Bonifaz Kaufmann

Design and Implementation of a Toolkit for the Rapid Prototyping of Mobile Ubiquitous Computing

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Studium Informatik

Alpen-Adria-Universität Klagenfurt

Fakultät für Technische Wissenschaften

Begutachter
Univ.-Prof. Dipl.-Ing. Dr. Martin Hitz
Institut für Informatik-Systeme

Vorbegutachter
Ass.-Prof. Mag. Dr. Gerhard Leitner
Institut für Informatik-Systeme

August 2010

Bonifaz Kaufmann

Design and Implementation of a Toolkit for the Rapid Prototyping of Mobile Ubiquitous Computing

MASTER THESIS

Thesis submitted in fulfillment of the requirements for the academic grade of

Master of Science (Dipl.-Ing.)

Study: Computer Science

University of Klagenfurt

Faculty of Technical Sciences

Supervisor
Univ.-Prof. Dipl.-Ing. Dr. Martin Hitz
Institute for Informatics Systems

Co-Supervisor
Ass.-Prof. Mag. Dr. Gerhard Leitner
Institute for Informatics Systems

August 2010

Declaration of Honor

I hereby confirm on my honor that I personally prepared the present academic work and carried out myself the activities directly involved with it. I also confirm that I have used no resources other than those declared. All formulations and concepts adopted literally or in their essential content from printed, unprinted or Internet sources have been cited according to the rules for academic work and identified by means of footnotes or other precise indications of source.

The support provided during the work, including significant assistance from my supervisor has been indicated in full.

The academic work has not been submitted to any other examination authority. The work is submitted in printed and electronic form. I confirm that the content of the digital version is completely identical to that of the printed version.

I am aware that a false declaration will have legal consequences.

Bonifaz Kaufmann Klagenfurt, 31. August 2010

This thesis is dedicated
to my beloved partner in life,
Monika

Acknowledgements

I greatly appreciate the support and advice of my advisor Prof. Dr. Martin Hitz before and during the completion of this work as well as Dr. Gerhard Leitner for his input and discussions. In addition, I like to thank Leah Buechley, David Mellis, Emily Lovell and Hannah Perner-Wilson for their useful hints and in particular for our numerous critical and valuable conversations during my stay at MIT.

I am very grateful for having a brother, Christian, who inspired me in so many ways and also for never getting bored when I euphorically talked about my studies. Most important, I would like to express my warmest thank you to my better half and best friend. Monika, thank you so much for your unconditional support and all your love, without you, this work and my studies would have never had happened.

Abstract

Ubiquitous computing applications often entail smartphones to communicate or control embedded systems. As a result, many projects can be found where smartphones have been connected to microcontrollers to create tangible interfaces, sense the environment, implement pervasive games, or control everyday things. Although smartphones are equipped with powerful sensors and actuators, using these interfaces outside the phone is difficult. Making smartphone interfaces available to other devices is a laborious process that requires deep knowledge in programming, electronics, as well as industrial and interaction design. This thesis explores how the complexity of prototyping mobile ubiquitous computing devices can be reduced with the aid of Amarino, a software toolkit to simplify the prototyping process when dealing with smartphones and microcontrollers in order to support developers getting quick results while focusing on their specific issues rather than spending time for example on implementing communication protocols. The Amarino software toolkit was developed in context of the present thesis and is discussed and examined exhaustively throughout this document.

Contents

Part I

Rapid Prototyping in Mobile Ubiquitous Computing

Chapter **1**

1 Introduction

Weiser's ubiquitous computing [1] promoted invisible computers, disappearing from the user's attention, but smartphones are the opposite of this compelling vision. In [1] Weiser clearly states that

> "*Even the most powerful notebook computer, with access to a worldwide information network, still focuses attention on a single box*".

This is also completely true for nowadays smartphones. Absorbing most of a user's attention, a smartphone can isolate a person when using it, resulting in temporary social distance from people nearby. Although mobile phones would fit Weiser's definition of ubiquitous computing as devices which people carry in their pockets, not being aware of the device itself anymore, as soon as it comes to interaction with the phone, this paradigm is broken. I believe that this is mostly a result of its paucity of connectivity and therefore a lack of interface alternatives.

