance describes the boot animation as illustrated in code 3.21 [44]. The actual images are located in *part0*, *part1*, etc. folders within the *bootanimation.zip* archive and contain incrementally named *.png* images.

Code 3.21: The schematic for a boot animation as described in *desc.txt*

```
<width> <height> <framerate>
p <loop> <pause> <folder0>
p <loop> <pause> <folder1>
...
```

An actual implementation of the *desc.txt* is shown in code 3.22.

Code 3.22: A sample boot animation description

```
480 800 30
p 2 10 part0
p 0 0 part1
```

The *p* stands for part and introduces a new sub-animation. The *loop* number defines the amount of iterations the sub-animation will play. When set to 0, the sub-animation will play indefinitely (until boot is completed). The *pause* field sets the pause duration in number of frames to be skipped until the next sub-animation will start [34].

Additionally, the *bootanimation.zip* includes a *boot.mp3* or *boot.ogg* audio file to be played during the boot process.

The *shutdownanimation.zip* can be edited in a similar manner.

3.5 Implications

This chapter identified several aspects of *Android* that require modification to be eligible for being embedded into an oven. However, it is likely that not all potentially relevant aspects were covered since particular behavior is usually identified through a precise requirement analysis that is conducted with regard to particular hardware (providing a specific set of capabilities) and a desired interaction model. The purpose of this theses, however, is to generally assess the applicability of *Android* for such an embedding project. Consequently, this chapter handled the fundamental aspects that are relevant to the embedding process with an oven in mind.

Some of the introduced embedding implementations in this chapter might seem redundant, such as overriding button handlers when their linkage to the respective key code is already detached. The purpose of these duplicate approaches is to provide multiple ways of achieving certain objectives.

Implementing an alternative solution might be useful due to a couple of reasons. One solution might be simply more straightforward and thus easier and faster to achieve than another. Furthermore, rather than removing a low-level module which might be useful in a later version of the software, a high-level alternative implementation might be just as effective.

A reason for redundant handling of a certain aspect might be thoroughness. Removing the status bar package from the AOSP when the status bar is already hidden (e.g. via a high-level modification) seems unnecessary but reduces the size of the AOSP and thus increases performance and stability.

As demonstrated in this chapter, it is a feasible task to tailor the *Android* OS into an appropriate system to be embedded into an oven. This is due to the open source nature of the AOSP. Further on, even application layer modifications can have a rather extensive working range and be of considerable value. The provided code snippets in this chapter illustrate that it is a manageable amount of implementation work to achieve the desired behavior.

A potential solution was found for each particular aspect which was introduced in this chapter that requires modification. Although the general features of *Android* that are relevant for embedding were handled, the future might bring new challenges, either from within the AOSP or by augmenting the requirements of the oven system. Nonetheless, it is very likely that such potential upcoming challenges can be handled when working with the AOSP.

4

Hardware Communication

This chapter will focus on the conceptual approach of adding support for custom hardware to the AOSP. The goal is to investigate potential inter-process communication (IPC) mechanisms that are eligible for establishing a communication channel between the user interface application of the oven and the oven hardware module/driver.

4.1 Overview

Adding support for new hardware in *Android* requires respective implementations throughout various layers.

There are several ways to create an IPC channel between the application framework and the *Linux* kernel module/driver responsible for the hardware in focus. In the case of *Android*, kernel space describes the *Linux* kernel while user space represents all libraries, processes etc. that are built on top of the kernel.

This chapter will examine the following IPC methods that are available on *Linux*:
System call (see chapter 4.2.1) is the standard way to make kernel space services available to user space processes.
Input/output control (**ioctl**) (see chapter 4.2.2) is a specialized system call to facilitate communication with specific device drivers.
sysfs (see chapter 4.2.3) is a virtual file system mechanism for exporting and accessing kernel objects, such as device files, which represent actual devices in *Linux*.

Furthermore, **netlink sockets** (see chapter 4.2.4) provide a full duplex communication link between kernel space and user space with a socket-type API.

The way that hardware is usually integrated into the AOSP is the use of Binder, system services, and HAL (as described in chapter 4.2.5) and comprises the following layers in order to access hardware functionality via the application framework API:

The *Linux* **kernel** must feature the desired hardware driver or hardware module that interfaces with the hardware.

Within the AOSP, the **hardware abstraction layer (HAL)** is a standard interface that exposes hardware functions to the *Android* system. There are no restrictions considering the interface and interactions between the hardware driver and the HAL implementation.

System services are modules that run in background and access the HAL interface. *System Server* is the main component in the system services and is responsible for starting other services.

Finally, the **Binder IPC** mechanism allows crossing process boundaries and thus enables the **application framework** to reach into system services.

Figure 4.1 depicts a high level view of the *Android* architecture in the scope of hardware functionality.

Following the examination of each IPC method, a summary (chapter 4.3) is given that provides a comparison between these diverse mechanisms by considering several aspects that are potentially relevant for the communication with an oven hardware module/driver.

Figure 4.1: A high level view of the *Android* system architecture in respect of hardware support [13].

4.2 IPC Mechanisms

The oven in focus of this thesis utilizes DBus2 as a proprietary serial data transmitter. DBus2 features several similarities to a controller area network (CAN) bus. For the purpose of this thesis a DBus2 (or similar) driver/module is assumed as given. The principal focus of this chapter is the examination of potential IPC mechanisms between the application layer and the kernel driver.

Each introduced IPC mechanism in this chapter is accompanied with code snippets that are intended to give a basic overview of their usage. Illustrating IPC mechanisms with the aid of particular code examples improves the understanding of their inner workings, such as dependencies and relevant components, and gives a rough estimate of the required implementation effort.

4.2.1 System Call

System call is the standard mechanism to enable communication between user space and kernel space (see figure 4.2). Practically any other IPC mechanism in *Linux*, such as ioctl (see chapter 4.2.2), sysfs (see chapter 4.2.3) or netlink sockets (see chapter 4.2.4) is ultimately based on system call.

System calls can be utilized to manage processes, files and devices via operations such as read, write etc. For identification purposes, each system call has a unique number [27]. There are about 300 system calls in *Linux* [28]. Acting as a layer between user space and hardware, system calls feature three principal aspects: For one thing, system calls provide abstraction in a way that when for instance interacting with files from another device, the actual low-level communication with the medium that stores the files (e.g. CD-ROM, USB flash drive etc.) is hidden from the user. Furthermore, system calls incorporate a mechanism to manage access permissions of system resources, thus ensuring security and stability. Finally, system calls as a common layer between user space and kernel space enable stable multitasking and virtual memory management [27].
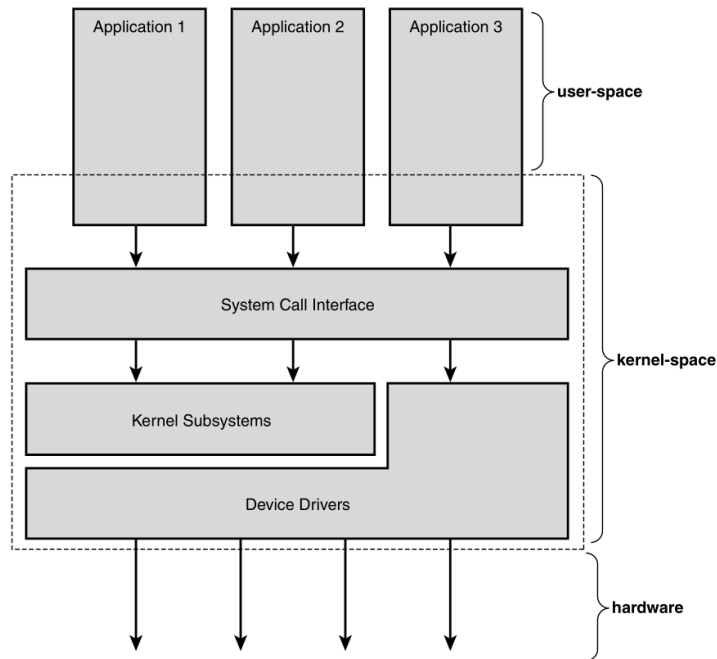
Figure 4.2: A schematic overview of the relationships between applications in user space, system
calls and the *Linux* kernel [27].

