Waveform Viewer App for Android

Porting of the Hades Waveform Viewer

Bachelor Thesis

Technical Aspects of Multimodal Systems

Department of Informatics

Faculty of Mathematics, Informatics and Natural Sciences

University of Hamburg

presented by

Lars Lütcke

Student number: 6310463

Field of study: Informatics

First supervisor: Dr. Norman Hendrich

Second supervisor: Dr. Andreas Mäder

Abstract

The purpose of this bachelor thesis was to port the existing waveform viewer *Styx*, belonging to the *HADES simulation framework*, to Android. In this context, the main focus was on the differences between the Java and the Android platform, as well as the down-scaling of the existing user interface, albeit maintaining its functionality.

That being said, this paper mainly describes how the necessary features of a waveform viewer like scrolling, zooming, drag and drop and others were achieved under Android. Other than that, the old *HADES* specific file format, its complications and a possible surrogate are described.

At the end of the paper the current usability of the waveform viewer will be discussed and a small outlook of future possibilities will be given.

Contents

List of Figures

Listings

1 Introduction

1.1 Motivation

On one hand there are the users. The amount of people owning a mobile device capable of running mobile applications is constantly growing. Many are using those devices daily for various activities, including entertainment, communication and shopping. A great source of information on this topic is *Our Mobile Planet* [GooOMP], a tool made available by *Google* as part of their *think with Google* [GooTWG] program.

On the other hand are the mobile devices themselves. Their performance is steadily increasing, allowing more and more sophisticated software to be run. As such the possibilities of performing tasks while being on the road, that are usually done at home on a desktop computer are rising.

It stands to reason, that people generally like to be able to do anything on their mobile device, that they are able to do at home on their desktop computer, assuming the controls and usability are acceptable. The exception proves the rule.

With this information and knowing that there is currently no existing waveform viewer for Android, at least not in the *Google Play Store* and not being able to find anything through search engines as well, the motivation to write the first one seems obvious.

1.2 State of the art

The term waveform viewer henceforth references to the type of waveform viewer, used for viewing signal levels of simulated circuit designs. As such, in the very least, a waveform viewer should graphically provide information on transitions and relations between signals of such a design, over a period of time. Circuit designs are normally written in a *hardware description language* like *VHDL*.

The collection of a signal's transitions is called a waveform, hence the name waveform viewer. A waveform viewer is usually used to view and analyze static waveform data, but can also be used to capture and display waveform data in real time. For example, this static data can be a file whose content is the output of a finished simulation.

A waveform viewer should usually provide more functionality than just displaying the waveforms. This includes, but is not limited to, features like zooming, searching, rearranging the signal order or inspecting *std_logic_vector* signals, where a *std_logic_vector* is an array of *std_logic_1164* signals [93; 1164].

Examples for professional tools are *Custom WaveView* (Synopsys), *Simvision* (Cadence Design Systems) and *GTKWave* [BW]. Figure 1.1 shows what the GUI of *GTKWave* looks like. To the left is an overview of all available signals and next to it a list of the currently displayed ones, as well as their waveforms. In this example, the highlighted *data_port[7:0]* is a *std_logic_vector* with 8 *std_logic_1164* signals. The vector is opened for inspection.

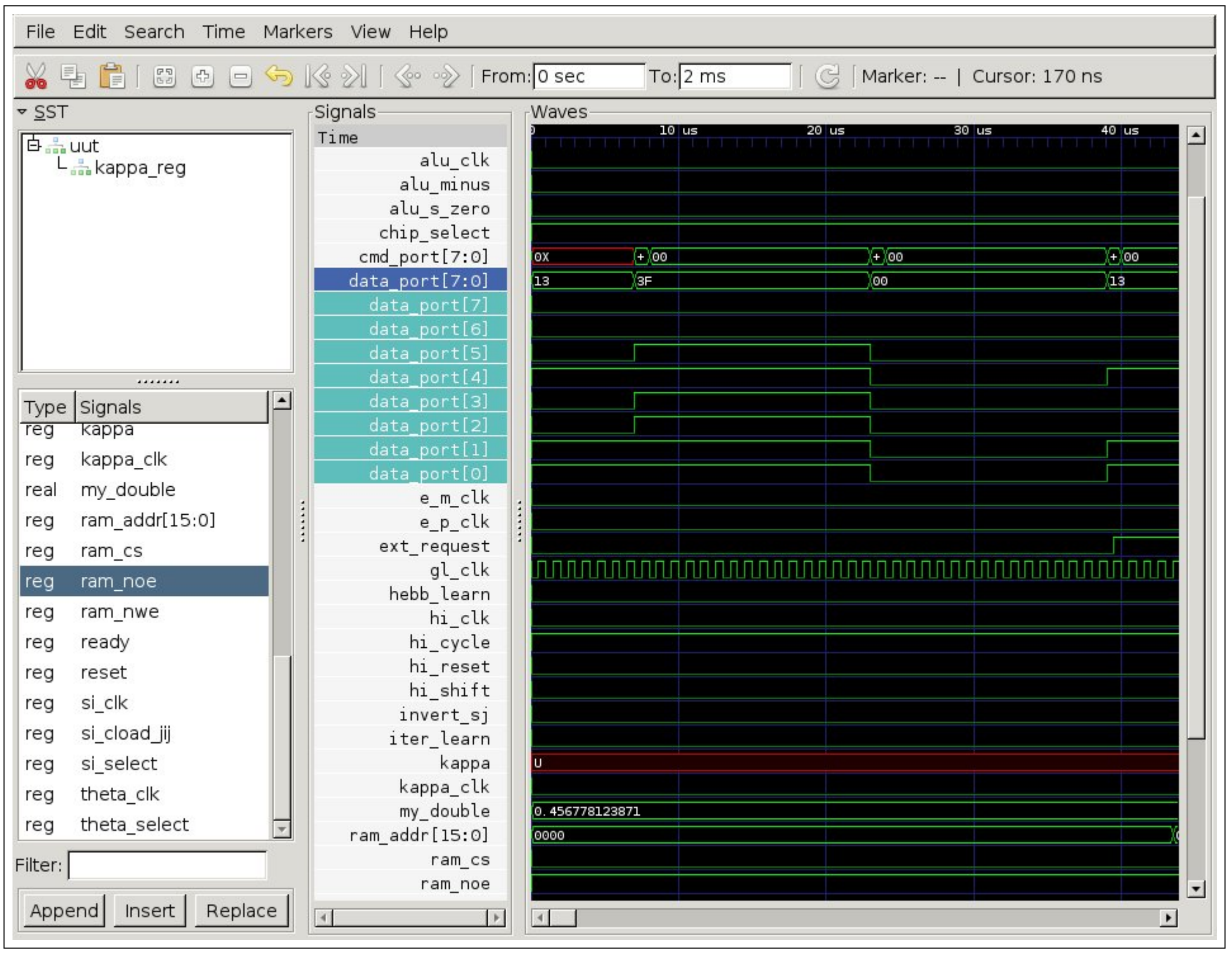


Figure 1.1: Illustration of the waveform viewer *GTKWave*.

1.3 Hades and Styx

HADES or *Hades* is the acronym for the *Hamburg Design System*. It is a framework for *interactive* simulation of circuits. Hades is written in Java and consists of a graphical editor, the simulation engine, libraries of simulation components, and tools like a *waveform viewer* and scripting shell [HaHo].

Figure 1.2 illustrates Hades's GUI on the basis of an example circuit for a 1 bit full adder. At the top is the menu providing many features like zooming, debugging, adding and removing probes to signals and more. By adding a probe to a signal its events will be traced and can later be analyzed with the waveform viewer. Simulation components like I/O elements, gates and wires can be accessed from a right-click menu. Probes can also be added or removed from this menu. At the bottom settings regarding the running time of the simulation can be made and the user is given the currently passed time. The simulation can be stopped, paused, started or resumed and made to run indefinitely.

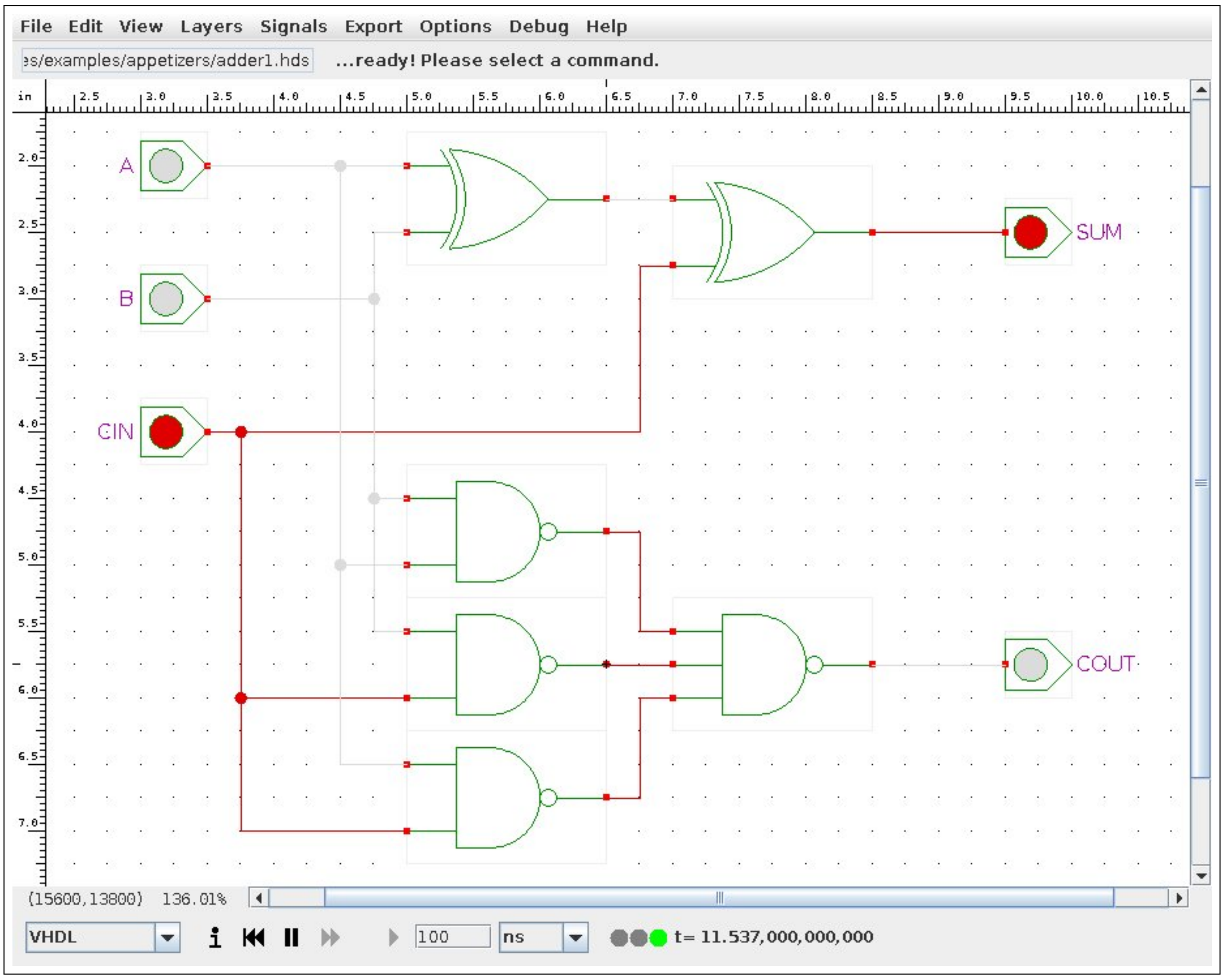

File Edit View Layers Signals Export Options Debug Help
es/examples/appetizers/adder1.hds
...ready! Please select a command.
A
B
CIN
SUM
COUT
(15600,13800) 136.01%
VHDL
100
ns
t= 11.537,000,000,000

Figure 1.2: Illustration of Hades's GUI. The current circuit demonstrates a 1 bit full adder.

In Greek mythology Hades is the name of the god of the underworld, as well as the name of the underworld itself. Following this naming convention the name of the waveform viewer is Styx, which is the name of a female river god and the river that separates the realms of the living and the dead.

The information site about *Styx* itself states, that waveforms play a much lesser role in Hades, since it offers interactive simulation and circuits can be observed in real-time [StyHo]. In this context, *Styx* is a light-weight waveform viewer that offers simple functionality and a simple user interface. Styx is only able to read the *Hades Waveform Viewer* file format.

Figure 1.3 illustrates Styx's GUI on the basis of some waveforms, showing at least one waveform for each supported signal type. At the top is the menu, that offers functionality to open and save files, change the number format of *integer* and *std_logic_vector* values and to update, clear or remove all waveforms. The *Search* item is empty, but under *Help* $\rightarrow$ *Keys* a key map can be found, regarding the features searching, zooming and waveform movement. At the bottom are buttons, that also offer functionality for waveform movement and zooming. Also featured at the bottom is the time and name of the waveform at the current crosshair position and if clicked the waveform's current value and type.

Great features that are not supported by Styx, include the inspection of *std_logic_vector* signals and the ability to only display a subset of signals.

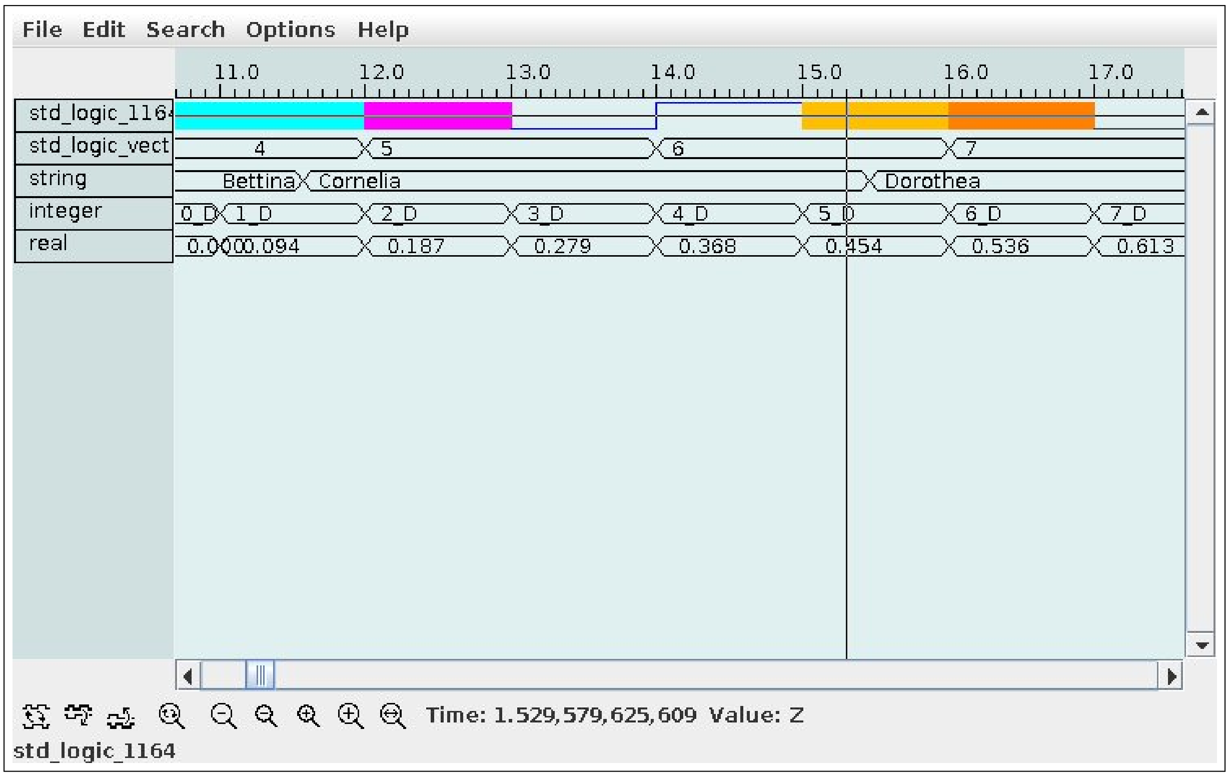


Figure 1.3: Overview of Styx's GUI.