1.1 Motivation

In smartphones, graphical user interfaces are still predominant and connecting a smartphone to an intuitive tangible interface is still difficult. Amft and Lukowicz [2] stress that interaction with current smartphones is entirely focused on the device itself and there are almost no connectivity options apart from Bluetooth and WLAN[1] to extend input and output facilities. Indeed, smartphones appear more as closed universes in terms of interface extensibility. Projects where

[1] Wireless Local Area Network

interaction with a smartphone or using of its embedded sensors is desired are difficult to realize. Despite these hurdles, I think that when smartphones are easier to combine with other electronics, they will have the potential to become more ubiquitous by being connected to tangibles. Conversely, tangible devices can benefit from connectable smartphones, as they might serve as mobile computation units due to their Internet connectivity, size, computing power, and distribution.

Tangible user interfaces (TUI) as proposed by Ishii and Ullmer [3] were without any doubt influential in interaction design wherein interactive surfaces and tangible objects are seamlessly coupled to digital information, enriching the user experience by appearing more natural to a user when grasping and manipulating real-world objects instead of moving onscreen windows and pressing virtual buttons. Tangible interfaces are recognized as easy to understand and use, largely because of their close mapping of action and perception, unifying input and output, as discussed in [4]. However, Shaer and Jacob [5][6] argue that tangible interfaces are still intricate to develop compared to conventional user interfaces. Traditional user interface design benefits from great tools and experience gathered over decades; tangible user interface design, on the other hand, is still in strong need of supportive methodologies, concepts, and tools to overcome design and development complexity. Facing the connectivity issue as well as the complexity of prototyping tangible interface projects, I have built Amarino, a toolkit to ease the development of innovative interfaces, in order to bridge the gap between smartphones and other interaction components by simplifying communication and interfacing among them.

The Amarino software toolkit consists of two components, an Android application which runs on a mobile device and a software library for the Arduino [7] open hardware prototyping platform. Both components are communicating via Bluetooth. Having a toolkit like Amarino enables developers to connect and exchange data between Android smartphones and Arduino microcontrollers without designing communication protocols or dealing with connectivity and reliability issues. Writing programs based on Amarino simplifies application development because communication handling is transparent. Moreover, Amarino provides access to most available built-in phone sensors without the need to write a mobile phone application at all.

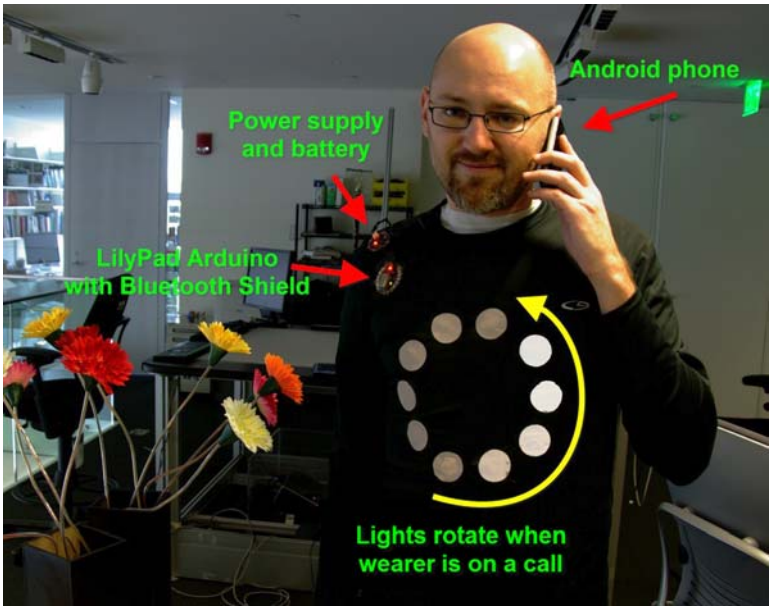


Figure 1: Shirt displaying the phone state, e.g. blinking when ringing, rotating lights when talking.

Using Amarino, a proof of concept project CallMyShirt (Figure 1) for a wearable interface showing the phone state, can be realized without writing a single line of code for the phone and only about 150 lines of code to program the microcontroller, mostly to control the LEDs. The CallMyShirt project is described in detail later in Chapter Showcases.