Considering *Linux*, a system call is not called directly from user space. It is invoked
indirectly by writing the respective system call number and desired arguments into desig-
nated registers of the CPU and causing an interrupt. An exception handler (a function
within the kernel) will handle this interrupt by reading the registers, checking for a valid
system call number within the system call table and invokes the appropriate kernel
function with the passed arguments. The system call number should be registered in the
system call table (see table 4.1) with a file and entry point of the target implementation
[27].

Name	eax	ebx	ecx	edx	esi	edi	Implementation
sys_restart_syscall	0x00	-	-	-	-	-	kernel/signal.c
sys_exit	0x01	int error_code	-	-	-	-	kernel/exit.c
sys_fork	0x02	struct pt_regs *	-	-	-	-	arch/alpha/kernel/entry.S
sys_read	0x03	unsigned int fd	char __user *buf	size_t count	-	-	fs/read_write.c
sys_write	0x04	unsigned int fd	const char __user *buf	size_t count	-	-	fs/read_write.c
sys_open	0x05	const char __user*filename	int flags	int mode	-	-	fs/open.c
sys_close	0x06	unsigned int fd	-	-	-	-	fs/open.c
...	...	...	...	...	...	...	...

Table 4.1: The top of a system call table [8]. The *ebx* to *edi* registers hold the first five arguments
of a system call. The *eax* register holds the system call number.

Adding a System Call

The *Linux Kernel Archives* [26] were utilized as a base for the following conceptual integration description.

In general, it is discouraged to create a multi-purpose system call by multiplexing system calls in *Linux*. A system call should serve exactly one purpose [27]. This does not mean a system call should be exclusive to certain modules but its dedicated purpose should be fixed.

In order to add a custom system call, the *arch/arm/include/asm/unistd.h* file has to be edited by including the new system call number (in this example with a new system call named *test_call*), as shown in code 4.1.

Code 4.1: Adding a new system call stub

```
...
#define __NR_SYSCALL_BASE 0
...
#define __NR_restart_syscall (__NR_SYSCALL_BASE+ 0)
#define __NR_exit           (__NR_SYSCALL_BASE+ 1)
#define __NR_fork           (__NR_SYSCALL_BASE+ 2)
...
#define __NR_rt_tgsigqueueinfo (__NR_SYSCALL_BASE+363)
#define __NR_perf_event_open (__NR_SYSCALL_BASE+364)
#define __NR_test_call    (__NR_SYSCALL_BASE+365) /* added */
```

Further on, the */arch/i386/kernel/entry.S* adds *test_call* to the system call table as illustrated in code 4.2.

Code 4.2: Adding a new system call table entry

```
ENTRY(sys_call_table)
...
.long SYMBOL_NAME(sys_rt_tgsigqueueinfo) /* 363 */
.long SYMBOL_NAME(sys_perf_event_open)  /* 364 */
.long SYMBOL_NAME(sys_test_call)   /* 365 */
```

The next step is the system call implementation with proper linkage which has to be added to the *Linux* kernel. A *test_call.h* file should be added either to the */include/linux* folder for an architecture agnostic system call or into the */include/asm* for an architecture dependent implementation.

The *test_call.c* file should be placed in the relevant folder to maintain consistency: */fs* for a file system relevant system call, */ipc* for a process synchronization relevant system call, */sound* for an audio relevant implementation etc. Finally the makefile of the chosen folder needs to include the added file.

The actual implementation should feature a function that matches the pattern in code 4.3.

Code 4.3: Adding a new system call stub in the *test_call.c* file

```
asmlinkage int sys_test_callN(int arg1, char* arg2) {}
```

Lastly, code 4.4 within the header file creates the binding to the system call with its respective function.

Code 4.4: Adding a new system call stub in the *test_call.h* file

```
_syscall2(int, test_call, int, arg1, char*, arg2);
```

The number of arguments are determined by *N* in *_syscallN*. The first parameter is the return type, followed by the function name. The next parameters are the arguments in the form of argument type and argument name.

According to Robert Love [27], there are several reasons not to implement a new system call but to fall back on alternatives such as ioctl (see chapter 4.2.2) or sysfs (see chapter 4.2.3). For one thing, a system call requires a unique number for identification purposes which should be an official attribution. Once a system call is implemented and part of the kernel tree, it cannot be changed without (immense) repercussions. Furthermore, each architecture potentially requires an independent registration of the new system call. Finally, a system call might be overkill for a simple information exchange channel.

For the scope of the oven-based *Android* embedding process, the validity of a few of

these arguments might be diminished. A newly built *Linux* kernel with a custom system call, specifically designed for an embedded scenario with a manageable count of target hardware, is not intended for refining the *Linux* kernel itself but to serve a specific purpose. Consequently the diversity of hardware would probably be slim and so would the effort for integration of the new system call. Besides, the new kernel would be a rather independent built for a distinct purpose and therefore more lightweight and flexible towards changes. Nonetheless, compatibility to the latest *Android*-friendly *Linux* kernel is always desirable, not least due the benefits that come with an improved and enhanced kernel built. By implication, this trade-off between compatibility and independence has to be considered before deciding whether a new system call should be added or not.

Utilizing System Calls

For plain read/write operations, e.g. reading/writing data from/to devices, the system calls *sys_read*/*sys_write* (see the system call table 4.1) can be utilized. In *Linux*, device files make devices accessible as if they were files [42]. The first parameter of *sys_read*/*sys_write* is the file descriptor that identifies the file, and thus the respective device. The second parameter is a pointer to data which is relevant for reading or writing. In the case of *sys_read*, kernel space writes data to this buffer. Considering *sys_write*, the buffer holds the data that is about to be written to the device. The third argument represents the number of bytes that are about to be read/written from/to.

When it comes to diverse devices, drivers can provide a distinct interface that might not be (easily) accessible with generic *sys_read*/*sys_write* system calls. As most devices are only directly accessible from within the kernel, another mechanism is desired which is specifically designed to communicate with device drivers.

4.2.2 ioctl

This exactly was the motivation behind **input/output control (ioctl)**. As its name suggests, ioctl is usually considered when it comes to controlling a device. ioctl was explicitly

designed as a system call that handles device-specific input and output [2]. The prototype ioctl call is shown in code 4.5.

Code 4.5: The ioctl prototype in user space in *sys/ioctl.h*

```
int ioctl(int fd, unsigned long request, ...);
```

The first parameter of a ioctl call is the file descriptor which specifies the device to talk to. The second parameter is the request code which tells the device to perform the action that is specified by the request code. The third optional argument is untyped (usually *char *argp*) and might also be a plain integer that adds additional information to the request code [2]. Besides, the third argument is often a pointer to relevant memory such as necessary data for the execution of an intended action or a pointer to data that is about to be written to by the driver when data from the driver is requested.

On the device driver side, the kernel space pendant to the ioctl function is illustrated in code 4.6 [2].

Code 4.6: The ioctl kernel space function of the receiver

```
int (*ioctl) (struct inode *inode, struct file *filp,
unsigned int request, unsigned long arg);
```

Both structs *inode* and *file* correspond to the file descriptor (*fd*) parameter of the user space function. The third argument is the same request code that is passed by the user space function.

In order to prevent ambiguity of ioctl calls, there is an official list of ioctl calls (see the *ioctl-number.txt* in the *Linux* documentation). Similar to the official assignment of a system call number, an official ioctl entry might not be of high priority in the scenario of an isolated embedded *Android* OS project. Nonetheless, in regard of future necessities, such as growing capabilities alongside with increasing hardware performance, compatibility might turn out to be of significant importance.

4.2.3 sysfs

Sysfs was introduced with the *Linux* 2.6 kernel [30] and enables user space processes to access kernel space data through a virtual file system [30]. Similar to ioctl (see chapter 4.2.1), sysfs originates from the need for communication between user space and a device driver. Usually, the virtual file system is mounted on in the */sys* directory and includes for instance a *devices* folder which contains a global hierarchy of the physical devices. In general, sysfs features two interfaces: the interface to export data from the kernel and the interface for accessing that data from user space. Due to a mapping between the files exposed in the virtual file system within user space and the actual kernel data, modifications to the former will cause respective changes of the latter. Internally, this mapping is implemented with the help of netlink sockets (see chapter 4.2.4) [31]. Table 4.2 depicts the representation of the data structure in the kernel and its equivalent in the virtual file system:

Internal	**External**
Kernel Objects	Directories
Object Attributes	Regular Files
Object Relationships	Symbolic Links

Table 4.2: The mapping of the internal data structures and the external virtual file system [30].