For further information regarding Hades or Styx, consider reading the Hades tutorial [Nor06].

1.4 The Android platform

The most important thing to note is that Android and Java use different platforms. Java is still used as the programming language and in parts of Android's build process, but the Android core libraries do not provide all the functionality that the Java core libraries offer. Support for some other languages exists through the *Android NDK* [DevNDK], although their use is not recommended in most cases.

The Android platform basically consists of the *Android runtime (ART)* and its predecessor *Dalvik*, which execute *Dalvik Executables (*.dex)* containing *Dex bytecode* [DevRun]. Figure 1.4 demonstrates the build process in Android and depicts how an *Android application package (apk)* is attained from the Java source. For a more detailed look at the build process consider visiting Android's *Build System Overview* [DevBSO].

1.4.1 What needs to be considered

As stated before, Android's core libraries do not provide all the functionality that the Java core libraries offer. An example for this is the *java.awt* package. Android only provides support for *java.awt.font* from this package, since drawing is handled differently from Java. Both Java and Android provide information about their APIs in form of online documentations [JavaAPI;

	Java	Android
drawing	java.awt.Graphics –	– android.graphics.Canvas – android.graphics.Paint
handling	keyboard – mouse –	– gestures – (soft) keyboard
memory	depends on OS, VM and – pointers used ~2 GB, 4 GB, 32 GB, 16 EB –	– Android sets a hard limit – around 20 - 100 MB – setting the *largeHeap* flag **can** increase that limit (discouraged)
opening files	javax.swing.JFileChooser –	– not always a built in option – hope a file chooser is installed – or write your own

Table 1.1: Some of the differences between the Android and Java platforms.

DroidAPI]. Table 1.1 gives a quick overview of the main differences that are of importance for us.

A minor difference is how stuff is drawn. Java offers the *java.awt.Graphics* class for this, where a Graphics object is not only used for drawing, but also for setting information about the color to use, font sizes, etc. Android on the other hand separates between these two things. It offers the two classes *android.graphics.Canvas* and *android.graphics.Paint*. The former is used for drawing and the latter one is used to specify the information about color, the style and so on. When drawing, a Paint object must always be given.

Another minor difference would be the ability to browse through the existing file structure and selecting file(s). Java provides the class *javax.swing.JFileChooser* for this, so here it is possible out of the box. Android offers no such class and although the *Storage Access Framework* [DevSAF] was introduced with Android 4.4 (API level 19), there is no guarantee that on a device running an older version a (pre-)installed application for this purpose exists. Since this ability is a requirement for our purposes, we will have to ensure its availability by providing our own file browser.

A bigger difference is the handling. In the original version of Styx this is achieved with the use of a physical mouse and keyboard, but these are often not available on a mobile device, especially the mouse. Instead, they offer a soft keyboard for certain aspects of an application, have a touch screen and are otherwise handled via touch input. This input must then be translated into gestures which are used to identify what action to perform. This is also the main topic of this thesis. The supported gestures, their features and the logic behind them are explained in great detail in chapter 3.

Last to be mentioned is the memory availability. Limits obviously exist for desktop computers as well as for mobile devices, but they are usually much lower in the latter case. In both cases the limits are defined by the overall available RAM. In Java they are furthermore depending on the operating system, virtual machine and the pointers used [JavaVM]. Android however sets a hard limit for each application only depending on the overall RAM available [DevMem]. Tests on a few devices reported limits around 20 MB to 100 MB. Android additionally allows to set a *largeHeap* flag that can increase the limit in some cases, but does not guaranteed to do so and its use is strongly discouraged. The memory availability is of importance for us, because waveform data can grow very fast. This will be explained in more detail in section 2.2.

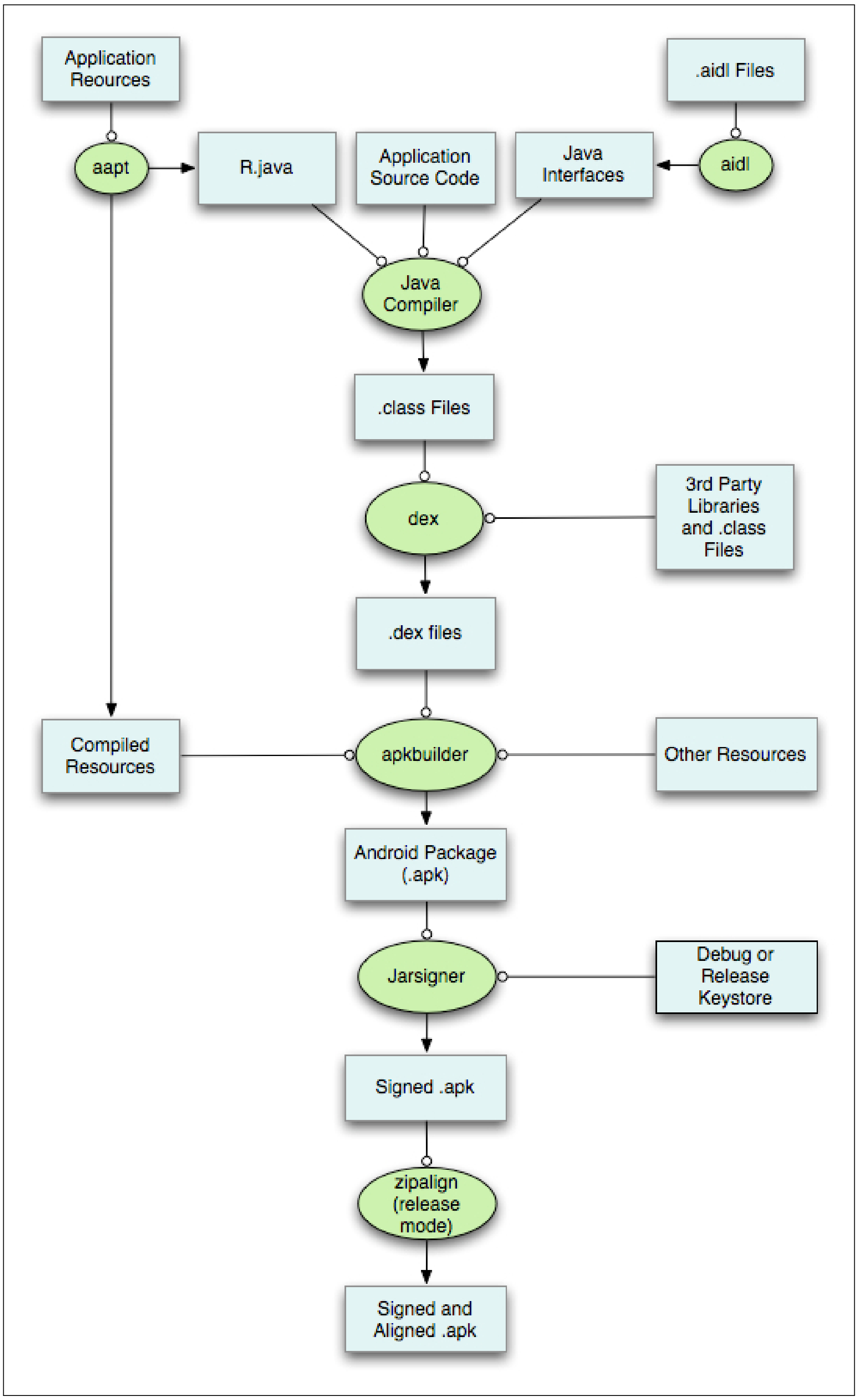


Figure 1.4: Android's build process in a nutshell [DevBSO].

2 The Hades Waveform Viewer file format

After having introduced Hades and Styx and having outlined what we need to consider during the port to Android, we will now explain the *Hades Waveform Viewer (hwv)* file format.

The main waveform class is *hades.styx.Waveform*, all other waveform classes extend this class, as shown in Figure 2.1. The diagram confines itself to the parts needed for reading and writing waveform data. The array reallocation will be shown later in the paper, in form of source code, the rest can generally be disregarded and is therefore not shown. For more information about the Hades API, consider reading the online class documentation [HaAPI]. This documentation can also be downloaded as a zip archive from the homepage.

Listing 2.1 shows the part of the source code, of the Waveform class, that is responsible for the writing of waveform data. It implements the *Serializable* interface which means that it possesses a *serialVersionUID* which is used to identify a class during deserialization. Here it is commented in line 3, because it is not explicitly set to a value and its consequences will be explained in section 2.1. The same holds true for all the subclasses of Waveform. In order to write the waveform data an *ObjectOutputStream* is used to basically just write the important fields, together with some meta information (e.g. the *serialVersionUID*), to a file. So the file format is just a series of serializable objects. No optimizations whatsoever are made before writing to a file.

```
 1  public class Waveform implements Serializable
 2  {
 3      //private static final long serialVersionUID = -1L;
 4
 5      protected double[] times;    // stores event times
 6      protected Object[] events;   // stores event values
 7      protected int size;          // current array sizes
 8      protected int fill;          // how many events there are
 9      protected String name;       // short name
10      protected String fullName;   // full name
11
12      // ...
13
14      private final void writeObject (ObjectOutputStream out)
15          throws IOException
16      {
17          out.writeObject(name);
18          out.writeInt(fill);
19          out.writeObject(fullName);  // new as of 02.07.03
20          out.writeObject(times);
21          out.writeObject(((Object) (events)));
22      }
23  }
```

Listing 2.1: Original source code, showing how waveforms are written to a file in the latest version of Styx.

The reading process works analogue to the writing process. The *serialVersionUID* is used to identify the currently read object and whether a compatible class exists. If no such class exists, a *ClassNotFoundException* is thrown. The only exception to the writing process is the conditional statement, in order to support both an old and a "newer" format.

```java
public class Waveform implements Serializable
{
    // ...

    private final void readObject (ObjectInputStream in)
        throws IOException, ClassNotFoundException
    {
        name = (String) in.readObject();
        fill = in.readInt();

        Object tmp = in.readObject();

        // handle the new format (fullName present)
        if (tmp instanceof String)
        {
            fullName = (String) tmp;
            times = (double[]) in.readObject();
            events = (Object[]) in.readObject();
        }
        // handle the old format (fullName not present)
        else
        {
            times = (double[]) tmp;
            events = (Object[]) in.readObject();
        }
    }
}
```

Listing 2.2: Original source code, showing how waveforms are read from a file in the latest version of Styx.

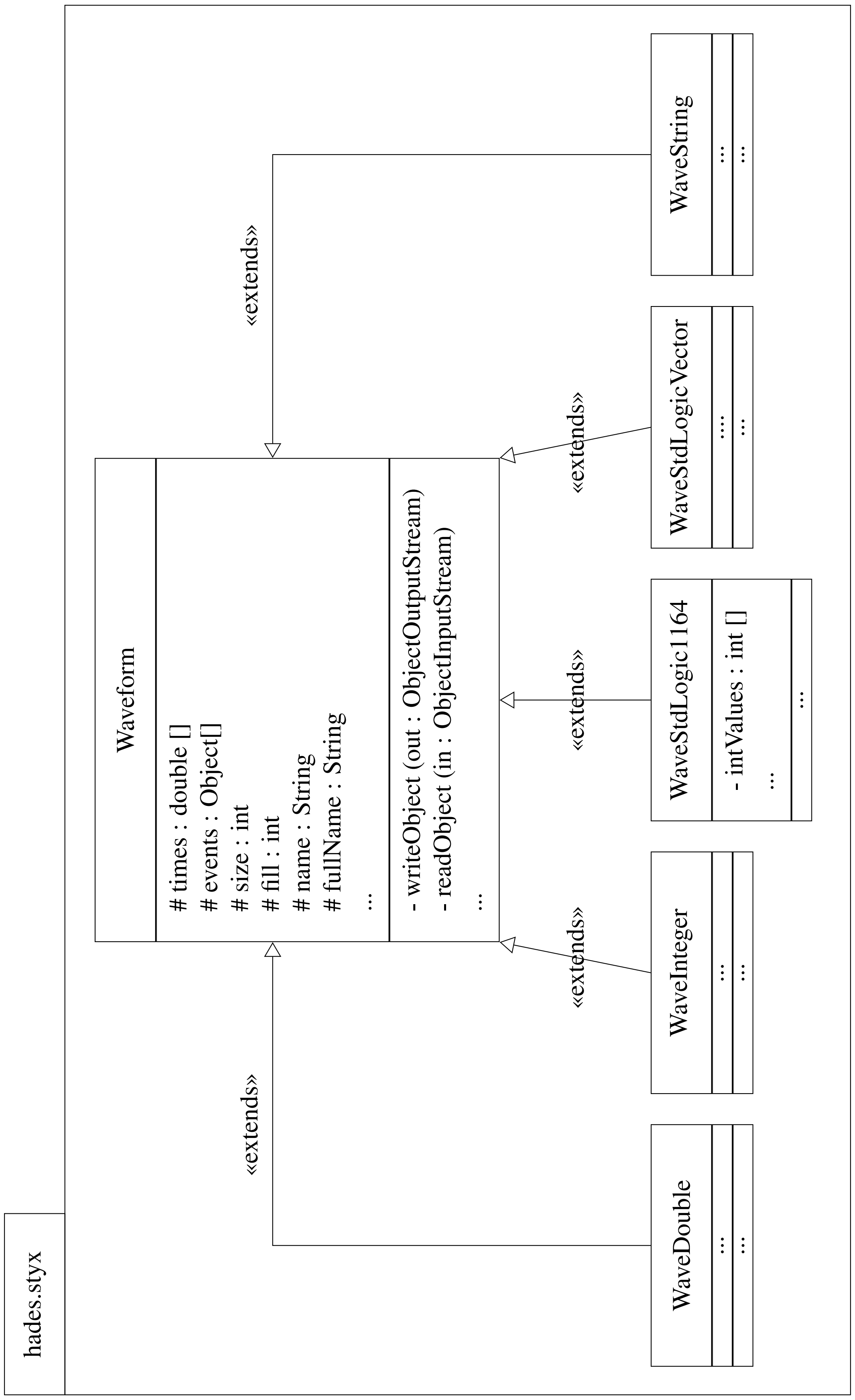


Figure 2.1: A very slimmed down version of the UML class diagram for the *hades.styx* package.

2.1 Compatibility

As previously mentioned, none of the Waveform classes explicitly set the *serialVersionUID*. This causes the serialization runtime to calculate one for us, but this calculation is sensitive to certain class details. Which changes can affect the calculation can be looked up in the *Java(TM) Object Serialization Specification* [JavaOSS]. Different compiler implementations can also cause a different result to be computed, for the same class. Because of these reasons it is strongly advised to manually set a *serialVersionUID* and ensure that changes do not affect the operability.

Now that we know about the role of the *serialVersionUID* and how waveform data is read and written we can talk about the compatibility of files between different Hades versions. Unfortunately, the lack of an explicit *serialVersionUID* is exactly the reason why there are incompatibilities between many different Hades versions, even though the files might actually be compatible in the end. The *hades-applet.jar*, *hades-signed.jar* and the standard *hades.jar*, which are all available from the Hades homepage, are just three examples for this.

Providing support for all versions might possibly still be attained, by writing the process of reading a byte stream of serialized objects from the ground up or by using a custom classloader to try and load the appropriate class for the current version. Rather than going this direction two other approaches will be showcased.

Approach 1 would be to just clean up the classes and remove all dependencies of classes that are not supported under Android. This is quick and easy to implement and there will not even be any changes that have to be made in the Java version. When trying to read an object with an incompatible class, the exception thrown will state the different *serialVersionUIDs*. The correct one will just have to be set explicitly in the new class and it needs to be ensured that the reading order of objects is correct for the current version. By doing this we would again only support one version at a time, but we would have a quick "fix" for our problem. This can then be extended to use the same *serialVersionUID* in all Hades versions, to support all files from this point onward. Older files before this would either have to be migrated or will no longer be supported.