1.2 Structure

This thesis is divided into two major parts, one which focuses on background information in mobile ubiquitous computing, whereas the second part is dedicated to practical details of Amarino. Part one has started with this introduction and moves on by providing an overview of research related to mobile ubiquitous computing. But instead of simply summarizing research about mobile ubiquitous computing, Chapter 2 wants also to stress the need of a prototyping toolkit in this area. The concept and purpose of such a toolkit is presented in Chapter 3, by reasoning about conceptional aspects, benefits and design as well as platform considerations. Since Amarino is based on Android and Arduino, these two platforms are outlined in Chapter 4 and 5. The second part starts with Chapter 6 by describing Amarino from a user's point of view, followed by Chapter 7 presenting architectural details. Technical details are discussed in Chapter 8 and Chapter 9 showcases some projects realized with the aid of Amarino. Finally, Chapter 10 draws a conclusion while emphasizing particular aspects of Amarino.

Chapter 2

2 Mobile Ubiquitous Computing

When Mark Weiser introduced ubiquitous computing (ubicomp) for the first time in 1991 [1], it quickly became clear that his vision of integrating "invisible" computers effectively and seamlessly into our living and working spaces couldn't be realized without difficulties [8]. Weiser states that an experimental approach, in particular building working prototypes, will be the default way of exploring ubiquitous computing [8]. Long before Apple's iPad started its success story, Weiser and his colleagues developed at Xerox PARC[2] a palm-sized interactive device with a pressure sensitive screen called Tab and a portable touch screen pad similar in size to the iPad, consistent to the scrap paper metaphor, which lies about the room and can easily be picked up and used by different people. The researchers among Weiser mainly developed these prototypes to gain insights into ubiquitous computing issues. One outcome of this exploration was that low-powered devices with high-speed wireless capabilities are needed as well as better usable pens to work on big screens. Weiser also addressed issues related to network protocols available at that time as well as emphasized the importance of appropriate interaction paradigms and supportive applications in order to pave the way for ubiquitous computing. [8]

Many of Weiser's examined issues have been addressed by later upcoming mobile computing research efforts [9]. And with the raise of mobile computing driven by the dissemination of smartphones, the possibilities of establishing wide-spread ubiquitous computing are more promising than ever. Using smartphones in context with wearables, tangibles, pervasive games or in sensor environments already led

[2] Palo Alto Research Center

to a wide selection of innovative research projects. A few of such intrigued projects will be presented next. However, beforehand it should be mentioned that each presented project developed its own solution to establish communication between a smartphone and a microcontroller in order to build ubiquitous devices. This is mainly because there were no supportive development tools available aiding the prototyping process of ubicomp devices. The Amarino software toolkit addresses the lack of available tools by providing a toolkit that simplifies the development of microcontroller driven devices that need to connect to smartphones. Smartphones are evidently playing an important role in ubiquitous computing scenarios, due to their significant processing power and networking capabilities and their establishment among a large user base.

In [10] a mobile group communication system is demonstrated where all group members have bracelets with different colored LEDs serving as a quiet, semi-public, tangible and intimate wearable ambient display to indicate specific group messages.

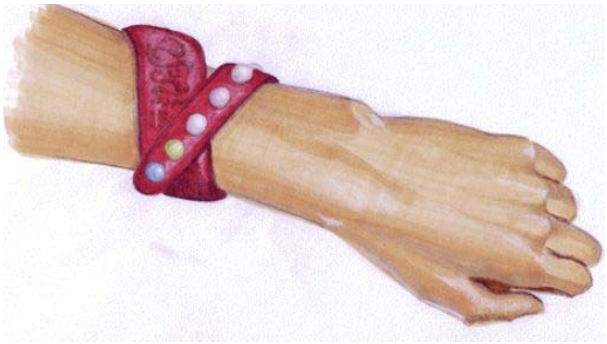


Figure 2: Sketch of the bracelet – a wearable ambient display as a group communication device (copied from [10])

All bracelets incorporate five multicolor lights, each mapped to an individual group member, plus one separate white light representing the entire group. The group light indicates the volume of sent messages within a group by changing its brightness accordingly. A color-coded message can be sent to the group by pressing or releasing three different metal snaps on the bracelet, one for red, green and blue. For example, when a participant wants to send a *red* signal to the group, she closes the red snap on her bracelet. This action will be transmitted via Bluetooth to her phone. The phone will then broadcast the message to all phones within the group. After the message has arrived on a recipient's phone, it is transmitted again

via Bluetooth to the recipient's bracelet and the light representing the sender is illuminated red.