An exported kernel object is usually called kobject. Directories and subdirectories reflect the hierarchical status of kobjects. Attributes of kobjects are usually represented by ASCII files with each file holding either a single attribute (for clarity) or several attributes (for the sake of efficiency). The kobject attribute is defined in code 4.7 [30].

Code 4.7: The definition of a kobject attribute

```
struct attribute {
  char *name;
  struct module *owner;
  umode_t  mode;
};
```

The *owner* struct points to the module where the attribute code is located. The *mode* value sets the file mode which manages permissions such as for instance read-only.

The following illustrates the API functions to edit kobjects (see code 4.8) as well as their attributes (see code 4.9) from user space [30].

Code 4.8: API functions to modify sysfs directories/kobjects

```
int sysfs_create_dir(struct kobject *k);
void sysfs_remove_dir(struct kobject *k);
int sysfs_rename_dir(struct kobject *, const char *new_name);
```

Code 4.9: API functions to modify sysfs files/kobjects' attributes

```
int sysfs_create_file(struct kobject *, const struct attribute *);
void sysfs_remove_file(struct kobject *, const struct attribute *);
int sysfs_update_file(struct kobject *, const struct attribute *);
```

The return values of type int comply with the *Linux* error code convention and return a 0 on success and a negative number (specifying the error code) on error. The API functions are relatively straightforward and feature the basic CRUD (create, read, update, delete) functions. Reading attributes will invoke the *show* function, which copies the attribute into the buffer that has page size. In *Linux*, page size usually starts from 4096 Bytes. That is why a single attribute file should not be too large in size.

Relationships between kobjects are defined via symbolic links [30]. Symbolic links hold a reference to another file or directory. These links can be edited using the functions of code 4.10.

Code 4.10: API functions to modify relationships between kobjects

```
int sysfs_create_link(struct kobject *kobj, struct kobject *target,
    char *name);
void sysfs_remove_link(struct kobject *, char *name);
```

When creating a link, the first kobject is the source of the link and the second parameter is the destination.

4.2.4 Netlink Sockets

Netlink sockets were added with *Linux* 1.3 [31] and represent an IPC mechanism for full-duplex connectionless communication between user space and kernel space. Netlink sockets are defined by the address family *AF_NETLINK*, as opposed to TCP/IP where the address family is *AF_INET*. In comparison to other potential IPC mechanisms, netlink sockets are far easier to integrate than system calls or ioctl [10]. These other options require a rather effortful integration at the risk of polluting the kernel, thus causing instabilities. Netlink sockets merely require the protocol type to be added to the *include/linux/netlink.h* file of the *Linux* kernel and a proper client/server implementation for both the user and kernel space to communicate in a socket style manner [10]. Due to the socket API that manages message bursts, netlink sockets are restricted to asynchronous communication. When sending a message via netlink, the message is placed in the receiver's queue whereupon the receiver's reception handler will be invoked [10].

As opposed to system call or ioctl, netlink sockets support multicasting via group addresses. A multicast group is defined by a bitmask with a single bit within 4 Bytes, hence 32 multicast groups can be distinguished. Furthermore, netlink sockets feature full-duplex communication between user space and kernel space, while system calls and ioctl merely provide simplex communication, meaning the communication link can only be initiated unidirectionally from user space to kernel space. Consequently, when utilizing system calls or ioctl, information from the kernel needs to be polled by user space processes either on demand or periodically when listening for kernel space events [10].

Netlink sockets feature an API that is similar to the TCP/IP variant: *socket()*, *sendmsg()*, *recvmsg()*, *close()*. Although user space and kernel space netlink sockets feature similar capabilities, the netlink socket API for each side differs.
In order for an *Android* application to communicate via netlink sockets, the JNI needs to be utilized to set up a communication between the application (given it is written in Java) and its netlink socket implementation.

Socket Creation

In user space

A socket in user space can be created as shown in code 4.11 [1].

Code 4.11: Creating a socket in user space

```
int socket(int domain, int type, int protocol_type)
```

The *domain* needs to be set to *AF_NETLINK* to make it a netlink socket. The *type* sets the socket type and can be either set to *SOCK_RAW* to be of raw type or *SOCK_DGRAM* when making use of datagrams. Finally, the *protocol_type* determines the type of the netlink protocol and can be assigned to the following types: *NETLINK_ROUTE*, *NETLINK_FIREWALL*, *NETLINK_ARPD*, *NETLINK_ROUTE6 NETLINK_IP6_FW*. Besides the given types, a custom protocol type can be added [22].

This socket creation will return the socket's file descriptor. The socket now exists in the name space but is still missing its address [21].

The *sockaddr_nl* struct (see code 4.12) describes a netlink client in either kernel space or user space [22].

Code 4.12: Structure that specifies a socket's address

```
struct sockaddr_nl
{
  sa_family_t   nl_family; // AF_NETLINK
  unsigned short nl_pad;  // zero
  __u32      nl_pid;  // process pid
  __u32     nl_groups; // multicast bitmask
} nladdr;
```

Addressing the local netlink socket via unicast is done via *nl_pid*. Similar to the multicast bitmask, the *nl_pid* is also an 8 Byte integer. The obvious choice for *nl_pid* is the PID of the given process. However, it is also possible to define multiple sockets within distinct threads in one single process. Consequently, assigning the PID to these sockets

would render them indiscernible. In order to handle such scenarios, there are different Formulas (see code 4.13) to generate an appropriate *nl_pid* [10].

Code 4.13: Formulas to generate a nl_pid

```
Formula 1: nl_pid = getpid();
Formula 2: pthread_self() << 16 | getpid();
```

Within kernel space, the *nl_pid* should always be 0.

Finally, the socket's name space needs to be connected with its address. The *bind()* function "is assigning a name to a socket" [21], as indicated in code 4.14.

Code 4.14: Binding a socket client to a socket name space

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

In kernel space

The kernel space API for creating a netlink socket differs from the user space implementation. In kernel space, a netlink socket is created as shown in code 4.15.

Code 4.15: Creating a socket in kernel space

```
struct sock *netlink_kernel_create(int unit,
void (*callback_func)(struct sock *sk, int len));
```

The *unit* parameter is the protocol type and the second parameter points to a designated callback function (*callback_func*) that is invoked on the event of a received message through this socket [10].

Sending and Receiving Messages

In user space

Sending a message from user space requires the message to carry information about the destination address. This is supplied via another *sockaddr_nl* struct as the one above, but instead of providing information about the sender, it holds information about

the receiver(s). A message that is intended for the kernel should set 0 as the *nl_pid*. In case the message is destined for a user space process, the target's *nl_pid* should be set as destination *nl_pid*, which is the same as the targets PID, given that Formula 1 was used to generate the target socket's *nl_pid*. In order to send a multicast, all desired groups' bitmasks should be ORed together to determine the final *nl_groups* integer. In case a unicast is intended, a 0 should be assigned to *nl_groups* [10].

Furthermore, the message requires a header (see code 4.16) that is common among all netlink protocol types [22].

Code 4.16: The header of a netlink message

```
struct nlmsghdr
{
  __u32 nlmsg_len; // message length
  __u16 nlmsg_type; // message type
  __u16 nlmsg_flags; // additional flags
  __u32 nlmsg_seq; // sequence number
  __u32 nlmsg_pid; // PID of sending process
};
```

The message length includes the message header and the payload. The sequence number needs to be managed by the application itself to track acknowledgements [10]. Messages are sent in datagrams, similar to UDP. Consequently, there is no guarantee for a successful transmission. However, as with any other UDP communication channel, mechanisms that ensure delivery can be built upon the netlink socket infrastructure. Such mechanisms will utilize attributes such as the sequence number of the message header.

A netlink message is sent using this standard socket API function (see code 4.17) [23].

Code 4.17: The standard socket API function to send a (netlink) message

```
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

The first argument sets the file descriptor of the sending socket. The second argument points to the header of the message. The *msghdr* struct requires several properties

having set prior to sending, such as the target socket struct pointed to by *msg.msg_name*. The third parameter enables advanced features for the sending process [23].

It is worth noting that this *sendmsg* function is a system call [1].

In order to receive a netlink message in user space, an application must first allocate a buffer that is suffice in size to store the message headers and payloads. A message is received via the standard *recvmsg* function (see code 4.18) that is used to receive messages through sockets [24].

Code 4.18: The standard socket API function to receive a message via (netlink) sockets

```
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

The parameters of the *recvmsg* function are analogue to the arguments of *sendmsg* [24].