In the second approach the *ObjectOutputStream* will be replaced with a *DataOutputStream*, thereby losing the necessity to implement the *Serializable* interface. Reading and writing to files would largely remain the same, but some changes would need to be made in the Java versions, mainly for writing the arrays that are storing the event data. With this approach there would also be no support for older files or they would have to be migrated first. Ultimately this is the approach that was used. It has been further extended to try and reduce the subsequent memory usage which will be explained in the next section. The final source code showing the writing and reading processes of the new Waveform superclass and exemplary for one if its subclasses can be seen in Listing 2.3, Listing 2.4, Listing 2.5 and Listing 2.6. Since this approach was chosen, a migrater for the currently latest standard *hades.jar* version has been written. The migrater is a simple, but integral tool for this project. Without it we would not have files to use under Android. All this migrater does is, it uses the appropriate class files of the Hades version to read a waveform file, creates a corresponding correct new Waveform object and then adds each event from the old waveform to this new one. After it has finished this process for all waveforms, it creates a new file into which it writes the new waveforms, using the new file format.

The new Waveform class is now abstract, because the *values* variable is now of the *Object* type that will later hold a reference to the actual *values* array of the subclass and not all methods can be implemented here because of that. Whenever possible, shared logic was implemented in this superclass, otherwise an abstract method was declared.

Therefore, the writing process is now split up into a part that the superclass performs and a part that is specific to each subclass. The superclass writes the waveform's names, its amount of events and all the event times. The *writeData()* method is always invoked through a subclass.

```java
public abstract class Waveform
{
    private String fullName;
    private String name;
    protected int size;
    protected int fill;
    protected double[] times;

    // will hold a reference to an array whose type depends
    // on the subclass
    private Object values;

    // ...

    public void writeData (DataOutputStream dos)
        throws IOException
    {
        dos.writeUTF(fullName);
        dos.writeUTF(name);
        dos.writeInt(fill + 1);

        // the values array will be written by the subclass

        // write all event times to the file,
        for (int i = 0; i < fill + 1; i++)
        {
            dos.writeDouble(times[i]);
        }
    }
}
```

Listing 2.3: Source code showing how waveforms are written to a file in the new Waveform superclass.

The subclass's *writeData()* method is always the one that gets invoked externally. Each subclass possesses a *TAG* that is used to identify the type of waveform while reading from a file. It can somewhat be compared to the *serialVersionUID*. At the beginning the subclass writes its *TAG* to the file, it then invokes the *writeData()* method of the superclass and afterwards writes all the event values.

```java
public class WaveInteger extends Waveform
{
    public static final int TAG = 2;
    private int[] values;

    // ...

    public void writeData (DataOutputStream dos)
        throws IOException
    {
        // write the tag for later identification
        dos.writeInt(TAG);

        // let superclass write important data
        super.writeData(dos);

        // write the values array
        for (int i = 0; i < fill + 1; i++)
        {
            dos.writeInt(values[i]);
        }
    }
}
```

Listing 2.4: Source code showing how waveforms are written to a file in the new WaveInteger subclass.

As with the original version, the reading process here is analogue to the writing process as well. The superclass reads the fields in the order they were written. Its *readData()* method is always invoked by a subclass.

```java
public abstract class Waveform
{
    private String fullName;
    private String name;
    protected int fill;
    protected int size;
    protected double[] times;

    // ...

    public void readData (DataOutputStream dis)
        throws IOException
    {
        fullName = dis.readUTF();
        name = dis.readUTF();

        size = dis.readInt();
        fill = size - 1;

        times = new double[size];

        // the values array will be read by the subclass
        for (int i = 0; i < size; i++)
        {
            times[i] = dis.readDouble();
        }
    }
}
```

Listing 2.5: Source code showing how waveforms are read from a file in the new Waveform superclass.

The subclass's *readData()* method is always the one that gets invoked externally. At the beginning the subclass invokes the *readData()* method of the superclass, so that the read order coincides with the write order. Afterwards it reads all the event values. The *TAG* will neither be read in a subclass, nor in the superclass. The *TAG* will be read at some other point and will be used to create a new waveform of the according type. It is a minor detail, but worth being noted in case someone wants to write their own reader for the new file format.

```java
public class WaveInteger extends Waveform
{
    private int[] values;

    // ...

    public void readData (DataOutputStream dis)
        throws IOException
    {
        // let superclass read important data
        super.readData(dis)

        values = new int[size];

        // read the values array
        for (int i = 0; i < size; i++)
        {
            values[i] = dis.readInt();
        }
    }
}
```

Listing 2.6: Source code showing how waveforms are read from a file in the new WaveInteger subclass.

All these read and write operations would also be possible with an *ObjectOutputStream*, but using one would implicate implementing the *Serializable* interface. Since there no longer is a need for this, the change for a *DataOutputStream* occurred.

2.2 Memory usage

As it has already been mentioned, the new classes also aim to reduce the memory usage. To give two related examples, the *ring-oscillator* demo (accessible in Hades via *Help* → *demo* → *ring-oscillator*) has once been run for about 11 ms and once for about 15 ms, probes had been added to all signals. The 11 ms run consisted of about 2200 events and the 15 ms run of about 1091000. The following snippet of some console output illustrates the size differences between the original and the migrated files:

```
->[~] % du -h ring-oscillator-1*
23M ring-oscillator-11ms.hwv
28K ring-oscillator-11ms-migrated.hwv
23M ring-oscillator-15ms.hwv
13M ring-oscillator-15ms-migrated.hwv
```

Granted, since both of the original files have the exact same file size, the simulation was part of the demos and it is one of those examples where the waveform data grows rapidly, it might have been that the initial array sizes were chosen generously to counter often reallocations. These examples still hold some value, since the *ObjectOutputStream* writes objects as they are and the waveforms would take up the 23 MB of memory every time they are viewed. The migration process, and generally the new way of writing data to a file, strips all this unnecessary garbage. This is especially important, since Android sets a pretty tight limit on the memory an application can use. A more appropriate example demonstrating the memory reduction might be the following:

```
->[~] % du -h paper-example*
12K paper-example.hwv
8.0K paper-example-migrated.hwv
```

The *paper-example.hwv* file is a file containing waveforms that are pretty much completely filled with events. The waveforms were already shown in Figure 1.3. After migrating the file it is still roughly 33% smaller. This can partly be attributed to the use of *primitive* arrays instead of their *Object* counterparts. The amount of saved memory depends from case to case. As of yet, some small tests have not identified a case where a file is bigger after a migration.

In order to be able to use *primitive* arrays instead of their *Object* wrappers, some changes in the code are needed, the most important ones being in the constructor and the array reallocation.

Listing 2.7 shows the original source code. In the constructor a *double* array and an *Object* array are created to store event times and values. If the *reallocate()* method is triggered, while trying to add a new event, it is first tried to simply create a new array of twice the size and copying the values into the new array. If this fails, the existing array will be reused by overwriting the first half of the events with the last half, and then reusing the last half for new events.

```java
public class Waveform
{
    protected int size;
    protected double[] times;
    protected Object[] events;
    protected int fill;

    // ...

    public Waveform ()
    {
        size = 10;
        times = new double[size];
        events = new Object[size];

        fill = 0;
        times[0] = 0.0;
        events[0] = "";
        fullName = "no full name set";
    }

    public void reallocate()
    {
        try
        {
            int newsize = 2 * size;
            double[] newtimes = new double[newsize];
            Object[] newevents = new Object[newsize];

            for(int i = 0; i < size; i++) // copy old values
            {
                newtimes[i] = times[i];
                newevents[i] = events[i];
            }

            times = newtimes;
            events = newevents;
            size = newsize;
        }
        catch(OutOfMemoryError outofmemoryerror)
        {
            int half = fill / 2;

            for(int i = 1; i < half; i++) // move last to first halve
            {
                times[i] = times[i + half];
                events[i] = events[i + half];
            }

            fill = half - 1;
        }
    }
}
```

Listing 2.7: Original constructor and reallocation method in the Waveform class.

In the original program everything is handled by the Waveform superclass. The new approach is a bit different. The Waveform superclass defines the method *newValuesArray()*, which is used in the constructor and during the array reallocation. Instead of trying to create a *values* array itself, it delegates this task to the subclasses. Other changes that have been made are the introduction of a lower maximum capacity for arrays and the use of *System.arraycopy()* instead of loops, to copy arrays or parts thereof. Other than that the classes are pretty much identical on this part. The source code for the new superclass can be seen in Listing 2.8.

The maximum capacity is still experimental and is there to try and further reduce memory usage under Android. Setting a lower limit obviously has some drawbacks. Only a look at the *ring-oscillator* example from before is needed, where the overall event count already exceeded 1000000 events, after only 15 ms. Because of this, support to turn this feature off or at least the ability to modify the limit should be added, as it does not yet exist. But since the Android version only supports reading waveform data from a file for now, the whole data will always be read anyway and the limitation is not important. It is relevant in case the Waveform classes in the original Styx version should be replaced by these.

```
public abstract class Waveform
{
    private static final int START_CAPACITY = 10;
    private static final int MAX_CAPACITY = 1000000;
    protected double[] times;
    private Object values;
    protected int size;
    protected int fill;
    private String fullName;
```

```java
    private String name;

    // ...

    public Waveform (String fullName, String name)
    {
        this.fullName = fullName;
        this.name = name;

        times = new double[START_CAPACITY];
        values = newValuesArray(START_CAPACITY);

        fill = 0;
        size = START_CAPACITY;
    }

    protected abstract Object newValuesArray (int arraySize);

    protected void reallocate ()
    {
        try
        {
            if (size == MAX_CAPACITY) // experimental
            {
                throw new OutOfMemoryError();
            }

            int newSize = Math.min(2 * size, MAX_CAPACITY);
            double[] newTimes = new double[newSize];
            Object newValues = newValuesArray(newSize);

            // use System.arraycopy instead of a loop
            System.arraycopy(times, 0, newTimes, 0, size);
            System.arraycopy(values, 0, newValues, 0, size);

            times = newTimes;
            values = newValues;
            size = newSize;
        }
        catch (OutOfMemoryError e)
        {
            int half = size / 2;
            System.arraycopy(times, (size + 1) / 2, times, 1, half);
            System.arraycopy(values, (size + 1) / 2, values, 1, half);

            fill = half;
        }
    }
}
```

Listing 2.8: Constructor and array reallocation in the new Waveform superclass.

A subclass's *newValuesArray()* method simply returns a new array of the appropriate type with the given *arraySize*. In the constructor, as well as the *reallocate()* method, the reference to the new *values* array will be obtained by using the *super.getValues()* method. The corresponding source code is depicted in Listing 2.9.

```java
public class WaveInteger extends Waveform
{
    private int[] values;

    // ...

    public WaveInteger (String fullName, String name)
    {
        super(fullName, name);

        // get the reference
        values = (int[]) super.getValues();
        values[0] = 0;
    }

    @Override
    public int[] newValuesArray (int arraySize)
    {
        return new int[arraySize];
    }

    @Override
    protected void reallocate ()
    {
        super.reallocate();
        values = (int[]) super.getValues();
    }
}
```

Listing 2.9: Source code for the constructor and reallocation method in the new WaveformInteger subclass.

Two further possibilities to reduce the memory usage, that have not been implemented, will be briefly addressed now. In order to use either or both of these methods, greater changes would need to be made regarding operations like adding and finding an event or accessing an event's value. The higher complexity of the algorithms would also increase their access time and computation costs. These are examples for a time-memory tradeoff.

By looking at Listing 2.1 again, it is easily understandable that a *double* array to store event times will be created for each waveform. This can be cause for redundancy, because often times many waveforms have events at the same time. Instead of storing the event times in a central place, each waveform stores this information for themselves.

Memory usage to store *std_logic_1164* event values can be reduced by packing them. Internally an *integer* is used to store one of its values, but since they can only take the values U, X, 0, 1, Z, W, L, H and D, only requiring 4 bits to represent all possible values, an *integer* can be used to actually store 8 *std_logic_1164* event values.

2.3 Surrogate format

The new surrogate for the old Hades Waveform Viewer file format offers even more features. The first 8 bytes and the last 8 bytes represent the magic number for *.hwv files. The magic number is simply the string "HWV-File" or as a number in hexadecimal format "0x4857562d46494c45".

A file's first and last bytes should be checked against this number to identify whether the file to read is actually using the new file format and therefore supported. After the first magic number, the amount of waveforms, stored in the file, follows. This number is immediately followed by the last byte position of each waveform inside the file. This information can be used to skip waveforms while reading waveform data. After this meta information, blocks of waveform data follow. Each block begins with a *TAG*, followed by the full name and then the name of the waveform. After this, the amount of events is stored. Afterwards all event times and lastly all the event values follow. Figure 2.2 illustrates this, once more, graphically.

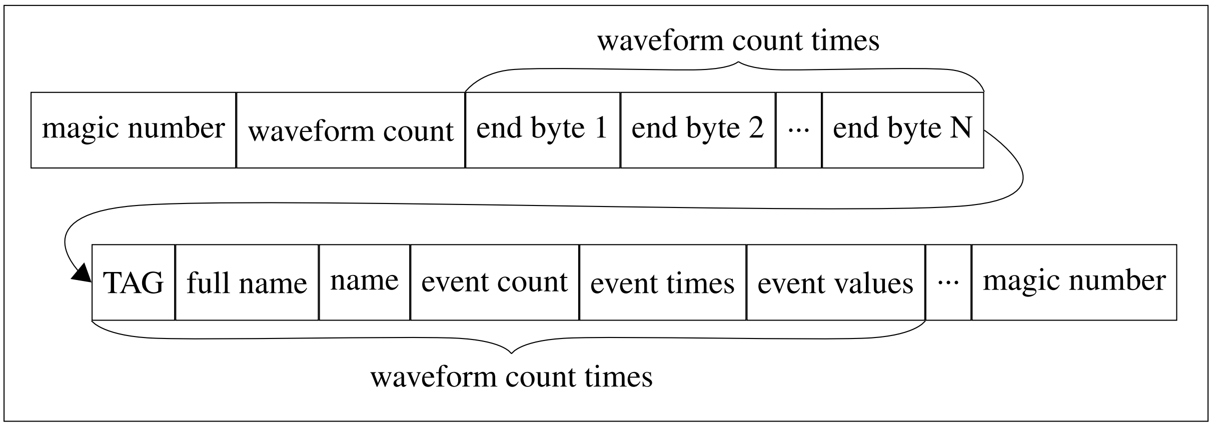


Figure 2.2: Surrogate for the old Hades Waveform Viewer file format.

For convenience the *styx.io.StyxFileReader* and *styx.io.StyxFileWriter* exist.

A StyxFileReader can be used to read files with the new format. It performs checks to make sure that the new format is actually used, before attempting to read the file. It offers the two methods *loadWaveforms()* and *readWaveforms()*. The former is used to load waveform information regarding the actual waveform count, all their names and all their approximated sizes. The latter is used to either read all waveforms, or a subset thereof. The *loadWaveforms()* method needs to have been called at least once before making calls to *readWaveforms()*. Furthermore, the class offers methods to retrieve the waveform count, the names and the approximated sizes.

A StyxFileWriter can be used to write waveform data to a file, using the new file format. It offers the method *writeToFile()* for this purpose. When calling this method, the waveform data to write needs to be given.

A diagram depicting the class hierarchy for the new Waveform classes is given in Figure 2.3. It is very similar to the one shown in Figure 2.1. It shows the type for the *values* array of each Waveform type. Notice the *StdLogicVectorDummy* array for *WaveStdLogicVector* waveforms. This class replaces the *styx.models.StdLogicVector* class, which is originally used for *hades.styx.WaveStdLogicVector* values. Do keep in mind that some methods also changed their access modifier, or got their logic changed and that a few new methods were introduced. All this is not shown, to keep the diagram clear.