WeWrite [11] uses a mobile phone to send messages via Bluetooth to a LilyPad Arduino [12] microcontroller which in turn drives LED-arrays mounted on a shirt to display a received text message on the wearable. Additionally, *WeWrite* demonstrates an alternative way to display text using a glove with one LED-column sewn on it; in this setup the text becomes readable only when the glove is moved in space. Another more intimate approach called *united-pulse* [13] enables partners to feel the pulse of each other by wearing a ring on the finger connected to a mobile phone to transmit the heartbeat to the other ring. A more generic approach to communicate using a wearable Bluetooth ring in combination with a mobile phone is explored in [14]. Constanza et al. [15] demonstrate *eye-q* an unobtrusive display mounted on the backside of an eyeglass frame delivering silent peripheral visual cues to the user about phone events without disturbing the direct environment.

Figure 3: The *eye-q* prototype showing the eyeglass and its embedded tiny displays on either side used to provide visual cues to the wearer when the phone is ringing or a message has arrived (copied from [15])

The authors of [16] designed a pervasive game that motivates players to be active over the day. A wearable accelerometer is logging movement of a player sending data to her mobile phone to boost a virtual representative in an online racing multi-player game. Using mobile phones to collect sensor data is also examined in [17] but in a different context. The researchers built a system *MobSens* for large scale environmental sensing with mobile phones. Mobile phones equipped with sensors to measure air pollution in traffic were used to generate a pollution map of a city.

Although developers always found a solution to establish communication between a microcontroller and a smartphone, they all developed individual solutions for the same issue. To my best knowledge, there is no tool to support the prototyping process when smartphones and microcontrollers need to communicate with each other via Bluetooth. Even Tokuhisa et al. [18] who developed *xtel*, a development environment to support prototyping in the realm of ubiquitous computing, have not considered the integration of smartphones. Their tool consists of three components: a micro control unit board with short range wireless capabilities, a virtual machine-based runtime and programming environment and a peer-to-peer network library to handle data flow. Although *xtel* shares the aim with Amarino to accelerate and support prototyping of ubiquitous devices it does not take mobile phones into account. Furthermore, in contrast to *xtel*, Amarino uses affordable, available and widely used off-the-shelf components that delight already a diverse audience.

Chapter 3

3 Design Considerations

3.1 Approach

This section describes the motivation for building a toolkit to ease the communication between mobile phones and microcontrollers. I present which goals I had in mind when designing Amarino and discuss them briefly. The platform decision is explained at the end of this chapter.

In Chapter 2 we saw that many applications exist where a smartphone with its decent processing power, sensors and its rich connectivity to network providers and the Internet, is used to communicate with an external microcontroller. Depending on the configuration, the phone might be seen as an extension of the microcontroller to display, process or forward data for it, or the microcontroller in turn acts as a generic interface extension for the phone. Both settings are likely to occur in a variety of projects.

From a developer's point of view there are three main concerns when dealing with smartphones in combination with tangible devices:

- Developing a smartphone application
- Building a tangible device and programming electronics
- Implementing a communication protocol

Each of these three tasks requires different development skills. Although some skills are overlapping they might be seen as separate subprojects. In order to build smartphone applications, one has to be familiar with the application programming

interface and the programming language of the desired platform. Programming a microcontroller requires expertise in microcontroller architecture and the programming language for the microcontroller which is usually different to the smartphone programming language. For designing the communication protocol, network expertise could be helpful and for designing tangible devices, electrical engineering skills are useful. Since the aim was to make prototyping of "smartphone-talks-to-microcontroller" applications as easy as possible, I was reasoning which steps could be simplified.