Similar to the *sendmsg* function, the *recvmsg* function is also a system call [1].

In kernel space

In kernel space, the API for sending a message requires the information about the sender and receiver to be added to the socket buffer (*skbuffer*) as illustrated in code 4.19 [10].

Code 4.19: Socket buffer attributes for sending a message from kernel space

```
NETLINK_CB(skbuffer).groups = loc_groups;
NETLINK_CB(skbuffer).pid = 0; //kernel pid is always 0
NETLINK_CB(skbuffer).dst_groups = dst_groups;
NETLINK_CB(skbuffer).dst_pid = dst_pid;
```

Having set this sender/receiver information, the actual message can be transmitted as shown in code 4.20 for a unicast or code 4.21 for a multicast [10].

Code 4.20: Sending a unicast from kernel space

```
int netlink_unicast(struct sock *ssk, struct sk_buff *skbuffer,
u32 pid, int nonblock);
```

Code 4.21: Sending a multicast from kernel space

```
void netlink_broadcast(struct sock *ssk, struct sk_buff *skbuffer,
u32 pid, u32 group, int allocation);
```

The *ssk* struct in code 4.20 and code 4.21 is the kernel space netlink socket as returned on creation when called *netlink_kernel_create()*. The message is held in *skbuffer->data* and the *pid* is the receiver's *pid*. In the case of a multicast (via *netlink_broadcast*) receivers are defined by their *group* bitmask [10].

To receive a netlink message in kernel space, the respective callback function should be defined on socket creation via the *netlink_kernel_create* function, as described in socket creation.

4.2.5 Binder and HAL

Binder is an inter-process communication (IPC) framework used in *Android*. As other operating systems, *Android* runs applications and services on separate processes due to memory management, stability, security etc. In order for these processes to communicate, Binder was introduced. Processes can be identified for instance via process identifier (PID), parent PID (PPID), group identifier (GID) or user identifier (UID). Binder is essential for substantial functions on *Android*, be it the application component management (such as the *Activity* life-cycle), utilizing the display, audio in- and output and any other hardware usage [5]. According to Dianne Hackborn, one of the developers of Binder, "In the Android platform, the binder is used for nearly everything that happens across processes in the core platform" [9]. Binder is not a single method for IPC but a set of mechanisms that are used by *Android* for IPC.

This chapter will look into two *Android* IPC mechanisms which utilize Binder, meaning Intent and Messenger. Afterwards, the components of the Binder framework will be examined followed by a conceptual approach on how to integrate and use a new HAL module.

Intent and Messenger

Intent and Messenger are IPC mechanisms on *Android* which are not components of the Binder framework but mechanisms that are based on Binder. Both variants can pass data between processes by eventually utilizing Binder's faculties. However, these implementations feature a certain latency due to their immanent overhead [5].

Intent utilizes *IntentResolver* which identifies the desired receiver among a list of registered receivers. Hence the potential delay correlates with the amount of registered receivers.

Messenger places a remote *Handler* in another process and pushes messages to the message queue. Consequently the delay depends on the current amount of pending messages.

In general, the *Intent* variant tends to feature a higher latency since the lookup of the desired receiver often exceeds the time of a message pending in the message queue [5]. However, in the scenario of this thesis where the AOSP is significantly trimmed by excluding several redundant components (see chapter 3.4.2), the amount of receivers will drop considerably. As a result, the delay of both variants, *Intent* and *Messenger*, might approximate.

A more efficient way for IPC on *Android* is making use of a custom Binder implementation with interfaces that are defined via *AIDL*.

AIDL

Android Interface Definition Language (AIDL) is used to describe the business operations of a service that can be accessed remotely by a client. The service is described in a .aidl file with a syntax similar to Java and may look as illustrated in code 4.22 [5]. Such an *AIDL* definition as shown in code 4.22 generates the respective Java code. In fact, code for two different purposes will be created. For one thing, a Proxy class for accessing the service by a remote client is generated. Additionally, a Stub class is created which is used by the service and holds the implementations of the remote methods. Proxies

and stubs are used by clients and services to abstract from the intricacies of the Binder protocol (see figure 4.3).

Code 4.22: A simple interface of a service described via AIDL

```
package com.name.appname;
import com.name.appname.Test;
interface ITestService {
  Test getTestById(int id);
  void save(inout Test test);
  void delete(in Test test);
}
```

The tags *in*, *out* and *inout* specify the direction of the marshalling process: caller to callee (*in*), callee to caller *out* and bidirectional *inout*. Marshalling is the process of transforming higher level data structures for storage or transmission purposes into *Parcels*. The reverse process is called unmarshalling and restores the high level data structures such as data objects. A *Parcel* is a message container that can be transmitted through the *IBinder* interface as defined via AIDL [12].
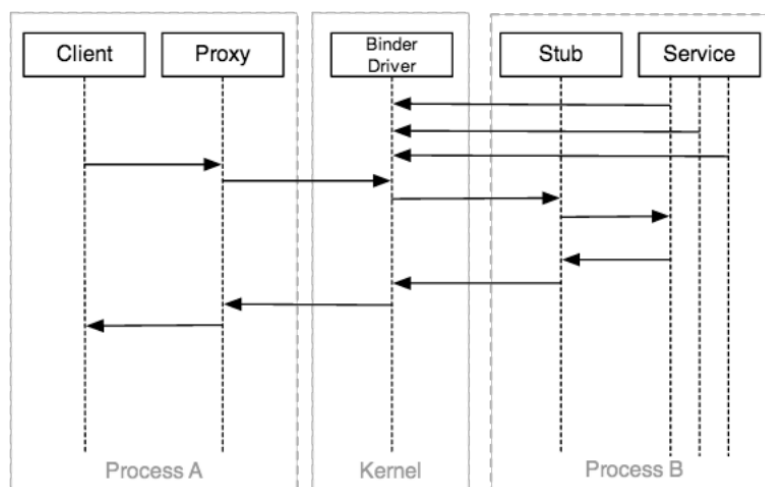
Figure 4.3: Clients and Services abstract from the Binder protocol via Proxies and Stubs [5].

Binder Driver

All Binder driven communication is enabled and conducted through the *Binder Driver*,
a kernel-level driver that primarily utilizes ioctl (see code 4.23) for communicating (see
chapter 4.2.2).

Code 4.23: The ioctl call is usually invoked by the *Binder Driver*

```
ioctl(binderFd, BINDER_WRITE_READ, &bwd);
```

BINDER_WRITE_READ is the most important command and enables data transmission.
The third argument is a reference to the data buffer and is defined as shown in code
4.24 [5].

Code 4.24: The definition of the *binder_write_read* struct

```
struct binder_write_read {
  signed long write_size; /* bytes to write */
  signed long write_consumed; /* bytes consumed by driver */
  unsigned long write_buffer;
  signed long read_size; /* bytes to read */
  signed long read_consumed; /* bytes consumed by driver */
  unsigned long read_buffer;
};
```

write_buffer holds commands that should be performed by the driver, such as increment-
ing/decrementing object references. Analogue to the *write_buffer* in kernel space, upon
return, the *read_buffer* contains commands for the user space thread to perform.
As described in chapter 4.2.2, ioctl is usually utilized when it comes to controlling a driver.
This is also the case with the *Binder Driver*. Other commands that can be sent to the
Binder Driver via the ioctl system call are for instance *BINDER_SET_MAX_THREADS*,
which sets the number of threads for each process when handling requests or for exam-
ple *BINDER_SET_CONTEXT_MGR*, which sets the *Binder Driver's ContextManager*
via first come first serve (see figure 4.4) [5].

Service

A service in *Android* is a component of an application that runs in background, either to perform certain tasks or to offer functionality to other applications.