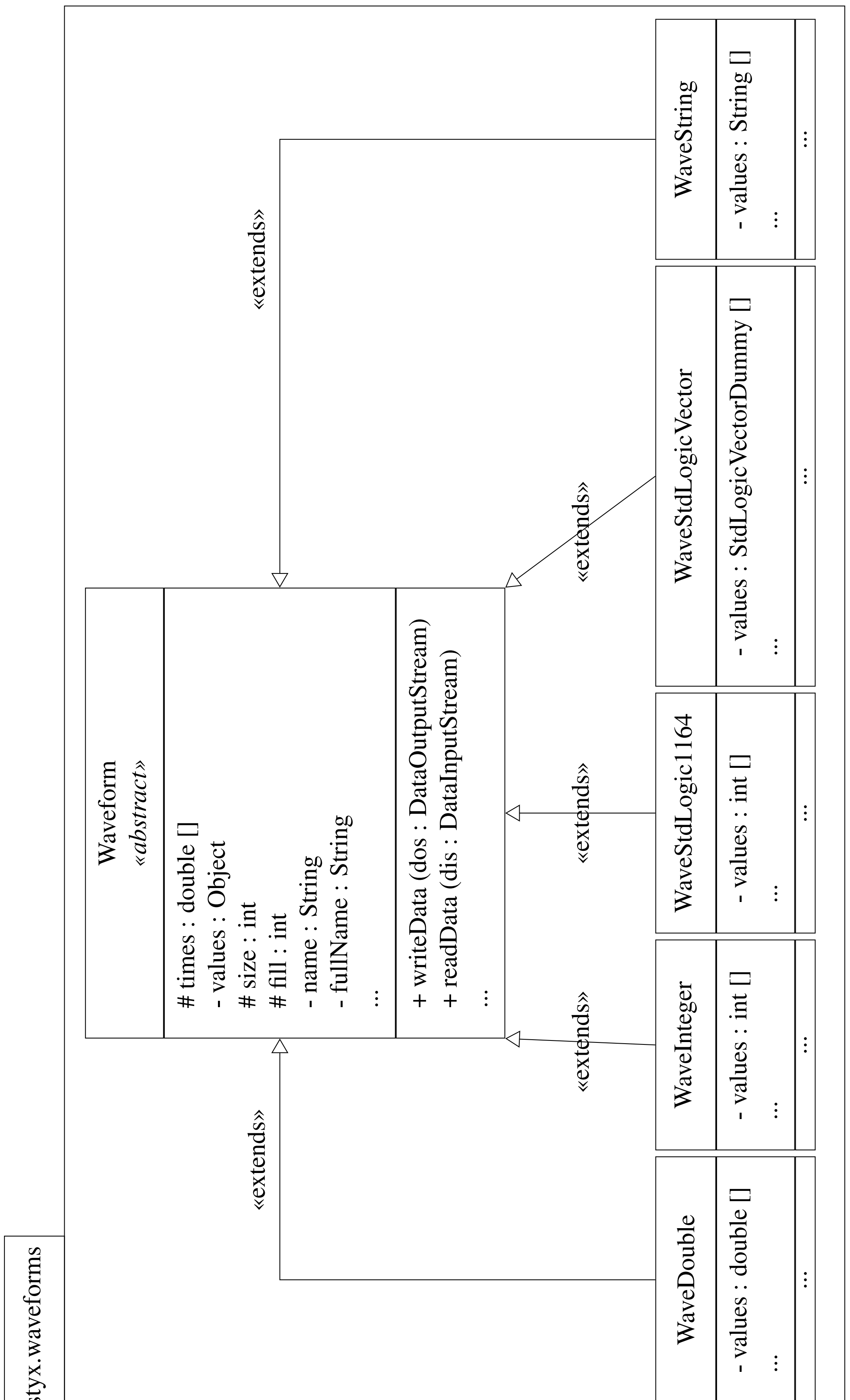


Figure 2.3: A very slimmed down version of the UML class diagram for the new *styx.waveforms* package.

3 User manual

This chapter addresses the main topic of this thesis. It lists all of the features that are supported, explains what they do and how to use them and additionally offers source code regarding their implementation. Because the code examples can get very long, some of them will be found in Appendix A. All the images supplied in this chapter were taken on a smartphone, with a diagonal screen size of only 4.5 inches and a resolution of 1280x720 pixels. Devices with higher screen resolutions are able to show more than just 5 waveforms, in either orientation.

3.1 Gesture detection

In order to be able to handle gestures performed by the user, the touch input needs to be intercepted. The *android.view.View* class is the root for all classes, used to visually present data on the screen. When implementing a custom view, this class or a more appropriate one of its subclasses, should be extended. This View class offers the *onTouchEvent()* method, that is called if a touch has been registered in the bounds of the view. A very low level way of translating touch input and detecting gestures is to overwrite this method and implement the needed logic oneself. Listing 3.1 shows a skeleton for this purpose. The *onTouchEvent()* method accepts an argument of the *MotionEvent* type. This motion event contains information related to a touch event, like the position on screen, the action performed, the time of the event and so on. This event must then be correctly interpreted, and if possible, be translated into a gesture. It must then be ensured that the correct action is performed, based on the detected gesture.

```java
public class MyView extends View
{
    // ...

    @Override
    public boolean onTouchEvent (MotionEvent event)
    {
        switch (event.getActionMasked())
        {
            case MotionEvent.ACTION_DOWN: // a pointer went down
                break;
            case MotionEvent.ACTION_MOVE: // a pointer moved
                break;
            case MotionEvent.ACTION_UP:   // a pointer went up
                break;
            // case ...
        }
        return true;
    }
}
```

Listing 3.1: How to handle touch input. This skeleton can be extended to detect gestures oneself.

When extending or creating a new *ViewGroup*, the *onInterceptTouchEvent()* method needs to be considered as well, but this is not case for the given problem.

For convenient and basic gesture detection, Android provides the two gesture detection classes *ScaleGestureDetector* and *GestureDetector*, in conjunction with the four gesture listeners. Two of those are *OnScaleGestureListener* and *OnGestureListener*. The other two are just variants of the first two, returning *false* for every event. They are equal to the implementation of the first two listeners, as depicted in Listing 8.1.

Listing 8.1 furthermore shows what type of gestures are detected, and how the detectors can be created and used. The creation happens in one of the view's constructors. To use the detectors, the current motion event received in the *onTouchEvent()* method, will be passed on to them. In case one of the supported gestures is detected, the appropriate method is invoked, for example the *onScroll()* method if the user dragged his finger across the screen.

Android's provided gesture detection is sufficient for the need of this project. For anyone wanting to support custom gestures beyond that scape, the first method is definitely the way to go.

3.2 Starting the application

When starting the application, the user is greeted with an empty waveform viewer. This start screen is shown in Figure 3.1. The only thing possible to interact with, at this point, is the menu in the *Action Bar*. Opening up the menu reveals 6 items. Only the items *Open File*, *Zoom fit*, *Smart zoom fit*, *Number Format* and *Settings* are enabled for now. The zoom fit and smart zoom fit items will not do anything, since no waveforms are visible, but the number format can already be used to set the number format for *integer* and *std_log_vector* values. Clicking on the *Settings* item opens the settings window and allows the user to make some adjustments, but this will be discussed in section 3.11. So for now we will start by opening a new file and dive right into section 3.3 about file picking.

3.3 File picking

It was already explained that there is no guarantee for the existence of a file picker/browser on Android devices. *Styx Android* comes with its own small file picker, for this reason. This file picker offers only the bear necessities of moving between directories, providing information about the current directory path, canceling the process of picking a file and actually picking a file. The file picker overlays the current view like a dialog, as can be seen in Figure 3.2. Moving up a directory, is achieved by clicking on the folder with the "**..**" label, but anyone who has at least once seen or worked with an Unix-like system will most likely know this already. When clicking on any other directory, the file browser will try to move into that directory. If this is not possible, the user will be notified about this. When clicking on a file, that file will be picked and the waveform viewer will try to read it. If reading the file is not possible for whatever reason, the user will be notified as well. Clicking the "Cancel" button at the bottom will stop this process. Once a valid file has been selected, the application will move over to the selection of which waveforms to read and display from that file.

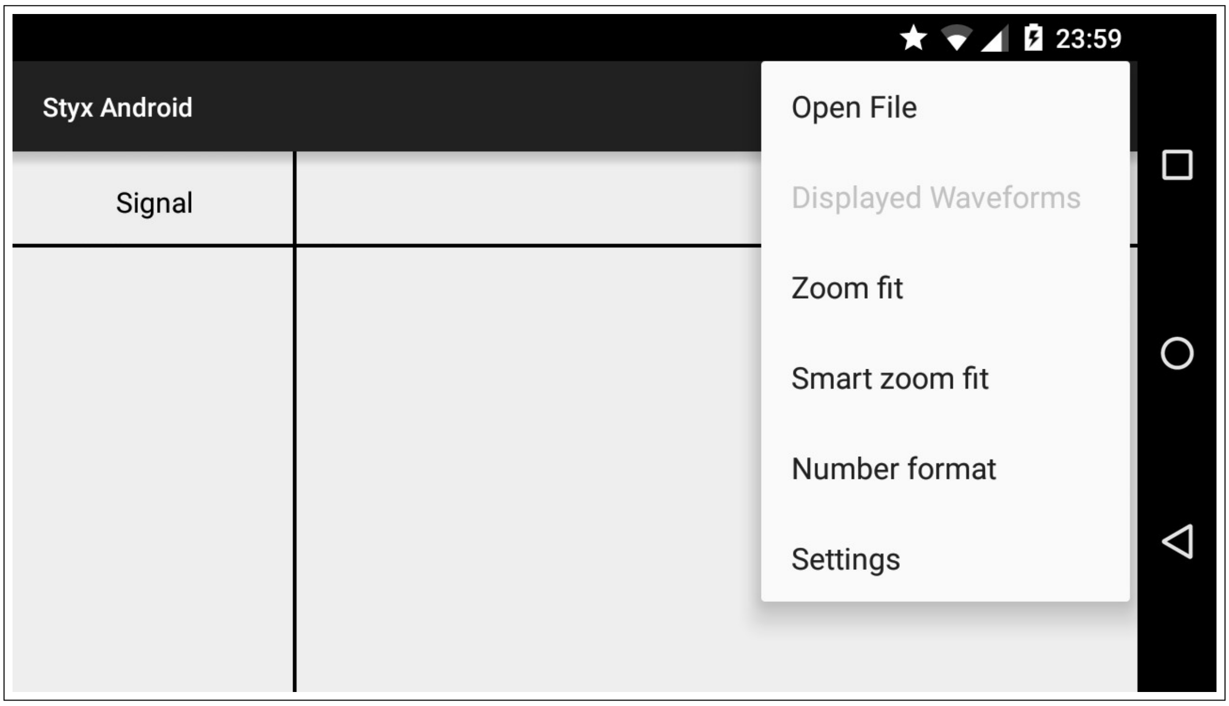


Figure 3.1: The application's start screen, with an open menu in the action bar.

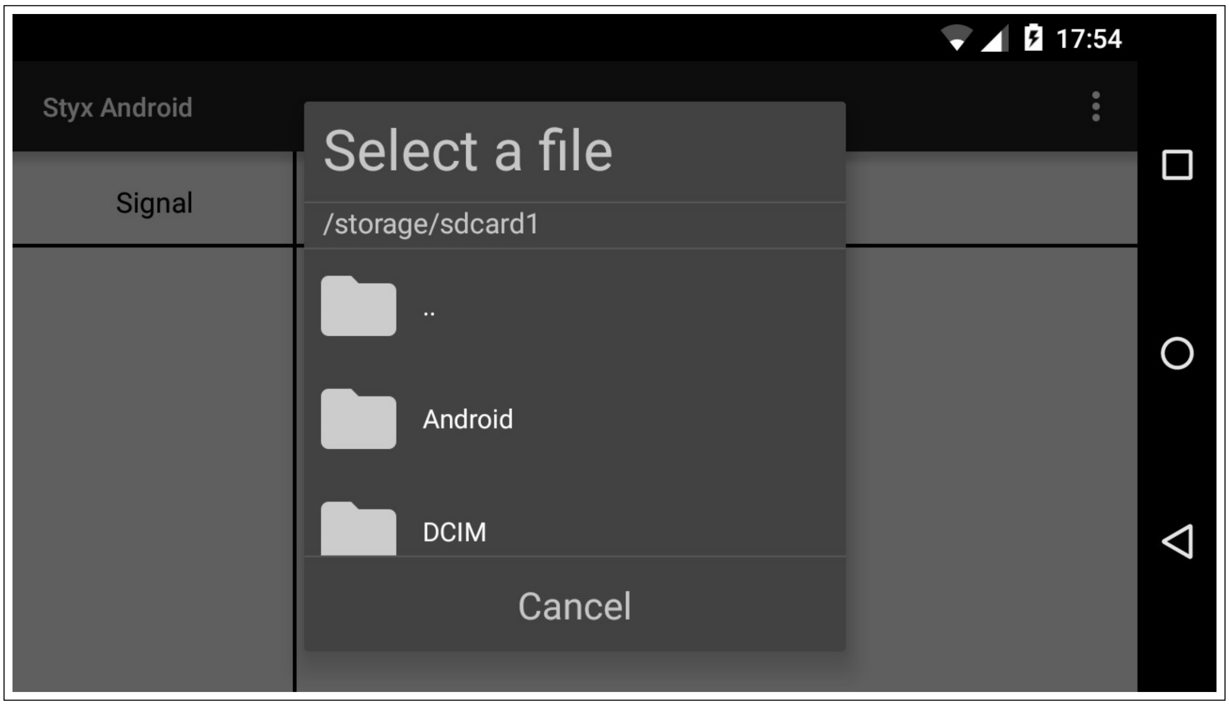


Figure 3.2: The file browser supplied with the waveform viewer.

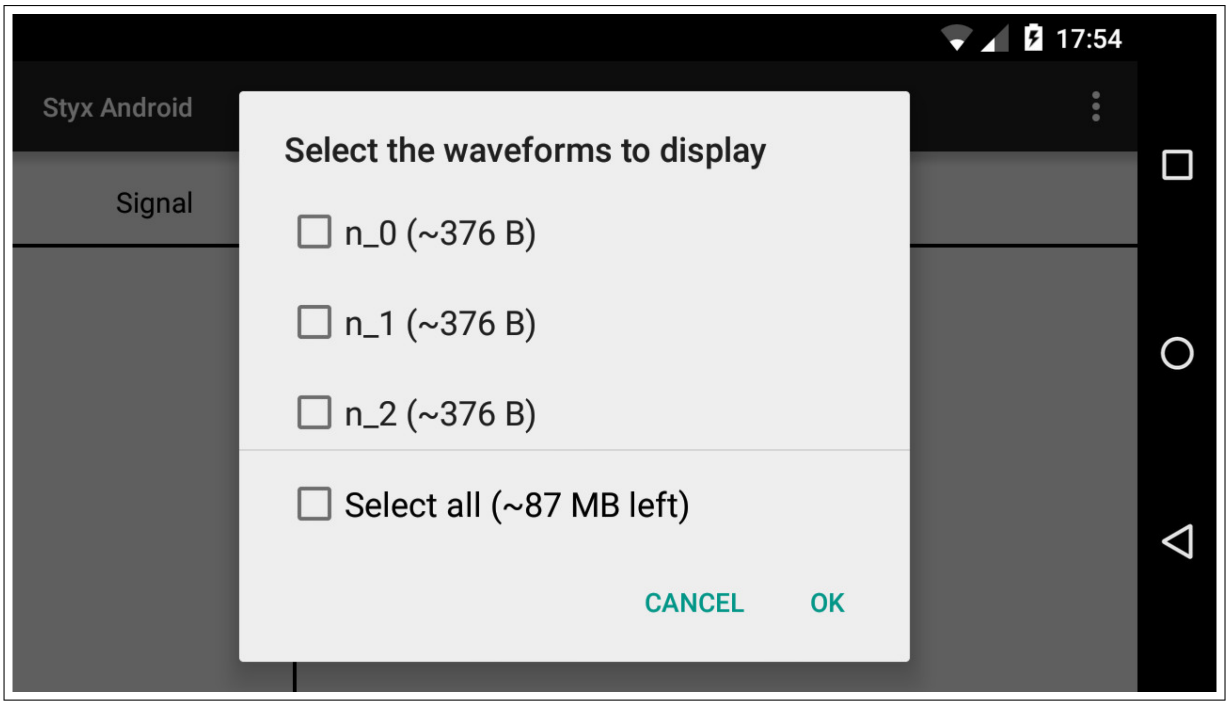


Figure 3.3: Dialog to select which waveforms to read and display.