The electronics part is entirely application dependent and can hardly be cut down, due to the range of possible application scenarios. In contrast, sending and receiving data from one device to another is a standard task which should be transparent to the developer in order to ease development. In addition, most smartphones share a common set of sensors. Thus, accessing smartphone sensors could be simplified by providing a preinstalled application allowing a developer to select a set of events to be transmitted to the microcontroller. A service running as a background process on the smartphone could take care of sending the events to the microcontroller whenever they occur. An event in this context might be a sensor reading, a GPS fix, a phone call, a text message, or any other action available on smartphones. Microcontroller programming can be deskilled by offering a library which implements the underlying communication protocol, optimized to communicate with the phone and an event handling mechanism which catches incoming events and offers them accordingly.

To address a diverse audience, I decided on three main goals to focus on:

- Simple to use (beginners)
- Extensible (experts)
- Open (community)

On the one hand side, Amarino should be easy to use, allowing beginners without mobile programming experience to create applications with the aid of Amarino, and on the other hand side, experts should have the option to extend the toolkit and add functionality. Since I expect the toolkit to be used in prototyping projects the source code should be open and modifiable. This will also allow novices to

study the internal structure and operation in order to learn and broaden their mobile programming skills or experts might optimize the toolkit for their particular projects.

3.2 Platform Considerations

When I started designing Amarino one main question was which platforms Amarino should support at first. Although the toolkit might be implemented later for several different platforms, I wanted to start with platforms which are supportive especially for the prototyping process. I had to look at mobile phone development platforms as well as microcontroller prototyping platforms. In this section I first take a look at predominant mobile operating systems and argue why Android has been chosen. Then the motivation for the Arduino microcontroller platform is described.

3.2.1 Mobile Operating Systems

Earl Oliver provides an overview in [19] considering preferable properties mobile operating systems should have to be valuable in research setups. Comparing Android, BlackBerry, iPhone, Symbian and Windows Mobile, he comes to the conclusion that none of them does really fit all his proposed requirements. In short this is because Android 1.x lacks Bluetooth support, BlackBerry does not allow programmatically scanning for Wi-Fi hotspots or cell towers, iPhone has no support for third-party applications to run in background, Symbian is considered the hardest platform to write applications for while Windows Mobile implementations are heavily manufacturer dependent.

When thinking of Amarino there are five requirements the preferred mobile development platform should have.

- Bluetooth Support
  - Bluetooth Serial Port Protocol must be supported
- Background Processes
  - Third party applications must be able to run in the background
- Openness

  - Ideally the operating system platform is open source
- Accessibility
  - Freely available development environment, no additional fees, no restriction when installing applications on smartphones, easy to redistribute
- Low Complexity
  - Easy to learn and well documented

Running background processes and Bluetooth support are absolutely necessary for Amarino to work. Furthermore, since Amarino is a tool other developers are likely to modify and extend, the platform should also reflect these properties. Consequently, openness of the underlying development platform is strongly desirable. Amarino is made to be used in prototyping setups, thus setting up and using the environment, installing code on real devices and being able to easily redistribute their own application, are further requirements condensed as accessibility. Moreover, in order to address also beginners the programming language should not be too hard to learn (low complexity).

Requirements / Platforms	Bluetooth Support	Background Processes	Openness	Accessibility	Low Complexity
Android 2.x					
Blackberry OS					
iPhone OS					
Symbian					
Windows Mobile					

Table 1: Platform Requirements for Amarino

Windows Mobile is basically not open. Running background processes on iPhone is only possible when the device has gone through a “Jail Break” procedure, but since I want Amarino to run on off-the-shelf devices this is not what I was looking for. Sadly, even the latest version of Apple's iPhone OS (Apple iOS 4) only allows very restricted background processing not suitable for the kind of background service Amarino is heading for. The iPhone OS is also not open and applications developed on iPhone OS can only be redistributed via Apple’s proprietary Apple

Store. Convincing programming novices to work with Amarino when it is built on Symbian would be difficult, because the framework is more complex and requires more time compared to other platforms to develop applications on, hence counterproductive in prototyping scenarios. BlackBerry OS is an open platform with Bluetooth support and the ability to run third-party applications in background. Although the BlackBerry OS is open, it is also a proprietary operating system and to use some of the low level API[3] functions, a developer has to pay for a certificate to digitally sign her application. Android is also open and able to run background applications and since Android 2.0, Bluetooth support has been integrated. In addition, there exists a solution to get Bluetooth working on consumer devices before Android 2.0 by using an unofficial Bluetooth API written by Stefano Sanna and Emanuele Di Saverio [20] that uses reflections to access hidden Bluetooth functions available on almost all 1.x Android devices.