In order to locate a service to communicate with, a *ServiceManager* is required. The *ServiceManager* (also known as *Context Manager*) is a service itself and registers with the *Binder Driver* in the early stages of *Android*'s init process [5]. Subsequently, other services register with the *Context Manager* through the *Binder Driver*. Then, a client can query the *Context Manager* to get a handle on the desired service (see figure 4.4).
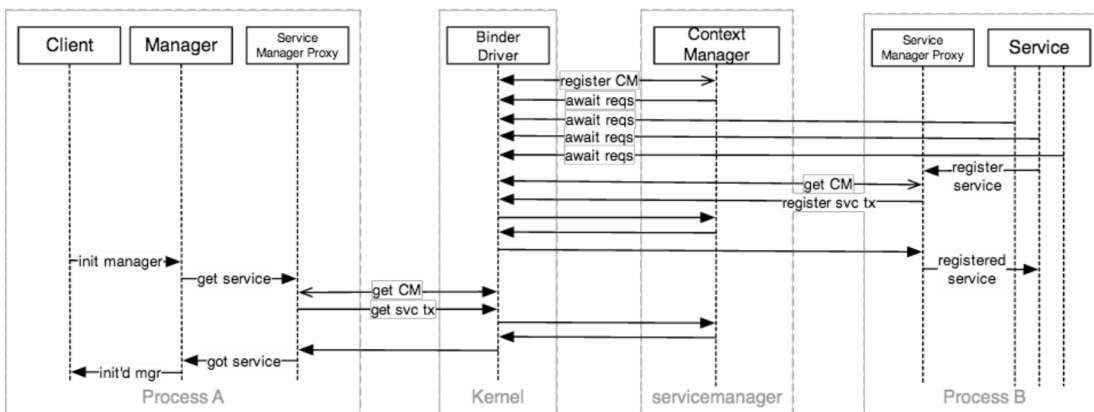
Figure 4.4: A service is registered with the *Context Manager* via the *Binder Driver*. When a service is requested by a client, the *Binder Driver* fetches the handle of the inquired service from the *Context Manager* and returns it [5].

Similar to Proxies, Managers provide a further level of abstraction towards the Client and trim the exposed functions to a subset which is relevant for the Client.

A service can either be added to an arbitrary application or directly to the *Android* framework as a system service by placing a service implementation (.java file) in the *frameworks/base/services/java/com/android/server/* folder. Such a system service is necessary when adding support for new hardware.

In *Android*, hardware types such as cameras or sensors are accessed through their respective system services which in turn have access to the devices' functions that are exposed by the HAL definition. There is for instance a camera system service and a camera HAL definition.

HAL

Figure 4.5 image depicts the individual components and affiliations relevant for a custom hardware integration.
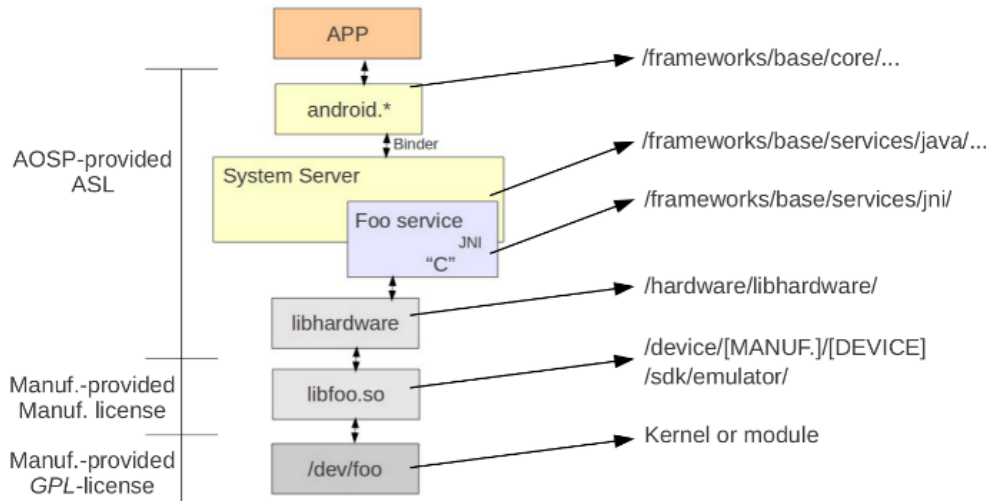

Figure 4.5: The implementation-specific components for extending *Android*'s HAL [43].

In the following, the individual components are described in the order as they appear in figure 4.5 from top to bottom (starting with the service).

In order for a system service to communicate with the HAL's C implementation, the C++ portion of the system service has to be added to *frameworks/base/services/jni/* (see figure 4.5). For the C++ implementation of the system service to be loaded in the first place, the *Android.mk* and *onload.cpp* within the *frameworks/base/services/jni/* folder need to include the respective file.

The constructor of the Java service should invoke a native initialization call of the C++ portion of the system service in order to load the HAL module. Via *hw_get_module* (see code 4.25) the *dlopen()* function will be invoked which will result in a shared library to be loaded into the address space of the system service and thus, the device functions will be available to the system service [44].

Code 4.25: The native initialization of the system service that loads a custom HAL module

```
static jint init_native(JNIEnv *env, jobject clazz)
{
  int err;
  hw_module_t* module;
  customhw_device_t* dev = NULL;

  err = hw_get_module(CUSTOMHW_HARDWARE_MODULE_ID, (hw_module_t
      const**)&module);
  if (err == 0) {
    if (module->methods->open(module, "", ((hw_device_t**) &dev)) !=
        0){
      return 0;
    }
  }


  return (jint)dev;
}
```

The common definition for a new hardware type is defined in a header file located in *hardware/libhardware/include/hardware/* (see figure 4.5) and is agnostic to the subtleties of particular hardware. Code 4.26 defines a new hardware type (a *customhw* device) with three prototype function definitions [44].

Code 4.26: Definition of a new hardware type with prototype function definitions

```
__BEGIN_DECLS
#define CUSTOMHW_HARDWARE_MODULE_ID "customhw"

struct customhw_device_t {
  struct hw_device_t common;
  int (*read)(char* buffer, int length);
  int (*write)(char* buffer, int length);
};
__END_DECLS
```

For the actual hardware module implementation (the shared library), different vendors, distributing varying hardware, usually provide custom HAL modules. Therefore the respective HAL modules are located in device specific path of the AOSP (see figure 4.5). Consequently, when building for the emulator, the target location is *sdk/emulator*. Code 4.27 indicates the contents of a HAL module [44].

Code 4.27: A basic implementation of a HAL module

```
int fd = 0;

int customhw_read(char* buffer, int length)
{
//implementation of the read function
}
int customhw_write(char* buffer, int length)
{
//implementation of the write function
}

static int open_customhw(const struct hw_module_t* module,
  char const* name, struct hw_device_t** device)
{
  struct customhw_device_t *device = malloc(sizeof(struct
    customhw_device_t));
  memset(device, 0, sizeof(*device));

  device->read = customhw_read;
  device->write = customhw_write;

  //further initialization steps were omitted

  fd = open("/<driver-path>", O_RDWR);

  return 0
}
```

The *open_customhw* function initializes the *device* struct of type *customhw_device_t*. This is the same type as previously defined in the hardware agnostic header file (see code 4.26) [44]. The prototype device functions (*read* and *write*) are mapped to the implementation of the specific HAL module functions. Additionally, the *open* function connects the device driver to the HAL and returns a file descriptor as already seen in system calls (chapter 4.2.1), ioctl (chapter 4.2.2) and sysfs (chapter 4.2.3). For clarification, the driver in figure 4.5 is located in the */dev/foo* folder.

Additionally, the two structs of code 4.28 have to be added to the HAL module in order to be registered as such [44].

Code 4.28: Structs required for a HAL module to be registered as such

```
static struct hw_module_methods_t customhw_module_methods = {
.open = open_customhw
};


const struct hw_module_t HAL_MODULE_INFO_SYM = {
.tag = HARDWARE_MODULE_TAG,
.version_major = 1,
.version_minor = 0,
.id = CUSTOMHW_HARDWARE_MODULE_ID,
.name = "Custom HW Module",
.author = "Example inc.",
.methods = &customhw_module_methods,
};
```

The two structs of code 4.28 render this HAL module accessible for opening via the system service. The *open* function of *customhw_module_methods* is the same function that is called by the system service's *init_native* function which invokes *module->methods->open(...)* as described earlier . Additionally, further information of the HAL module is set such as its id and other meta data, e.g. the name of the module and its author [44].