3.4 Waveform display selection

The user has the possibility to select which waveforms to read and display. To help the user decide during this process, the application will not only display the waveform names, but also an approximation of their sizes and the remaining available memory for the application. This is by no means an accurate result, but it is enough to let the user know in case the memory limit might be about to be reached. What this selection dialog looks like is shown in Figure 3.3. Changes in the selections will only be applied if they are confirmed by clicking the “OK” button, in all other cases will the changes be discarded. If at least one waveform is selected for display, the necessary waveforms will be read from the file. The displayed data looks very similar to that of the Java version, and some example data is illustrated in Figure 3.4. If the user wishes to change the selections at a later point in time, the dialog can be reopened via the *Displayed Waveforms* item from the menu. This item will become enabled, as soon as a valid waveform file is picked.

3.5 Scrolling

Android labels the possibility of scrolling in both the x and y axes as *panning*, whereas scrolling itself defines the general act of moving a viewport. So this chapter should more specifically be called *Panning*, but *Scrolling* was chosen because this term is more familiar to most users and includes panning as well. The application supports two types of scrolling, these two being dragging and flinging. When a user reaches an end of the viewport while scrolling, it will be indicated by an edge glow, see Figure 3.5 for an example.

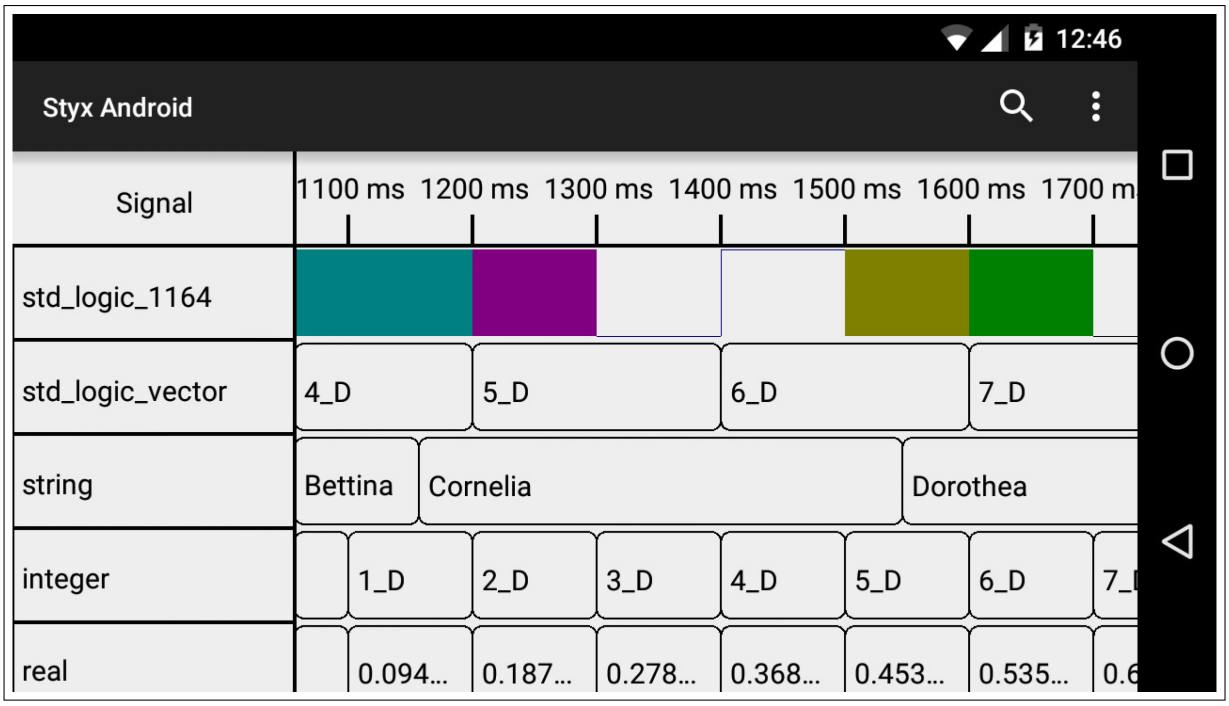


Figure 3.4: The waveform viewer displaying some example data.

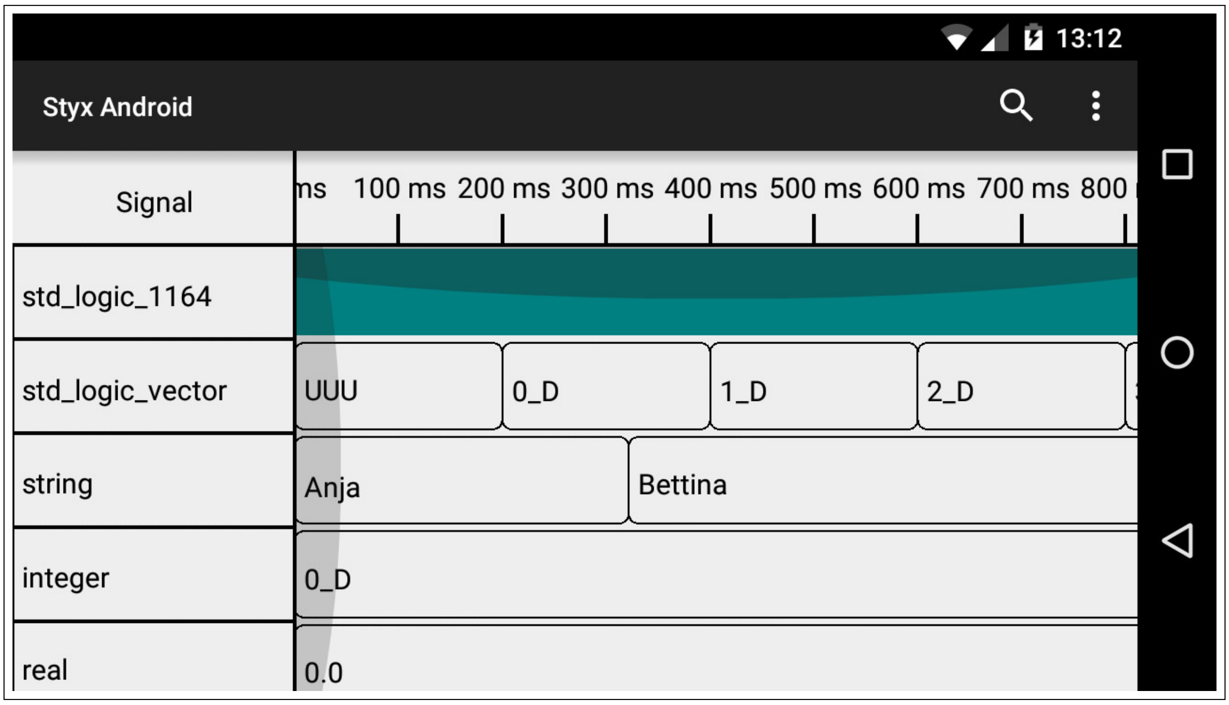


Figure 3.5: An example for the edge glows. In this case the user reached the left and the top edge while dragging the viewport.

3.5.1 Dragging

As the name implies, *Dragging* describes the type of scrolling by dragging ones finger across the touch screen. If the gesture detector detects this, the *onScroll()* method is invoked. The source code showing how the gesture is handled can be seen in Listing 8.2.

First it is checked whether the user is currently selecting an area for the *selection zoom* feature and cancels it if necessary. It then adds the currently detected travel distance on each axis to the appropriate counter. The reason for using these counters is to also support scrolling on only one axis.

Afterwards another check is performed to determine where the pointer went down before any scrolling was detected. If the pointer went down in the area dedicated for the waveform names, one more check needs to be done. This check is used to verify if a drag and drop operation is currently active and makes some adjustments for this operation if necessary. The method then returns prematurely to disable any other kind of scrolling. If no drag and drop operation is active at least the horizontal scrolling will be disabled. Horizontal scrolling should only be available if the pointer went down in the area used to draw the waveform events.

After having determined on which axes scrolling is allowed, it is checked whether the user is already scrolling or not. In case the user is not scrolling it is determined on which axis the longer distance was traveled and scrolling is initiated on it. If on the other hand the user is already scrolling on exactly one axis, a check is performed to see whether scrolling on the second axis should be initiated as well.

Once these checks are done the actual scrolling will be performed. This is done by adjusting a variable used to store the current scroll position. The counter needs to be adjusted as well, and a flag is set to indicate that scrolling has actually happened.

The last part is only executed if the viewport needs adjusting. This is done to reduce calls to *postInvalidateOnAnimation()*, which will update the screen. Other checks are also performed to see if an edge of the viewport was reached, and whether the user needs to be notified about this. What this can look like can bee seen in Figure 3.5.

3.5.2 Flinging

Flinging starts out like dragging and is then initiated by quickly lifting the finger during the drag. The viewport will then keep on scrolling without further touch input. The scroll speed decreases over time, until it finally stops moving. If the gesture detector detects a fling, the *onFling()* method is invoked. Flinging is split up into three parts, the source codes can be found in Listing 3.2, Listing 3.3 and Listing 8.3.

The source code for the *onFling()* method can be found in Listing 3.2. In this method a check is first performed to determine whether a drag and drop operation is in action. Flinging is only available when this is not the case. Upon availability, another check is performed to determine where the pointer went down, likewise to the test for the dragging operation. If the finger went down in the area dedicated for the waveform names, the possibility of flinging will be disabled. As with the dragging, a horizontal fling should only be allowed if the pointer went down in the area used to draw the waveform events.

When horizontal flinging is allowed, further checks are performed to determine whether the user wants to fling on only one axis or both. In either case will the fling be delegated to the *fling()* method.

```java
@Override
public boolean onFling (MotionEvent e1, MotionEvent e2, float
    velocityX, float velocityY)
{
    // only allow flinging if we are not currently performing a drag
    // and drop operation
    if (!mDragAndDropActive)
    {
        if (hitTestNames(e1.getX(), e1.getY()))
        {
            //only allow flinging vertically
            fling(0, (int) -velocityY);
        }
        else if (e1.getX() > mContentRect.left)
        {
            if (mFlingFactor > 0 &&
                Math.abs(velocityX) > mFlingFactor *
                    Math.abs(velocityY))
            {
                fling((int) -velocityX, 0);
            }
            else if (mFlingFactor > 0 &&
                     Math.abs(velocityY) > mFlingFactor *
                         Math.abs(velocityX))
            {
                fling(0, (int) -velocityY);
            }
            else
            {
                fling((int) -velocityX, (int) -velocityY);
            }
        }
    }

    return true;
}
```

Listing 3.2: How flinging is handled.

The source code for the *fling()* method can be found in Listing 3.3. This method takes two arguments, the horizontal and the vertical fling velocity. This method performs further checks on how to execute the fling.

The reason for these checks will now be explained. Due to the fact that the actual width for a canvas needed, to draw all waveform events at a certain zoom level, can be extremely large, its size cannot always be represented by an *integer* value. For this reason the horizontal scroll position represented by a *long*. This will be explained in more detail in section 3.6.

Another information needed to explain the reason and to understand the source code, is that *mScroller* is an *OverScroller* object. This class is provided by Android and can be used to calculate individual scroll positions during a fling animation. An *OverScroller*'s *fling()* method takes *integer* values for all arguments. The complete signature can be seen in lines 3-6 of Listing 3.3.

The reason for these checks is basically that the current horizontal scroll position cannot always be directly used with the *OverScroller* object. In order to bypass this problem and still be able to use the *OverScroller*, three cases will be distinguished. They are discerned in this method as opposed to the *onFling()* method, in order to reduce the amount of conditional statements.

In the first case the user wishes to fling to the left, while he can reach the left edge of the viewport with a single fling. This is called a *careful left fling*. In this case we can safely use the current horizontal scroll position as the argument for *startX*. The corresponding flag will be set.

In second case the user wishes to fling to the right, while he can reach the right edge of the viewport with a single fling. This is respectively called a *careful right fling*. In this case 0 is used for the horizontal start position and the maximum position is calculated based on the current width and scroll position. The corresponding flag will be set and the scroll position at the start of the fling will also be remembered, because of the explained problem.

The last case is if neither a careful left nor a careful right fling needs to be performed. In this case 0 will be used for the horizontal start position as well. For the horizontal minimum and maximum scroll positions, the maximum fling amount and its negative value can safely be used. The scroll position at the start of the fling will be remembered.

The vertical fling does not pose this problem, because an insane amount of waveforms would be needed. It is just assumed that this will never be the case. The calculation will now be delegated to the *OverScroller* object.

```
 1  private void fling (int velocityX, int velocityY)
 2  {
 3      // mScroller.fling(int startX, int startY,
 4      //                 int velocityX, int velocityY,
 5      //                 int minX, int maxX,
 6      //                 int minY, int maxY,
 7      //                 int overScrollX, int overscrollY)
 8
 9      //if we have do to a careful left fling
10      if (velocityX < 0 && mScrollX <= mMaxFlingVelocity)
11      {
12          mCarefulLeftFling = true;
13          mScroller.fling((int) mScrollX, mScrollY,
14                          velocityX, velocityY,
15                          0, mMaxFlingVelocity,
16                          0, mSizeY - mContentRect.height(),
17                          mContentRect.width() / 2,
18                          mContentRect.height() / 2);
19      }
20      //if we have do to a careful right fling
21      else if (velocityX > 0 && (mSizeX - mScrollX) <= mMaxFlingVelocity)
22      {
23          mFlingStartX = mScrollX;
24          mCarefulRightFling = true;
25          mScroller.fling(0, mScrollY,
26                          velocityX, velocityY,
27                          0, (int) (mSizeX - mScrollX -
                               mContentRect.width()),
                        0, mSizeY - mContentRect.height(),
                        mContentRect.width() / 2,
                        mContentRect.height() / 2);
    }
    //all other flings
    else
    {
        mFlingStartX = mScrollX;
        mScroller.fling(0, mScrollY,
                        velocityX, velocityY,
                        -mMaxFlingVelocity, mMaxFlingVelocity,
                        0, mSizeY - mContentRect.height(),
                        mContentRect.width() / 2,
                        mContentRect.height() / 2);
    }

    postInvalidateOnAnimation();
}
```

Listing 3.3: Source code for the *fling()* method.

An *OverScroller* object will make calls the *computeScroll()* method of the view it belongs to, while calculating scroll positions during a fling. By overwriting this method, the actual fling can be performed. The source code for this method can be found in Listing 8.3.

A the beginning a check is performed, to figure out whether the *OverScroller* can calculate a new scroll position. This is necessary because this method can also get invoked in other situations. One of those cases is when the *GestureDetector* detected dragging, so it needs to be ensured that a fling animation is in process.

First the flags for a careful left and careful right fling will be examined. If either one is set, it is also checked whether the horizontal scroll position is out of the viewport's bound, meaning if one of its edges has been overstepped. The scroll position will then be adjusted to the correct bound, and the user will be notified about this through the use of an edge glow. If the new position is still inbound, the new scroll position will just be set. Except for the careful left fling, the correct scroll position will be calculated by using the scroll position we saved at the beginning of the fling.

The latter steps are then also applied to adjust the vertical scroll position.