At the end Android was selected as the preferred development platform. I agree with Earl Oliver [19] that Android has the lowest entry level for developers enabling even beginners to start with. Each developer can download the Android SDK[4] for free, without registration and install applications on Android driven devices. Furthermore, since Android 2.0, the Android SDK fulfills all mentioned requirements for Amarino.

3.2.2 Microcontroller Prototyping Platforms

To make "things" talk to smartphones, a microcontroller is normally embedded or attached to the object or to the installation, implementing the communication infrastructure and some logic to control electronics. In prototyping projects, generic, easy to use microcontrollers are preferred with the option to change the setup quickly. Fortunately, a recent open hardware prototyping platform named Arduino [7] has seen wide acceptance for prototyping projects like this. A comparable integrated development platform is Wiring [21]. Both share the same concept and either platform looks similar due to the reason that Arduino is based on the language syntax of Wiring and either environment is based on Processing [22].

[3] Application Programming Interface

[4] Software Development Kit

I decided on Arduino because it is widely used by researchers as well as hobbyists, the selection of available hardware is broader and the community is more active. Arduino has a steadily growing community with more than 120,000 Arduinos sold [23] since its foundation in 2005. Furthermore, 230,944 posts and 25,679 registered members were counted on the official Arduino forum whereas 2,879 posts from 835 members only on the Wiring forum (numbers from June 23, 2010). Arduino offers an open source development platform and a rich selection of input/output prototyping boards. The most popular board is the Arduino Duemilanove which has an ATmega168 microcontroller and 16 digital I/O pins and 6 analog pins. Many variations exist based on Arduino like the LilyPad [12] mentioned in the previous section which is a sewable version of Arduino optimized for use on wearables, or Boarduino [24] which has been modified to be plugged directly onto a breadboard. Furthermore, a selection of daughter boards, sensors and communication shields like Bluetooth-, XBee- and Ethernet shields are available to be easily hooked up to Arduino.

In the following two chapters the Android Operating System as well as the Arduino open hardware prototyping platform is introduced in detail and characteristics important to Amarino are discussed.

Chapter 4

4 Android – Open Source Mobile Operating System

"Android is a software stack for mobile devices that includes an operating system, middleware and key applications. The Android SDK provides the tools and APIs necessary to begin developing applications on the Android platform using the Java Language." [25]

4.1 Background

Android was released to the public in October 2008 and was initially implemented on one smartphone only, the HTC Dream commonly known as the T-Mobile G1. Since then many more smartphone manufacturers released mobile devices driven by Android. In June 2010, there were 60 different Android based mobile device models distributed to 49 countries by 59 carriers and 21 OEMs[5] [26]. Due to Android's openness, its rich developer toolset and because of the Java language, Android became quickly popular among mobile application developers. About 110,000 Android applications and games can be found in the Android market rising by about 15 percent each month [27] making it the second biggest mobile application store behind Apple's App Store which offers about 225,000 apps[6].

The following description of Android is solely a brief introduction of Android's building blocks and concepts aimed to introduce terms and notions later used to

[5] Original Equipment Manufacturers

[6] Colloquial usage of smartphone applications and games

describe Amarino. It is neither an exhaustive, nor a complete description of Android rather than focusing on key aspects in the context of Amarino.

4.2 Android System Architecture

The Android software stack consists of five key components as depicted in Figure 4: applications, an application framework, libraries, the Android runtime and a Linux kernel. In short, libraries written in C/C++ sitting on top of a Linux kernel and are accessible by applications through an application framework that offers services to applications and the Android runtime [28].