Finally, after the integration is complete, in order to utilize these device functions, the system service can call functions of the HAL definition via JNI, for instance *device->read(...)*.

An application can access these device functions through the respective system service. The applciation can obtain a handle on the system service through the *ContextManager* as described in figure 4.4.

It becomes clear that the purpose of the HAL is, as its name suggests, abstraction of (manufacturer specific) hardware. To be more precise, the header file located in *hardware/libhardware/include/hardware/* holds the common ground API functions of a device, while their implementation is manufacturer specific and thus, located in the device specific folder of the manufacturer within the AOSP (see figure 4.5).

Device functions (e.g. to control or read from a device) are defined from the system service all the way down to the device driver (see figure 4.5). The communication between the HAL module (the shared library) in user space and the hardware modules in kernel space is usually not uniform. Potential mechanisms for such communication are for instance sockets (see chapter 4.2.4) or sysfs (see chapter 4.2.3). This freedom is possible because the *Android* OS does not specify the interaction between the shared library and the driver [44].

Besides this user space to kernel space communication, the application that intends to interact with the hardware needs to communicate with the respective system service, hence a further IPC channel from user space to user space is necessary. This can be achieved with the Binder framework as described in this chapter.

4.3 Summary

This chapter covered several IPC mechanisms that are potentially eligible for utilizing when interacting with oven hardware from an *Android* OS. The application implementing the user interface should be able to interact with the oven hardware.

In order to make a valid choice of which IPC mechanism is applicable, several aspects of the kernel module/driver and the system as a whole have to be considered.

For one thing, the uniqueness of provided hardware functions is of relevance. The IPC mechanism should not only be able to handle all the existing capabilities of the driver, but should also be rather easily expandable.

Furthermore, latency of an IPC mechanism is generally a point of interest. However, oven hardware might not be prone to suffer from delays in the range of milliseconds. Nonetheless, a mechanism that enables kernel space to initiate messaging, as well, or that provides event based signaling would be preferred.

Finally, the type of data that will be interchanged between the application and the driver has to be considered. *Linux* supports different driver types [2]. Character devices are usually accessed through the file system with a stream of bytes, such as devices featuring a serial/parallel port interface. Similar to character devices, block devices are also accessed via the file system. A block device (for instance a hard disk drive) usually hosts its own file system and can be accessed by reading/writing data of block size (usually 512 bytes or more). Lastly, network interfaces are devices (or software modules) that can exchange data with other hosts by receiving as well as sending data packets. Consequently, some IPC mechanisms might not be as well suited for specific driver types as others.

The oven hardware will most likely resemble a character device.

In chapter 4.2, four *Linux*-based IPC mechanisms were examined. These four IPC mechanisms are generally provided by *Linux* and thus also work on *Android*.

System Call

As a conceptual fact, it is worth noting that every introduced mechanism in chapter 4.2 ultimately utilizes a system call implementation of some sort underneath.

Utilizing existing system calls to interact with the oven hardware is technically possible but its practicability depends on the implementation of the device driver. In general, a minimal character device driver usually implements system calls such as *read*, *write*, *open* and *close*. Even though system calls might be practicable for communication between the application and the device driver, this might change in the future when the hardware is enhanced and the driver is developed further. However, system calls are

indeed a potential choice for interacting with a simple driver.

For supporting a more complex driver, integrating a new system call is generally discouraged [27] (e.g. a system call that is specifically designed for interacting with a certain hardware module/driver) and is entailed by rather effortful implementation work within the kernel. Besides, due to the rather straightforward nature of an oven hardware driver, such an undertaking does not seem appropriate.

ioctl

For controlling the oven hardware, the ioctl system call (see chapter 4.2.2) seems to be a valid IPC mechanism. As ioctl is specifically designed for interacting with device drivers via the file system, utilizing ioctl confirms the separation of responsibilities within the *Linux* kernel. The ioctl system call takes a command as well as a pointer to memory for executing operations. This way, simple operations (specified by the command parameter) can be performed, such as turning hardware components on/off. Furthermore, the additional parameter can point to relevant memory for the execution of the respective command, either by providing additional information or by pointing to memory that is designated to be written to by the hardware module/driver, such as inquired values.

However, adding a ioctl system call seems to result in more modifications within the kernel as opposed other IPC mechanisms such as sysfs or netlink sockets. Further on, ioctl is intended to be initiated from user space and thus, information of interest that might potentially change needs to be polled periodically.

sysfs

Aside from ioctl, sysfs (see chapter 4.2.3) might also seem to be a valid choice for controlling the oven hardware. In their capabilities, ioctl and sysfs are rather similar. In general, utilizing sysfs seems to result in a more assessable interface by providing a straightforward framework of CRUD functions to modify devices and their attributes. Furthermore, sysfs enables establishing symbolic links, such as binding a driver to a device. A rather complex hardware driver might benefit from sysfs as an IPC mechanism as it seems to feature better scalability. This is also an advantage in terms of potential expansion of the hardware's capabilities in the future which will result in a more complex driver. Since ioctl utilizes the command parameter, this is usually handled by a switch-

case statement which can grow large as the capabilities of the hardware increase. Considering events initiated from kernel space, the virtual file system in user space is updated by the kernel via netlink sockets. However, this only affects the virtual file system and not the application itself. By implication, the application still needs to poll data from the file system in order to determine changes in the hardware. Additionally, sysfs provides a text based interface which turns out to be impracticable. The application needs to parse data on input and compose it respectively on output to comply with the file system.

Netlink Sockets

The next IPC mechanism that was examined utilizes netlink sockets (chapter 4.2.4) to establish a communication channel between user space and kernel space. Initially, netlink sockets were introduced as character driver interfaces that enable bidirectional communication between user space and kernel space. Netlink sockets provide a socket API to send and receive messages from both user space and kernel space. In comparison to IPC methods such as system call, ioctl and sysfs, which usually require messaging to be invoked from user space, netlink sockets enable the kernel space to initiate messaging as well. It seems natural that netlink sockets are the preferred IPC mechanism when it comes network applications which intend to communicate with kernel space [1]. Considering an oven hardware driver to utilize netlink sockets as an IPC mechanism might turn out to be a flexible choice due to the generic socket interface. Additionally, the full-duplex communication provided by netlink sockets increases responsiveness over most other IPC mechanisms. Furthermore, the implementation of a netlink socket interface will result in a very thin implementation within the kernel system.

The fact that system call, ioctl and sysfs usually don't provide a way to initiate interactions from the kernel requires these IPC mechanisms to periodically poll for potential new data that is of interest. The user space application needs to invoke periodical read operation to retrieve the current hardware status, even if no changes occurred. This implies that at the time a value of interest actually changes, the application will receive this information at the time of the next read operation, causing a certain delay.

A simple example for such interesting data of the oven hardware would be its current

temperature. The user might want to see the actual temperature of the oven (as ascertained by the hardware) and possibly a (visual) clue in case for instance the preheating process is finished. Due to periodic polling, such information will be displayed in a delayed manner through the *Android* application in user space. A tolerable delay obviously varies from a case to case basis so the polling frequency should depend on its potential result. However, as mentioned earlier, the scenario of interacting with an oven might not suffer severely from such delays.

The open source nature of *Linux*, accompanied with continuous development, results in a wide and still increasing variety of components of the *Linux* kernel. As a consequence, when a mechanism is lacking specific desired capabilities, it is likely that another mechanism was already merged into the *Linux* kernel tree that provides that missing feature. It so happens that there are indeed mechanisms that enable notifications when changes in the virtual file system occur. *libudev* is a library for sysfs that is intended to provide rich capabilities for device management [30] and incorporates a monitoring interface. This interface tracks changes within the virtual file system and returns a handle to the object that changed. The object contains a string of the action that occurred, for instance *add*, *remove*, *change* or *move*. There is a filtering mechanism within the monitoring interface of *libudev* which filters the directories and files that will be monitored.

Binder and HAL

Finally, chapter 4.2.5 described the conventional process of integrating and communicating with hardware within the *Android* OS. Both the integration as well as the communication appear to require significantly more overhead than the previously examined IPC mechanisms.

One reason for this extra effort is the HAL. As *Android* is intended to be a portable OS, the HAL defines a device type interface that is agnostic to the particular hardware. This HAL definition is exposed to the *Android* framework via system services.