3.6 Zooming

If at least one waveform has been selected for display, the maximum possible viewport width will be determined. This value will depend on the screen resolution and the simulation time. The maximum width for a viewport with a minimum width of 720 pixels and a simulation time of 5 seconds will be $720 \cdot 5 \cdot 10^{15} = 3{,}6 \cdot 10^{18}$ pixels. This obviously exceeds the limits of an *integer* by far, but is still in the range of a *long*. For this reason *long* values are used to store information about the current viewport width and horizontal scroll position. In this example zooming is possible up to the femtosecond scale. The use of *long* values unfortunately also means that after a certain point zooming will only be possible unto higher timescales. Only horizontal zooming is possible as of now.

3.6.1 Pinch zoom

Pinch zooming describes the type of zooming, that happens when the user has two pointers on the touch screen and drags at least one of them. In newer versions this is also the case if the user drags a finger across the screen right after a double tap. This feature has been disabled, because double taps will be used to initiate other actions.

When the scale gesture detector detects the beginning of a zoom, the *onScaleBegin()* method will be called. The source code can be found in Listing 3.4.

This method will stop the process of selecting an area for the *selection zoom* feature, if necessary. It will then save the current distance between the two pointers and the relative horizontal focus between them, based on the current scroll position and viewport width. From now on the *onScale()* method will be called.

```
@Override
public boolean onScaleBegin (ScaleGestureDetector detector)
{
    //stop zoom selection
    if (mSelectingArea)
    {
        mSelectingArea = false;
    }

    //set current span x
    lastSpanX = detector.getCurrentSpanX();

    //set relative focus x position
    relativeFocusX = (mScrollX + (double) detector.getFocusX() -
        mContentRect.left) / mSizeX;

    return true;
}
```

Listing 3.4: Initiation of a pinch zoom.

Whenever *onScale()* gets invoked, the new viewport width will be calculated based on the current and old distance between the two pointers. If the new width is out of bounds, it will be adjusted accordingly. Afterwards the new timescale, and the interval at which marks in the time bar should drawn, will also be recalculated. It is then tried to center the visible viewport around the relative horizontal focus. This enables the user to zoom around a specific time. The last remembered distance between the two pointers will then be replaced by the current one, for use in the next call. The corresponding source code can be found in Listing 3.5.

```
@Override
public boolean onScale (ScaleGestureDetector detector)
{
    //calculate new x size
    curSpanX = detector.getCurrentSpanX();
    mSizeX *= curSpanX / lastSpanX;

    //adjust size, if necessary
    if (mSizeX > mMaxSizeX)
    {
        mSizeX = mMaxSizeX;
    }
    else if (mSizeX < mContentRect.width())
    {
        mSizeX = mContentRect.width();
    }

    //calculate new time scale
    mTimeScale = (int) Math.floor(Math.log10(
            mOverallMaxTime * mContentRect.width() / mSizeX));
    mTimeMarkInterval = mSizeX / mOverallMaxTime * Math.pow(10,
        mTimeScale);

    //adjust scroll to center the relative focus
    mScrollX = (long) (relativeFocusX * mSizeX) -
        (mContentRect.width() / 2);

    if (mScrollX < 0)
    {
        mScrollX = 0;
    }
    else if (mScrollX > mSizeX - mContentRect.width())
    {
        mScrollX = mSizeX - mContentRect.width();
    }

    //update span x
    lastSpanX = curSpanX;

    postInvalidateOnAnimation();

    return true;
}
```

Listing 3.5: Source code for the actual pinch zoom.

3.6.2 Selection zoom

Selection zooming describes the type of zooming, that lets the user first select an area which is then zoomed into. Here it is initiated by a double tap on the area used to draw the waveform events. The area can then be selected by dragging the pointer. The selected area will be visualized by a gray overlay, for reference see Figure 3.6. Upon release, the zoom into the selected area will be performed.

A double tap will invoke *onDoubleTap()*. The method's source code is shown in Listing 3.6. The method checks, whether the double tap occurred in the correct area. If that is the case, the method sets the two horizontal position values, as well as the flag that is used to indicate whether a *selection zoom* is in process. The method will furthermore disable the ability of a long press. This is done because even if the user drags the pointer across the screen after a double tap, a long press will be triggered when its time threshold has been exceeded. Whether this behavior is intended or a bug is unknown. Events after a double tap will trigger the *onDoubleTapEvent()* method. From here on out the *selection zoom* process will be handled by that method.

```java
        @Override
        public boolean onDoubleTap (MotionEvent e)
        {
            //disable long press
            mGestureDetector.setIsLongpressEnabled(false);

            if (e.getX() > mContentRect.left)
            {
                //set x coordinates
                mSelectZoomX1 = e.getX();
                mSelectZoomX2 = mSelectZoomX1;

                //set flag for drawing
                mSelectingArea = true;
            }

            return true;
        }
```

Listing 3.6: Source code for the initiation of a selection zoom.

The code in *onDoubleTapEvent()* will only be executed if a selection zoom has been initiated beforehand. The action of the received event is then determined and used to select what action to perform. If the user dragged the pointer the selected area will be adjusted. It is ensured that this value stays within its given bounds. In case the user lifts the pointer or a *MotionEvent.ACTION_CANCEL* is received the zoom into the selected area will be performed. This does not include the cases where the selection zoom is ended in the *onScroll()* and *onFling()* methods. The actual zooming is delegated to the *zoomSelectedArea()* method and its invocation depends on the two position values. The long press will be re-enabled and the flag indicating an active selection zoom will be reset. Listing 3.7 shows the corresponding source code.

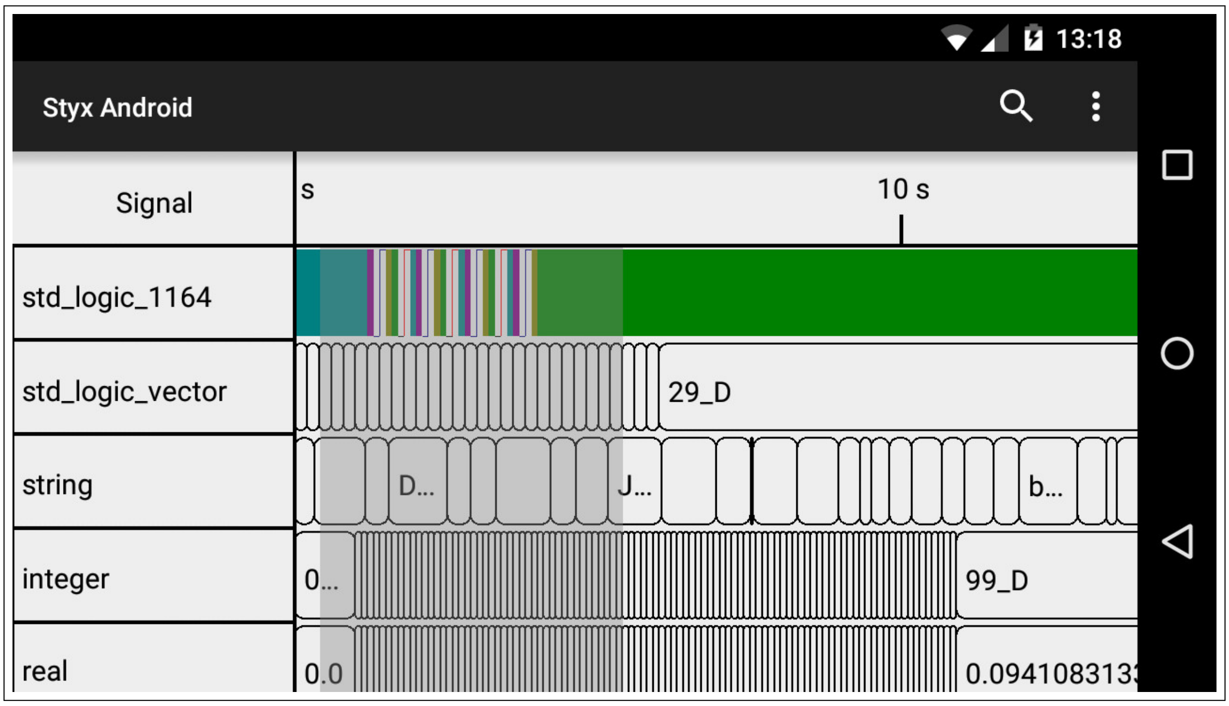


Figure 3.6: Image showing a currently selected area for use during the selection zoom.

```java
@Override
public boolean onDoubleTapEvent (MotionEvent e)
{
    //only execute if the double tap was on the time bar
    //or the wave canvas
    if (mSelectingArea)
    {
        switch (e.getActionMasked())
        {
            case MotionEvent.ACTION_MOVE:
                //set second x coordinate
                mSelectZoomX2 = e.getX();
                if (mSelectZoomX2 < mContentRect.left)
                {
                    mSelectZoomX2 = mContentRect.left;
                }
                else if (mSelectZoomX2 > mContentRect.right)
                {
                    mSelectZoomX2 = mContentRect.right;
                }
                break;
            case MotionEvent.ACTION_UP:
            case MotionEvent.ACTION_CANCEL:
                //perform the zoom
                if (mSelectZoomX1 < mSelectZoomX2)
                {
                    zoomSelectedArea(mScrollX + (long) (mSelectZoomX1
                        - mContentRect.left));
                }
                else
                {
                    zoomSelectedArea(mScrollX + (long) (mSelectZoomX2
                        - mContentRect.left));
                }
                //re-enable long press
                mGestureDetector.setIsLongpressEnabled(true);
                //reset flag
                mSelectingArea = false;
                break;
        }

        postInvalidateOnAnimation();

        return true;
    }

    return false;
}
```

Listing 3.7: Source code for the area selection of a selection zoom.

The *zoomSelectedArea()* method takes one argument. This should be the absolute left position of the selected area to zoom into and is needed to set the correct scroll position after the zoom. It is ensured that an actual area has been selected in order to avoid divisions by 0. The zoom factor will be calculated based on the width of the selected area. This zoom factor is then used to determine the new width of the viewport. The timescale and interval at which to draw marks in the time bar will have to be adjusted as well.

```java
private void zoomSelectedArea (long absoluteX1)
{
    if (Math.abs(mSelectZoomX2 - mSelectZoomX1) > 0)
    {
        float zoomFactor = mContentRect.width() /
            (Math.abs(mSelectZoomX2 - mSelectZoomX1));
        double relativeLeft = absoluteX1 / (double) mSizeX;

        // calculate new x size
        mSizeX *= zoomFactor;

        // adjust size, if necessary
        if (mSizeX > mMaxSizeX)
        {
            mSizeX = mMaxSizeX;
        }

        // calculate new time scale
        mTimeScale = (int) Math.floor(Math.log10(
                mOverallMaxTime * mContentRect.width() / mSizeX));
        mTimeMarkInterval = mSizeX / mOverallMaxTime * Math.pow(10,
            mTimeScale);

        // adjust scroll to the left time of the selected area
        mScrollX = (long) (relativeLeft * mSizeX);
    }
}
```

Listing 3.8: Source code used to perform the actual zoom into a selected area.

3.6.3 Zoom fit

The *zoom fit* feature is only available from the menu in the action bar. It will zoom out so all events will fit on the screen. The use of this feature is not advised, because even a reasonably small amount of events can already be cause for long drawing times, which can freeze up the GUI. The use of the *smart zoom fit* feature is recommended instead.

3.6.4 Smart zoom fit

The *smart zoom fit* feature is accessed via the menu in the action bar. By default it is also used to select a feasible timescale after selecting waveforms to display, but this behavior can be disabled in the settings. The main purpose is to find a timescale on which only a reasonable amount of events need to be drawn. The algorithm is by no means perfect. It will happen more often than not that the zoom will be onto a deeper timescale than needed. The use of this feature is advised nonetheless. The source code for the algorithm is given in Listing 8.4.

First the current relative left position of the viewport will be calculated, so the scroll position can be restored after the zoom. If the smart zoom factor has not yet been calculated, it will be done now. This is done by calculating all intervals between events. The mean interval time will then be determined and used to identify whether zooming onto a lower timescale is needed and will cause the zoom factor to be calculated. This zoom factor will then be used to adjust the viewport width. Afterwards the timescale and time mark interval will need to be updated as well and at the end the scroll position will be restored.

3.7 Drag and drop

Waveforms can be rearranged using the drag and drop feature. For this to work at least one waveform needs to be selected, then performing a long press on the name canvas will initiate the drag and drop of the selected waveforms. A waveform can be selected by performing a single tap on its name. The current insert position is visualized by a gray shadow in the name canvas. The code used to initiate the drag can be seen in Listing 8.5, the code for moving the drag and drop items on the screen has already been seen in Listing 8.2. What a drag and drop in action looks like, is illustrated in Figure 3.7

During the initiation the ability for a long press will be disabled because the motion event will have to be resend to enable dragging the items across the screen. The drag and drop will only be initiated when the long press occurred on the name canvas. All the selected waveforms will be moved to an array, that only exists for this purpose. All the unselected waveforms will thereafter be moved up in the original array. The height of the viewport will then be adjusted.

The end of the drag and drop operation is performed when the user lifts the pointer and will need to be performed in the *onTouchEvent()* method, because the gesture detector does not offer support for this. The source code can be found in Listing 3.9.

When ending a drag and drop, long presses will be re-enabled. The insert position will be calculated and all waveforms after this position will be moved down. Then the waveforms will be dropped, the height of the viewport adjusted and the flag that indicates an active drag and drop will be reset.

```java
@Override
public boolean onTouchEvent (MotionEvent event)
{
    if (mWaveforms != null)
    {
        // ...

        //handle when the uses lifts the finger
        int actionMasked = event.getActionMasked();
        if (actionMasked == MotionEvent.ACTION_UP || actionMasked ==
            MotionEvent.ACTION_CANCEL)
        {
            //enable long presses
            mGestureDetector.setIsLongpressEnabled(true);

            //stop drag and drop if necessary
            if (mDragAndDropActive)
            {
                //current insert position
                int insertPosition = (int) Math.min(
                        (mScrollY + mDragPosY - mContentRect.top) /
                           mWaveformHeight,
                        mWaveforms.length - mSelectedWaveformsCount);

                //move waveforms after insert position down
                for (int i = mWaveforms.length - 1;
                     i > (insertPosition + mSelectedWaveformsCount -
                        1);
                     i--)
                {
                    mWaveformsOrder[i] = mWaveformsOrder[i -
                       mSelectedWaveformsCount];
                    mSelectedWaveforms[i] = false;
                }

                //drop waveforms at position
                for (int i = 0; i < mSelectedWaveformsCount; i++)
                {
                    mWaveformsOrder[i + insertPosition] =
                       mDraggedWaveforms[i];
                    mSelectedWaveforms[i + insertPosition] = true;
                }

                //adjust mSizeY
                mSizeY = Math.max(mContentRect.height(),
                   mWaveforms.length * mWaveformHeight);

                //reset drag and drop flag
                mDragAndDropActive = false;

                postInvalidateOnAnimation();
            }
        }
    }

    return handled;
}
```

Listing 3.9: Source code for dropping waveforms after a drag and drop.

3.8 Searching

The *Search* feature is accessed via the magnifier symbol in the action bar. While the search is active, an overlay exists, that is used to determine the direction to search in. Searches will only be performed on selected waveforms. During the search no distinction between numbers and strings will be made. For example, if a search for "15" is performed, *std_logic_vector* and *integer* values as well as *double* and *String* values starting with "15" will be searched for. To search for any event, the wildcard "*" should be used. What an active search looks like is depicted in Figure 3.8. If a search yields a match, a cursor will be drawn at the found position and it will be tried to center that position in the viewport. When searching for a *std_logic_1164* value, no distinction between lowercase letters and uppercase letters will be made. If a cursor is set, the search will be performed beginning from its position.