The second reason for the overhead that results from this standard *Android* process for hardware integration and communication is the system service. The interface for the system service needs to be defined (via AIDL) and the interaction between the system service and the application is established with the help of the Binder framework.

All these mechanisms are usually built on top of the previously described IPC methods. When utilizing Binder, system services and the HAL, a common ground is accomplished that is conform to the *Android* framework. Having to decide which IPC mechanism to choose, once more this circumstance requires a decision between compatibility, causing overhead, and individuality, implying less effort but also losses in compatibility. In comparison to similar challenges with the previously discussed IPC mechanisms, this *Android* particular method implies a considerable amount of extra effort, rendering this particular decision significantly more crucial.

In conclusion, *Linux* provides a wide variety of possibilities to achieve the desired task. The utilized mechanism should be chosen by considering the requirements and potential future development. Furthermore, economic factors have to be factored in, such as the development effort and component licensing.

Considering the *Linux*-based IPC mechanisms and estimating a rather rich set of oven hardware capabilities, either sysfs or netlink sockets appear to be the preferred mechanism for the user space oven application to communicate with the respective kernel space oven hardware module/driver. Thinking about a long-term *Android* deployment, the standard *Android* method seems, despite the extra effort, the preferred choice, thus ensuring a high level of compatibility towards future *Android* versions to come.

5

Conclusion and Future Work

This thesis was conducted in cooperation with BSH with the purpose to investigate the applicability of *Android* to be embedded in an oven to provide a user interface. This investigation consisted of three main topics.

Starting with chapter 2, the graphical performance of *Android* was examined by utilizing two different implementations. One implementation was achieved with a layout of *Views* while the second variant was conducted with the *LibGDX* library. For testing data, three actual oven interfaces were recreated (see chapter 2.2) with the help of documentation that was provided by BSH. Such realistic data is necessary for the conclusions that were drawn from this analysis to be valid. The three implemented interfaces featured varying workload, meaning a different amount of simultaneously rendered elements and animations.

In order to measure performance (see chapter 2.3), a *FrameInspector* class was introduced in both implementation variants. The *FrameInspector* gathered a timestamp after each rendering of a frame. Consequently, the frames per second can be derived from the intervals between these timestamps. Both implementation variants included a nearly identical *FrameInspector* class. Therefore the relation between the results of both variants can be considered valid.

The results made clear that both variants are able to render rather simple graphics with up to 60 fps (see chapter 2.4). When increasing the workload, the *View* based implementation drops to 30-40 fps while the *LibGDX* implementation is still able to maintain close to 60 fps.

Consequently, it becomes clear that *Android* is capable to deliver the desired graphi-

cal performance. Even though the *View* based implementation may suffer when the workload gets rather high, it still handled rendering with a solid 30 fps which is generally sufficient. The *LibGDX* variant illustrated that there is still room for even more complex graphics.

Chapter 3 investigated the required modifications of an *Android* application as well as of the *Android* OS itself to be eligible for being embedded into an oven. As *Android* is designed for mobile devices [37], some aspects require modification to achieve an appropriate behavior for a stationary oven.

Such aspects comprise for instance disabling physical as well as virtual *Android*-specific buttons (see chapter 3.4.3). Since only one application should be accessible on the oven and this oven application is supposed to be controlled solely via its user interface, *Android*-specific buttons (such as the back button or home button) are rendered obsolete. Further on, the oven application should immediately start after the device has finished its booting process and never lose focus (see chapter 3.4.1). To ensure this application to permanently run in foreground, any potential entry point to other system components or applications should be blocked, this comprises the handling of any uncaught exceptions that might occur (see chapter 3.4.4). Additionally, components/applications that are not relevant to the oven scenario, such as the default *Android* home application, should be removed from the AOSP to further ensure stability and improve performance (see chapter 3.4.2).

Besides the identification of such aspects, respective solutions for these modifications were proposed in this chapter. In some cases, alternative solutions were given which accomplish the same objective but in a different way. The alternative to for instance disabling a component would be the removal of the such. The decision between the former and the latter solution could be influenced by potential future requirements which imply the usage of these components in future versions. A component that could be of use in future versions to come might be kept in the system. Consequently simply disabling said component would be a valid approach in this case. On the other hand, components that are most likely of no relevance to the current system as well as in regard of future versions could be disabled and removed entirely.

Since all identified aspects that require modification seem manageable with reasonable amount of work, the implication can be drawn that *Android* is an appropriate base for an embedding project.

Chapter 4 examined several potential IPC mechanisms that are potentially eligible to provide a communication channel between an *Android* application in user space and a hardware module/driver in kernel space that is responsible for the oven hardware. The *Android* application is supposed to provide the oven's user interface. Consequently, the application should be capable to read and control the underlying oven hardware.

Since *Android* is built on top of a *Linux* kernel, *Linux* specific IPC mechanisms were examined. The basic IPC method for calling the kernel space from user space in *Linux* is system call (see chapter 4.2.1). In fact all investigated IPC mechanisms in this chapter make use of system calls. However, it is discouraged to add a new custom system call into the *Linux* kernel and the usage of existing system calls might turn out to be too generic for the interface of a specific device module/driver.

Consequently, the ioctl system call was specifically introduced to control devices from user space (see chapter 4.2.2). This makes ioctl generally a valid IPC method for an *Android*-driven oven.

Besides ioctl, sysfs is another potentially eligible IPC mechanism for the oven scenario (see chapter 4.2.3). Since devices are usually represented via device files in *Linux*, which mirror the properties of devices, sysfs enables the export of these files into user space and provides methods to access said files. Using sysfs to modify device files usually results in a simple and clear interface.

A further potential method for IPC is provided via netlink sockets (see chapter 4.2.4). Netlink sockets can be used for the communication between user space and kernel space with a socket-like API. As opposed to the previously mentioned IPC methods, when using netlink sockets, messaging can be initiated from kernel space, as well. This way, the user space process does not need to periodically poll for potential data of interest but the kernel can send a message to the user space process in case a value of interest has changed.

However, there are other mechanisms available on *Linux* that serve the purpose to

invoke such events when kernel space data changes. The *libudev* library for instance can be used when working with sysfs and is able to monitor changes in the virtual file system, which holds a mapping of the device properties.

In *Android*, new hardware is usually integrated with the HAL, which abstracts from the actual hardware and defines a generic interface of a hardware type. In order to access the functions of a HAL definition, a system service has to be defined which interfaces with the HAL. An *Android* application can communicate with the respective system service by making use of the Binder framework. In *Android* hardware is usually accessed in this manner. Therefore, complying with this infrastructure seems to be a valid approach when considering long term deployment of *Android*. Nonetheless, other IPC mechanisms such as ioctl or sysfs should also be considered as they require less effort. The decision between potential IPC mechanisms should consider this trade-off between compatibility to the *Android* system, effort and actual applicability of the particular IPC mechanism that is in focus.

Chapters 3 and 4 illustrated the necessary work when embedding *Android* into home appliances, such as ovens. There are usually multiple solutions for specific objectives, which implies that *Android* is generally applicable as an embedded OS for diverse devices. This is not least due to the open source nature of the *Linux* kernel as well as the AOSP, which renders both systems highly customizable.
The implementation work to make *Android* suitable for embedding seems manageable and there is a lot of support for both the *Linux* kernel as well as for *Android*.
Furthermore, as *Android* is already optimized for restricted hardware, it seems to provide a very efficient base which enables developers to work with a high level programming language.

Additional work that was not discussed in this thesis is the implementation and/or integration of a specific device driver for the oven hardware. Such work was not in the focus of this thesis since it is a matter of the *Linux* kernel and not specifically concerning the AOSP. Additionally, the hardware module/driver itself is not of much relevance to the assessment of *Android* as an embedded OS for home appliances, specifically for

ovens. In order to identify potential other tasks, a thorough requirement analysis has to be conducted, which is out the scope of this thesis.

In regard of future work in the long term, an estimation of future requirements can provide conclusions about the applicability of *Android*. Considering the prevalence of *Android* in diverse sub-categories of the mobile domain [36], such as smartphones, tables or smartwatches, it seems reasonable to consider *Android* as an embedded OS for diverse devices, such as home appliances. Although *Android* might cause a certain overhead when deployed in rather simple devices, their capabilities tend to increase in the future and the wide range of *Android*'s features might turn out to be a considerable advantage. The user interface of an oven, such as the oven model introduced in figure 1.1, already provides a rich set of capabilities that are well suited for being handled with *Android*.