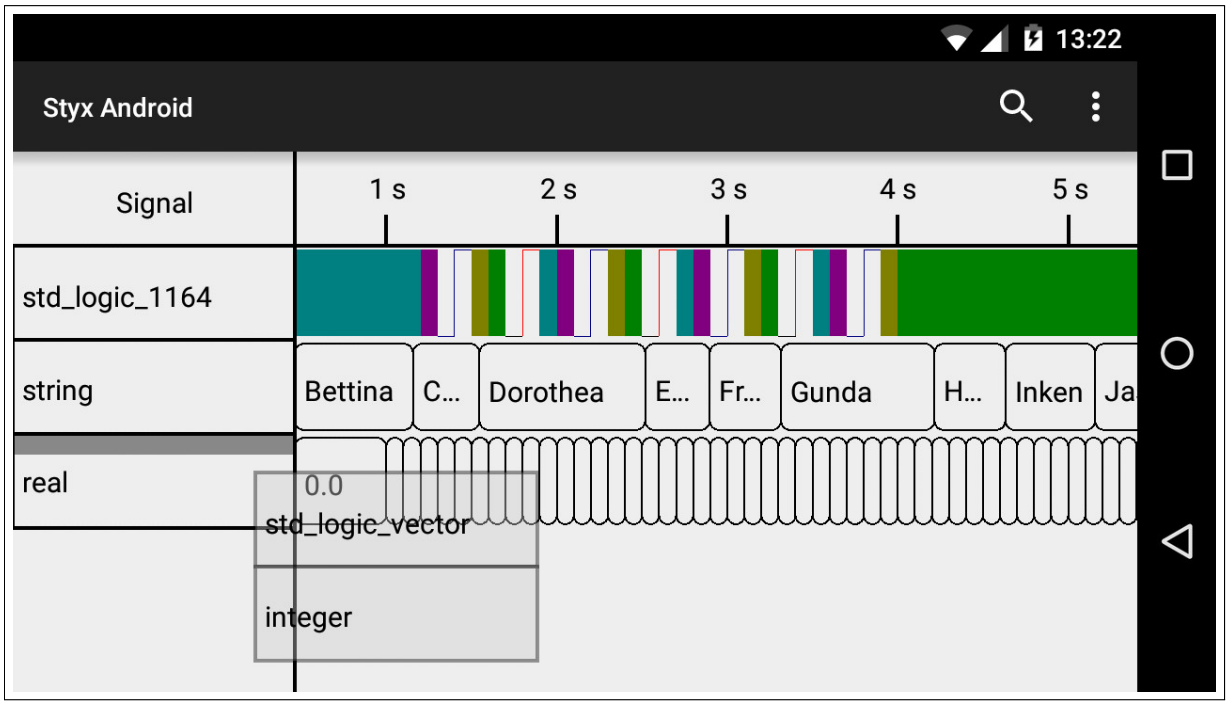


Figure 3.7: Illustration of a drag and drop operation.

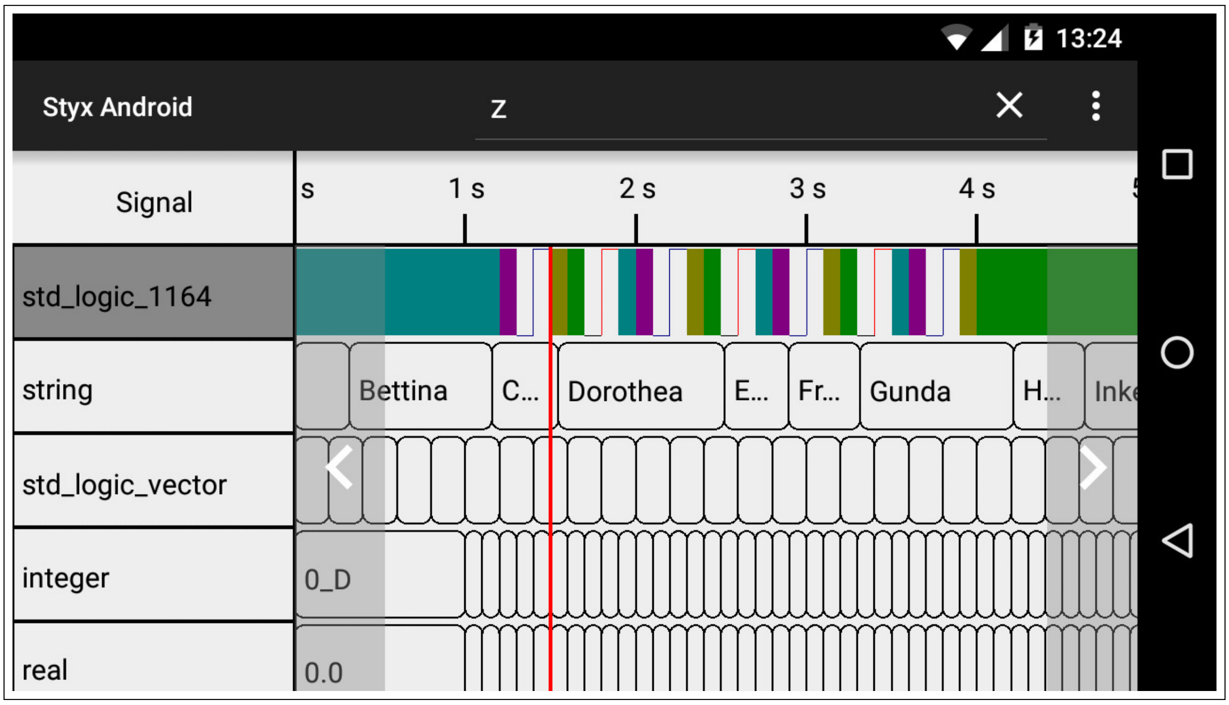
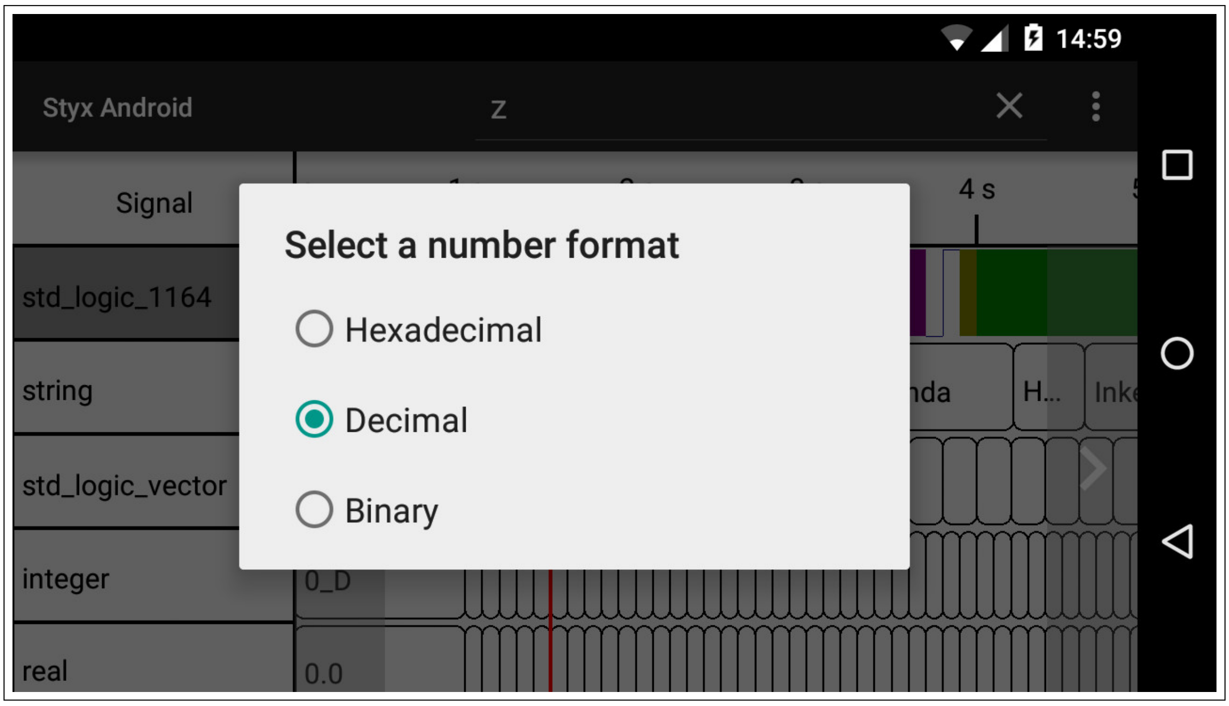


Figure 3.8: Illustration of an active search.



Select a number format

Hexadecimal [✓ Decimal] Binary

Figure 3.9: Illustration of the dialog, used to change the number format.

3.9 Number format

As previously mentioned, the user can change the number format that is used for *integer* and *std_logic_1164* values. This can be achieved via the item *Number format* in the action bar menu. The user can choose between *hexadecimal*, *decimal* and *binary*. The decimal format will be used by default. Changes will persist, as long as the application is running. On the next start it is reset to the default decimal format. What the dialog to pick a format looks like can be seen in Figure 3.9.

3.10 Cursor

The user can manually set a cursor that can be used to jump to, by doing a single tap anywhere on the wave canvas. Keep in mind that this cursor and the cursor used during a search are shared. Jumping to the cursor is performed, when the user executes a long press anywhere on the wave canvas. What the cursor looks like, can be seen in Figure 3.8 as a vertical red line.

3.11 Settings

The user has the ability to make some adjustments, more appropriate for their taste. These adjustments can be done in the settings window, reached via the *Settings* item in the action bar menu. The user is able to change colors used for *std_logic_1164* values, the threshold used to initiate scrolling on a second axis, the threshold used to initiate flinging on a seconds axis and whether or not to use the *smart zoom fit* feature. The last point only applies to the automatic use

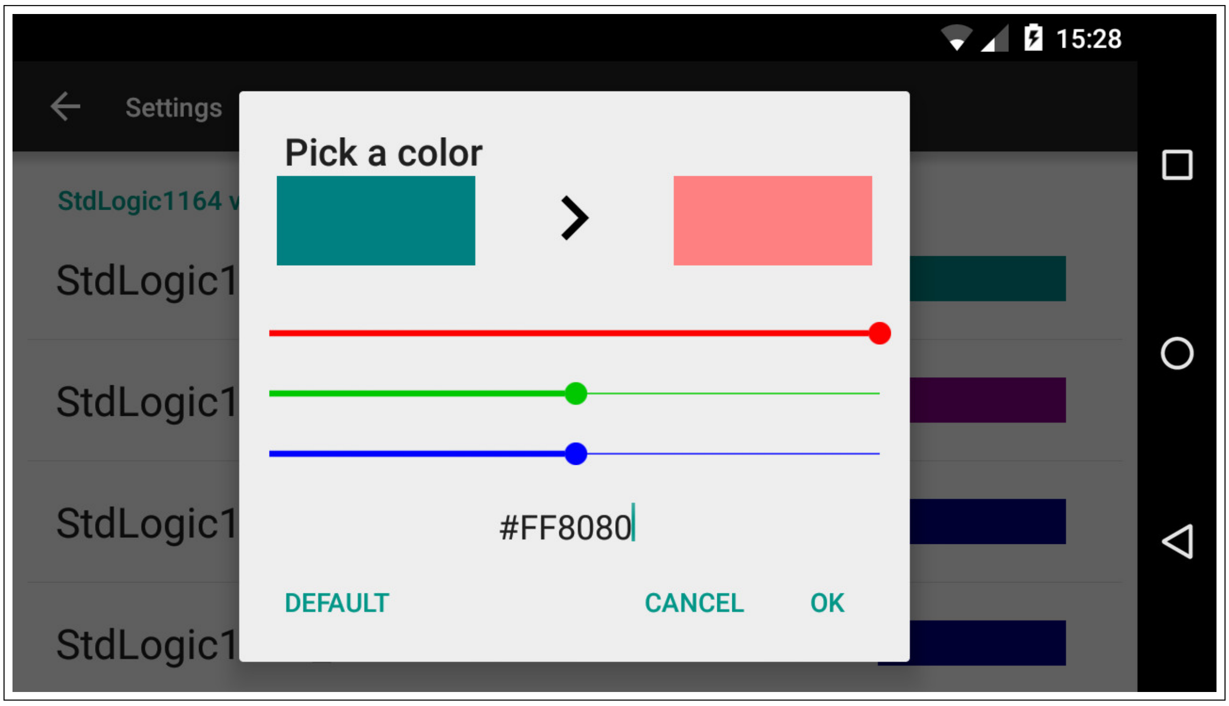


Figure 3.10: Illustration of the dialog, used to change a *std_logic_1164* color.

of the *smart zoom fit* feature, directly after the selection of which waveforms to display. The use via the *Smart zoom fit* item from the action bar menu is unaffected.

The currently used colors for *std_logic_1164* are displayed in the settings window. Tapping on one of these opens a dialog that allows the user to choose a new color. What this dialog looks like can be seen in Figure 3.10. At the top, two boxes indicate the currently used color and the new color to use henceforth. The left box indicates the current color and the right box the new color. To change a color, 3 sliders are provided to set the red, green and blue amount to use. Below the sliders is a text field showing the current hexadecimal value of the chosen color. This field can be used to set a color directly, in case its hexadecimal value is known. The "DEFAULT" button will reset the color to its default value. The "CANCEL" button will close the dialog. A new color will only be applied, if the "OK" button is clicked. Unlike the number format, all settings done in the settings window will persist, even through application restarts.

4 Usability

The waveform viewer is generally usable and small simulations pose no problems. At the University of Hamburg, Hades is used in practical courses of some fields of study at the department of informatics, in order to teach the students about the computer architecture. Simulations that need to be developed within the scope of this course should be a good example for a practical use. A student could review and analyze the data or just quickly freshen up the memory while being on the to university.

The application can even be used with bigger waveform data, once the data has been read and the *smart zoom fit* feature is used. This does not mean that reviewing and analyzing the data will be pleasant. The performance of mobile devices is still worse than even the performance of a low-end desktop computer. As such, bigger simulations with more waveform data can cause problems. Either by means of memory requirements or by means of overview. This means that in those cases the user cannot zoom out to get an overview of the complete data and thereon decide which part of the data is of interest, since the GUI would most likely freeze up. If the GUI is unresponsive for too long, Android will report this to the user who might be surprised by this, not knowing what went wrong.

The *ring-oscillator* example given in section 2.2 can already be counted as one of those cases. The time it took to read the whole waveform data on the tested devices was around 40 to 60 seconds. This excludes the drawing time, which in comparison, is relatively low with only around 5 seconds. This is definitely not a fault of the new file format, and rather due to the mobile device and the hardware used. On the desktop computer, the time needed to read the same file was only around 1 to 2 seconds.

Another similar problem arises, because the user is able to change orientations. When the orientation is changed, the current activity gets destroyed and needs to be recreated. Since waveform data can be in the megabytes they are not directly used for recreation, instead a reference to the waveform file is remembered. This means that every time the orientation is changed, the waveform data needs to be read anew. Support for the ability to change orientations is not explicitly disabled, because it is a nice feature to have, when used with smaller data and its drawbacks are easily avoided. For example, the user can lock the orientation explicitly via Android's settings.

It is ultimately advised to think about whether it makes sense to try and use this waveform viewer, with the supplied data.

The application has been tested on Android versions ranging from API level 16 to the currently latest API level 23. All versions worked as intended. Lower versions should be supported as well.

Lastly, it is worth mentioning that parts of the software are definitely qualified to be used as replacements in the Java version of Styx, to improve its functionality and/or performance. Examples for this are the memory improvements and the ability of reading and displaying only a subset of all waveforms.

5 Outlook

A port of the Hades simulator already exists for Android. As of now, Hades and Styx are two separate applications on Android and cannot be used together. Styx can only work as a standalone program to display waveform data. It is very likely that they will be combined in the future.

Support for the ability to inspect a *std_logic_vector* was planned but not implemented. The same goes for the support of reading other file formats, mainly the *Value Change Dump* (vcd) file format. The possibility of vertical zooms were also taken into consideration. These features might still get implemented by the author of this paper, even beyond the scope of this bachelor thesis.

At the end of the deadline for this thesis, the author of this paper became aware of the *SurfaceView* class. This class might be able to solve some of the problems regarding the drawing time and the GUI freezes in general. The main reason being, that drawing is performed in a different thread, as opposed to drawing on the GUI thread. Tests have not been made as of yet, so this is just speculation, after reading some information about this class. If the use of this class truly increases performance, it would definitely benefit the usability.

Ultimately time will also benefit the application. The hardware used is constantly improved, providing better and better performances on mobile devices. This will reduce problems like the reading and drawing time, as explained in chapter 4.

Appendix A

Gesture detection

```java
public class MyView extends View
{
    private ScaleGestureDetector mScaleGestureDetector;
    private GestureDetector mGestureDetector;

    // ...

    public MyView (Context context)
    {
        super(context);

        // create the detectors
        mScaleGestureDetector = new ScaleGestureDetector(context, new
           MyScaleGestureListener());
        mGestureDetector = new GestureDetector(context, new
           MyGestureListener());
    }

    @Override
    public boolean onTouchEvent (MotionEvent event)
    {
        boolean handled = false;

        // let the detectors handle the event
        handled = mScaleGestureDetector.onTouchEvent(event);
        handled = mGestureDetector.onTouchEvent(event) || handled;

        return handled;
    }

    private class MyScaleGestureListener extends
       ScaleGestureDetector.OnScaleGestureListener
    {
        @Override
        public boolean onScale (ScaleGestureDetector detector)
        {
            return false;
        }

        @Override
        public boolean onScaleBegin (ScaleGestureDetector detector)
        {
            return false;
        }

        @Override
        public void onScaleEnd (ScaleGestureDetector detector)
        {
        }
    }

    private class MyGestureDetector extends
        GestureDetector.OnGestureListener
    {

        @Override
        public boolean onDown (MotionEvent e)
        {
            return false;
        }

        @Override
        public void onShowPress (MotionEvent e)
        {
        }

        @Override
        public boolean onSingleTapUp (MotionEvent e)
        {
            return false;
        }

        @Override
        public boolean onScroll (MotionEvent e1, MotionEvent e2, float
            distanceX, float distanceY)
        {
            return false;
        }

        @Override
        public void onLongPress (MotionEvent e)
        {
        }

        @Override
        public boolean onFling (MotionEvent e1, MotionEvent e2, float
            velocityX, float velocityY)
        {
            return false;
        }
    }
}
```

Listing 8.1: Usage of gesture detection classes, that are provided by Android.

Dragging

```java
@Override
public boolean onScroll (MotionEvent e1, MotionEvent e2, float
    distanceX, float distanceY)
{
    // stop zoom selection
    if (mSelectingArea)
    {
        mSelectingArea = false;
    }

    // variables storing information about the
    // distance traveled on each axis
    mMovedHorizontally += distanceX;
    mMovedVertically += distanceY;

    // whether we scrolled
    boolean scrolled = false;

    // in case the pointer responsible for scrolling
    // went down in the area for signal names
    if (hitTestNames(e1.getX(), e1.getY()))
    {
        if (mDragAndDropActive)
        {
            // perform drag for drag and drop
            mDragPosX -= distanceX;
            mDragPosY -= distanceY;

            // update screen
            postInvalidateOnAnimation();

            // prematurely return (disables scrolling while performing
            // a drag and drop operation)
            return true;
        }
        else
        {
            // set mMoveHorizontally to 0, to only allow scrolling
            // vertically
            mMovedHorizontally = 0;
        }
    }
    else if (e1.getX() > mContentRect.width())
    {
        // first test for scrolling
        // if we are not yet scrolling
        if (!mIsBeingDraggedHorizontally && !mIsBeingDraggedVertically)
        {
            // test on which axis to start scrolling
            if (Math.abs(mMovedHorizontally) >
                Math.abs(mMovedVertically))
            {
                mIsBeingDraggedHorizontally = true;

                // reset mMovedHorizontally, or else we will have a
                // jump at the beginning
                mMovedHorizontally = 0;
            }
            else
            {
                mIsBeingDraggedVertically = true;
                mMovedVertically = 0;
            }
        }
        // if we are already scrolling vertically check for horizontal
        // scrolling with an increased threshold
        else if (!mIsBeingDraggedHorizontally &&
                 Math.abs(mMovedHorizontally) > mScrollFactor *
                    mTouchSlop)
        {
            mIsBeingDraggedHorizontally = true;
            mMovedHorizontally = 0;
        }
        else if (!mIsBeingDraggedVertically &&
                 Math.abs(mMovedVertically) > mScrollFactor *
                    mTouchSlop)
        {
            mIsBeingDraggedVertically = true;
            mMovedVertically = 0;
        }

        // now do the actual scrolling
        if (mIsBeingDraggedHorizontally &&
            Math.abs(mMovedHorizontally) >= 1)
        {
            // adjust the scroll position
            mScrollX += (int) mMovedHorizontally;

            // adjust the counter
            mMovedHorizontally -= (int) mMovedHorizontally;

            // set the flag
            scrolled = true;
        }
        if (mIsBeingDraggedVertically && Math.abs(mMovedVertically) >=
           1)
        {
            mScrollY += (int) mMovedVertically;
            mMovedVertically -= (int) mMovedVertically;
            scrolled = true;
        }

        //in case we scrolled, update screen
        if (scrolled)
        {
            //check for horizontal edge glow, add to pull if necessary
            if (mScrollX < 0)
            {
                mEdgeGlowLeft.onPull((float) mScrollX /
                   mContentRect.width());
                mEdgeGlowLeftActive = true;
                mScrollX = 0;
            }
```

```
                else if (mScrollX > mSizeX - mContentRect.width())
                {
                    mEdgeGlowRight.onPull((float) (mScrollX - mSizeX +
                        mContentRect.width()) /
                                                mContentRect.width());
                    mEdgeGlowRightActive = true;
                    mScrollX = mSizeX - mContentRect.width();
                }

                //check for vertical edge glow, add to pull if necessary
                if (mScrollY < 0)
                {
                    mEdgeGlowTop.onPull((float) mScrollY /
                        mContentRect.height());
                    mEdgeGlowTopActive = true;
                    mScrollY = 0;
                }
                else if (mScrollY > mSizeY - mContentRect.height())
                {
                    mEdgeGlowBottom.onPull((float) (mScrollY - mSizeY +
                        mContentRect.height()) /
                                                mContentRect.height());
                    mEdgeGlowBottomActive = true;
                    mScrollY = mSizeY - mContentRect.height();
                }

                postInvalidateOnAnimation();
            }
        }

        return true;
}
```

Listing 8.2: How dragging is handled.

Flinging

```java
@Override
public void computeScroll ()
{
    super.computeScroll();

    // do the flinging
    if (mScroller.computeScrollOffset())
    {
        //horizontal fling
        if (mCarefulLeftFling)
        {
            if (mScroller.getCurrX() < 0)
            {
                mScrollX = 0;

                //absorb velocity for edge glow if necessary
                if (!mEdgeGlowLeftActive)
                {
                    mEdgeGlowLeft.onAbsorb((int) mScroller.getCurrVelocity());
                    mEdgeGlowLeftActive = true;
                }
            }
            else
            {
                mScrollX = mScroller.getCurrX();
            }
        }
        else if (mCarefulRightFling)
        {
            if (mFlingStartX + mScroller.getCurrX() > mSizeX - mContentRect.width())
            {
                mScrollX = mSizeX - mContentRect.width();

                //absorb velocity for edge glow if necessary
                if (!mEdgeGlowRightActive)
                {
                    mEdgeGlowRight.onAbsorb((int) mScroller.getCurrVelocity());
                    mEdgeGlowRightActive = true;
                }
            }
            else
            {
                mScrollX = mFlingStartX + mScroller.getCurrX();
            }
        }
        else
        {
            mScrollX = mFlingStartX + mScroller.getCurrX();
        }

        //vertical fling
        mScrollY = mScroller.getCurrY();
```

```

            if (mScrollY < 0)
            {
                mScrollY = 0;

                //absorb velocity for edge glow if necessary
                if (!mEdgeGlowTopActive)
                {
                    mEdgeGlowTop.onAbsorb((int)
                       mScroller.getCurrVelocity());
                    mEdgeGlowTopActive = true;
                }
            }

            if (mScrollY > mSizeY - mContentRect.height())
            {
                mScrollY = mSizeY - mContentRect.height();

                //absorb velocity for edge glow if necessary
                if (!mEdgeGlowBottomActive)
                {
                    mEdgeGlowBottom.onAbsorb((int)
                       mScroller.getCurrVelocity());
                    mEdgeGlowBottomActive = true;
                }
            }

            postInvalidateOnAnimation();
        }
    }
```

Listing 8.3: Source code for the *computeScroll()* method.

Smart zoom fit

```java
public void smartZoomFit ()
{
    if (mWaveforms != null)
    {
        double relativeLeft = mScrollX / (double) mSizeX;

        if (mSmartZoomFitFactor == -1)
        {
            //stores the interval times between events
            Map<Double, Integer> eventTimeIntervals = new TreeMap<>();

            double[] times;

            //for each waveform add all event intervals to our map
            for (Waveform wf : mWaveforms)
            {
                times = wf.getTimes();

                for (int i = 0; i < times.length - 1; i++)
                {
                    double key = times[i + 1] - times[i];

                    //if our interval time already exists increase its
                        count
                    if (eventTimeIntervals.containsKey(key))
                    {
                        eventTimeIntervals.put(key,
                            eventTimeIntervals.get(key) + 1);
                    }
                    //otherwise make a new entry
                    else
                    {
                        eventTimeIntervals.put(key, 1);
                    }
                }
            }

            //calculate the average interval time
            double mean = 0;
            int amount = 0;

            for (Double key : eventTimeIntervals.keySet())
            {
                mean += key * eventTimeIntervals.get(key);
                amount += eventTimeIntervals.get(key);
            }

            mean = mean / amount;

            //if a certain threshold of approximated events to be
                drawn is exceeded
            if (mean > 0 && mOverallMaxTime / mean > 100)
            {
                //calculate a feasible zoom factor
                mSmartZoomFitFactor = mContentRect.width() /
```

```
                   (mContentRect.width() /
                     (mOverallMaxTime / mean *
                     100));
      }
    }

    //zoom onto the new time scale so fewer events need to be drawn
    if (mSmartZoomFitFactor > -1)
    {
      // calculate new x size
      mSizeX = (long) (mContentRect.width() *
        mSmartZoomFitFactor);

      // adjust size, if necessary
      if (mSizeX > mMaxSizeX)
      {
        mSizeX = mMaxSizeX;
      }
      else if (mSizeX < mContentRect.width())
      {
        mSizeX = mContentRect.width();
      }
    }

    // calculate new time scale
    mTimeScale = (int) Math.floor(Math.log10(
        mOverallMaxTime * mContentRect.width() / mSizeX));
    mTimeMarkInterval = mSizeX / mOverallMaxTime * Math.pow(10,
      mTimeScale);

    // adjust scroll to the left time of the selected area
    mScrollX = (long) (relativeLeft * mSizeX);

    if (mScrollX < 0)
    {
      mScrollX = 0;
    }
    else if (mScrollX > mSizeX - mContentRect.width())
    {
      mScrollX = mSizeX - mContentRect.width();
    }

    postInvalidateOnAnimation();
  }
}
```

Listing 8.4: Source code for the smart zoom fit algorithm.

Drag and Drop

```java
@Override
public void onLongPress (MotionEvent e)
{
    //disable further long presses so we don't trigger it again after
    //resending the down event and end up in a loop
    mGestureDetector.setIsLongpressEnabled(false);

    if (hitTestNames(e.getX(), e.getY()) &&
        mSelectedWaveformsCount > 0)
    {
        //initiate drag and drop
        mDragAndDropActive = true;

        //move selected waveforms to new array
        int index = 0;
        for (int i = 0;
             i < mSelectedWaveforms.length && index <
                mSelectedWaveformsCount;
             i++)
        {
            if (mSelectedWaveforms[i])
            {
                mDraggedWaveforms[index] = mWaveformsOrder[i];
                index++;
            }
        }

        //move unselected waveforms up in the original array
        index = 0;
        for (int i = 0;
             i < mWaveforms.length && index < mWaveforms.length -
                mSelectedWaveformsCount;
             i++)
        {
            //if the current waveform is not selected move it to
            //current index position
            if (!mSelectedWaveforms[i])
            {
                int tmpIndex = mWaveformsOrder[index];
                mWaveformsOrder[index] = mWaveformsOrder[i];
                mWaveformsOrder[i] = tmpIndex;
                mSelectedWaveforms[index] = false;
                index++;
            }
        }

        //adjust mSizeY
        mSizeY = Math.max(mContentRect.height(),
                          (mWaveforms.length -
                             mSelectedWaveformsCount) *
                             mWaveformHeight);

        //adjust scroll position if necessary
        if (mScrollY > mSizeY - mContentRect.height())
        {
            mScrollY = mSizeY - mContentRect.height();
        }

        //resend down event, so we can continue dragging/ordering signals
        onTouchEvent(e);
    }
    else if (e.getX() > mContentRect.left)
    {
        jumpToCursorPosition();
    }

    postInvalidateOnAnimation();
}
```

Listing 8.5: Source code showing how the drag and drop is initiated.

Bibliography

[1164] IEEE. *Stdlogic1164*. URL: https://standards.ieee.org/downloads/1076/1076.2-1996/std_logic_1164-body.vhdl (visited on 10/28/2015).

[93] *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Stdlogic1164)*. 1993. DOI: 10.1109/IEEESTD.1993.115571.

[BW] Tony Bybell and Joel Wheeler. *GTKWave*. URL: http://gtkwave.sourceforge.net/ (visited on 10/28/2015).

[DevBSO] Android Developers. *Build System Overview*. URL: http://developer.android.com/sdk/installing/studio-build.html (visited on 10/28/2015).

[DevMem] Android Developers. *Managing Your App's Memory*. URL: https://developer.android.com/training/articles/memory.html (visited on 10/28/2015).

[DevNDK] Android Developers. *Android NDK*. URL: https://developer.android.com/ndk/index.html (visited on 10/28/2015).

[DevRun] Android. *ART and Dalvik*. URL: https://source.android.com/devices/tech/dalvik/ (visited on 10/28/2015).

[DevSAF] Android Developers. *Storage Access Framework*. URL: http://developer.android.com/guide/topics/providers/document-provider.html (visited on 10/28/2015).

[DroidAPI] Android Developers. *Package Index*. URL: http://developer.android.com/reference/packages.html (visited on 10/28/2015).

[GooOMP] Google et al. *Our Mobile Planet*. URL: https://think.withgoogle.com/mobileplanet/en/ (visited on 10/28/2015).

[GooTWG] Google. *Think With Google*. URL: https://www.thinkwithgoogle.com/ (visited on 10/28/2015).

[HaAPI] TAMS. *Hades API*. URL: https://tams.informatik.uni-hamburg.de/applets/hades/classdoc/index.html (visited on 10/28/2015).

[HaHo] TAMS. *Hades Simulation Framework*. URL: https://tams.informatik.uni-hamburg.de/applets/hades/webdemos/index.html (visited on 10/28/2015).

[JavaAPI] Oracle. *Java^{TM} Platform, Standard Edition 7 API Specification*. URL: http://docs.oracle.com/javase/7/docs/api/ (visited on 10/28/2015).

[JavaOSS] Oracle. *Java Object Serialization Specification*. URL: https://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html (visited on 10/28/2015).

[JavaVM] Oracle. *Java HotSpot^{TM} Virtual Machine Performance Enhancements*. URL: https://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html (visited on 10/28/2015).

[Nor06] Hendrich Norman. *HADES Tutorial*. Dec. 2006. URL: https://tams.informatik.uni-hamburg.de/applets/hades/archive/tutorial.pdf (visited on 10/28/2015).

[StyHo] TAMS. *Hades waveform-viewer*. URL: https://tams.informatik.uni-hamburg.de/applets/hades/webdemos/styx.html (visited on 10/28/2015).

Selbständigkeitserklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin mit der Einstellung der Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik einverstanden.

Hamburg, den 28.10.2015

Lars Lütcke