As opposed to personal computing, where one person is usually interacting with one computer, the first implications of ubiquitous computing are already apparent. People are surrounded by a growing number of computers in their everyday lives. These computers tend to become more intricate as their capabilities grow. Everyday objects, such as home appliances, might feature integrated sensors for a more autonomous behavior and/or come with network interfaces to participate in the internet of things. There are already such devices available on the market. They might not be abundant and as rich in their features yet, but their expansion parallels the technological progress.
As new features bring new challenges, it is very likely that the AOSP will be extended to meet future requirements.

List of Figures

List of Tables

Bibliography

[1] Christian Benvenuti. *Understanding Linux network internals*. O'Reilly, Sebastapol, Calif, 2006.

[2] Jonathan Corbet. *Linux device drivers*. O'Reilly, Beijing Sebastopol, CA, 2005.

[3] Joshua Drake. *Android hacker's handbook*. Wiley, Indianapolis, IN, 2014.

[4] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying magic: high-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 81–90, New York, NY, USA, 2009. ACM.

[5] Aleksandar (Saša) Gargenta. Deep dive into android ipc/binder framework at android builders summit 2013. `http://events.linuxfoundation.org/images/stories/slides/abs2013_gargentas.pdf`, 2013.

[6] Philip Geiger, Marc Schickler, Rüdiger Pryss, Johannes Schobel, and Manfred Reichert. Location-based mobile augmented reality applications: Challenges, examples, lessons learned. In *10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps*, pages 383–394, April 2014.

[7] BSH Hausgeräte GmbH. Bosch online presse-center. `http://presse.bosch-home.at/News_Detail.aspx?id=24539&menueid=2016`, 2015. access: 2015-05-17.

[8] gregose. Linux syscall reference. `http://syscalls.kernelgrok.com/`, 2013. access: 2015-09-01.

[9] Dianne Hackborn. Lkml: Dianne hackborn: Re: [patch 1/6] staging: android: binder: Remove some funny & usage.

[10] Kevin Kaichuan He. Kernel korner: Why and how to use netlink socket. *Linux J.*, 2005(130):11–, 2005.

[11] Andrew Hoog. *Android Forensics: Investigation, Analysis and Mobile Security for Google Android*. Syngress, 2011.

[12] Google Inc. Android interface definition language (aidl) | android developers.

Bibliography

[13] Google Inc. Android interfaces and architecture | android open source project. `https://source.android.com/devices/`. access: 2015-06-25.

[14] Google Inc. Hardware acceleration | android developers. `http://developer.android.com/guide/topics/graphics/hardware-accel.html`. access: 2015-06-18.

[15] Google Inc. Hiding the navigation bar | android developers. `https://developer.android.com/training/system-ui/navigation.html`. access: 2015-07-11.

[16] Google Inc. Hiding the status bar | android developers. `https://developer.android.com/training/system-ui/status.html`. access: 2015-07-13.

[17] Google Inc. Intent | android developers. `http://developer.android.com/reference/android/content/Intent.html`, 2015. access: 2015-07-03.

[18] Google Inc. Key layout files | android developers. `https://source.android.com/devices/input/key-layout-files.html`, 2015. access: 2015-07-23.

[19] Google Inc. Viewgroup | android developers. `http://developer.android.com/reference/android/view/ViewGroup.html`, 2015. access: 2015-06-18.

[20] Texas Instruments Incorporated. Am335x evaluation module - tmdxevm3358 - ti tool folder. `http://www.ti.com/tool/tmdxevm3358`, 2012. access: 2015-06-02.

[21] Michael Kerrisk. bind(2) - linux manual page. `http://man7.org/linux/man-pages/man2/bind.2.html`, 2015. access: 2015-09-25.

[22] Michael Kerrisk. netlink(7) - linux manual page. `http://man7.org/linux/man-pages/man7/netlink.7.html`, 2015. access: 2015-09-25.

[23] Michael Kerrisk. recvmsg(3): receive message from socket - linux man page. `http://man7.org/linux/man-pages/man2/sendmsg.2.html`, 2015. access: 2015-09-25.

[24] Michael Kerrisk. recvmsg(3): receive message from socket - linux man page. `http://man7.org/linux/man-pages/man2/recvmsg.2.html`, 2015. access: 2015-09-25.

[25] KioWare®. Kiosk software from kioware - lockdown kiosk mode and secure kiosk browser. `http://www.kioware.com/`, 2015. access: 2015-09-17.

[26] Inc. Linux Kernel Organization. The linux kernel archives. `https://www.kernel.org/`, 2015. access: 2015-09-01.

[27] Robert Love. *Linux kernel development.* Addison-Wesley, Upper Saddle River, NJ, 2010.

[28] Robert Love. *Linux system programming.* O'Reilly Media, Inc, Sebastopol, CA, 2013.

[29] 42Gears Mobility Systems Pvt Ltd. Surelock | mobile device lockdown | supports android, ios, windows 7/8 and windows mobile/ce. `https://www.42gears.com/products/surelock/`, 2015. access: 2015-09-17.

[30] Patrick Mochel. The sysfs filesystem. In *Linux Symposium*, pages 313–326, 2005.

[31] Pablo Neira-Ayuso, Rafael M Gasca, and Laurent Lefevre. Communicating between the kernel and user-space in linux using netlink sockets. *Software: Practice and Experience*, 40(9):797–810, 2010.

[32] Dan R. Olsen. *Developing User Interfaces (Interactive Technologies).* Morgan Kaufmann, 1998.

[33] Android Open Source Project. Graphics architecture | android open source project. `https://source.android.com/devices/graphics/architecture.html`, 2015. access: 2015-06-10.

[34] Haroon Q Raja. How to change, customize & create android boot animation [guide]. `http://www.addictivetips.com/mobile/how-to-change-customize-create-android-boot-animation-guide/`, 2011. access: 2015-07-25.

[35] Marc Schickler, Rüdiger Pryss, Johannes Schobel, and Manfred Reichert. An engine enabling location-based mobile augmented reality applications. In *Web Information Systems and Technologies - 10th International Conference, WEBIST 2014, Barcelona, Spain, April 3-5, 2014, Revised Selected Papers*, LNBIP. Springer, 2015.

[36] Johannes Schobel, Marc Schickler, Rüdiger Pryss, Fabian Maier, and Manfred Reichert. Towards process-driven mobile data collection applications: Requirements, challenges, lessons learned. In *10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps*, pages 371–382, April 2014.

[37] Johannes Schobel, Marc Schickler, Rüdiger Pryss, Hans Nienhaus, and Manfred Reichert. Using vital sensors in mobile healthcare business applications: Challenges, examples, lessons learned. In *9th Int'l Conference on Web Information Systems and Technologies (WEBIST 2013), Special Session on Business Apps*, pages 509–518, May 2013.

[38] Johannes Schobel, Marc Schickler, Rüdiger Pryss, and Manfred Reichert. Process-driven data collection with smart mobile devices. In *Web Information Systems and Technologies -*

10th International Conference, WEBIST 2014, Barcelona, Spain, Revised Selected Papers, LNBIP. Springer, 2015.

[39] Andreas Schrade. How-to create a working kiosk mode in android. `http://www.andreas-schrade.de/2015/02/16/android-tutorial-how-to-create-a-kiosk-mode-in-android/`, 2015. access: 2015-07-17.

[40] Smieh. Android-system-architecture. `https://commons.wikimedia.org/wiki/File:Android-System-Architecture.svg`, 2012. access: 2015-06-25.

[41] VT. How-to create kiosk mode on the nexus 7. `https://thebitplague.wordpress.com/2013/04/05/kiosk-mode-on-the-nexus-7/`, 2013. access: 2015-07-18.

[42] Brian Ward. *How Linux works : what every superuser should know*. No Starch Press, San Francisco, CA, 2015.

[43] Karim Yaghmour. Embedded android workshop with lollipop.

[44] Karim Yaghmour. *Embedded Android: Porting, Extending, and Customizing*. O'Reilly Media, 2013.

Name: Patryk Boczon   Matrikelnummer: 721828

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .... 20.10.2015, Patryk Boczon ....

Patryk Boczon