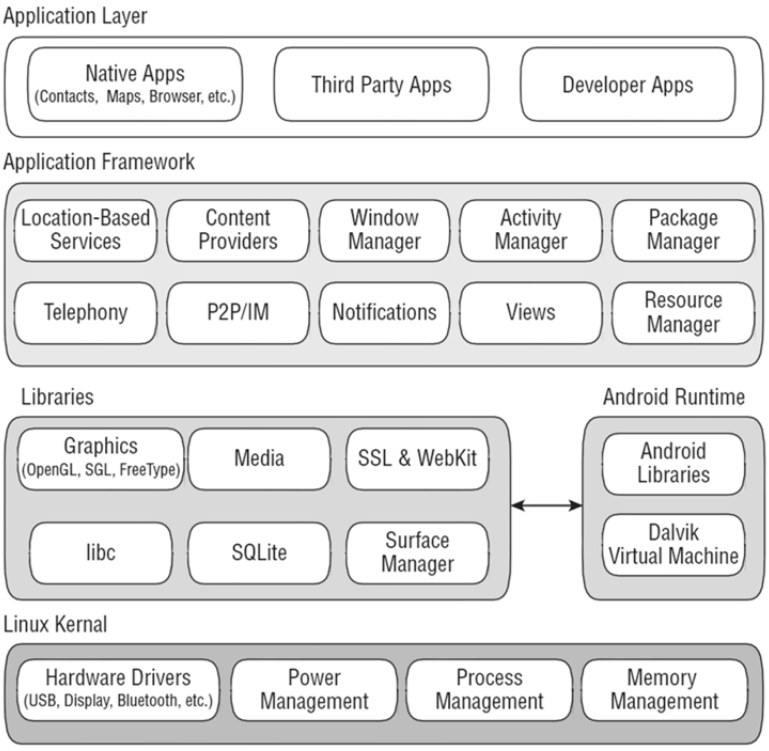


Figure 4: Software stack of Android (copied from [29])

4.2.1 Applications

Each device that comes with Android has a bunch of applications preinstalled. This is at least a home application, an email client, a calendar, a text messaging app, a browser, a contacts application and some others. They operate in the same layer as third-party applications and according to a Google advocate [29] Android makes no difference between native and third-party applications. Although often used as a marketing phrase, this is not entirely true. There are functions accessed by core applications normally hidden to application developers.

4.2.2 Application Framework

The application framework layer located above the C/C++ libraries and runtime provides access to the hardware by offering classes used to create Android applications. Additionally, the layer contains important managers responsible for controlling the application life-cycle, providing access to shared data like contacts, managing installed applications and resources and offers various other similar services.

4.2.3 Android Runtime

Below the application framework, the Android runtime can be found. The runtime consists of core Android libraries and the Dalvik virtual machine (VM). Besides Android specific libraries, the Android runtime includes a subset of core libraries from the Java Standard Edition and the Java Mobile Edition. The Dalvik VM is a virtual machine similar to Java's VM, however differs in some important aspects. Designed and developed by Dan Bornstein at Google, the Dalvik VM does not execute .class files, but highly optimized, compact .dex, Dalvik Executables, files. Android applications which are usually written in Java (actually it is also possible to develop C/C++ code using the Android NDK[7]) and compiled to .class files using the Java language compiler are transformed during compile time to .dex files utilizing the "dx" tool that is part of the Android SDK. The Dalvik VM has been designed for low memory requirements and to run multiple VM instances efficiently, while each Android application is executed in its own VM running in its own process. [25][30]

4.2.4 Libraries

Also below the application layer are native libraries written in C/C++. These libraries are implemented and complied for the specific underlying hardware architecture. A selection of common C/C++ libraries like the OpenGL 2D and 3D graphics library, the SQLite database library, the standard C system library (libc) and various other libraries are forming among additional Android specific libraries the layer on top of the Linux kernel.

[7] Native Development Kit

4.2.5 Linux Kernel

The Linux kernel, originally developed by Linus Torvalds in 1991, builds the foundation for Android's memory and process management as well as security and power management and networking services. In addition, the kernel contains all necessary hardware drivers providing a hardware abstraction layer for layers on top of the Linux kernel layer. [25]

4.3 Application Components

An Android application is a bundle of application components, resources and data files packaged into an Android package (.apk) file. Nonetheless, an Android application is a loosely coupled conglomerate of separately executable components. Therefore, instead of having a *main* method as a single entry point to start an application, familiar from desktop application development, in Android each component of an application can be started individually assuming the application allows it. For example, if application A implemented a fancy scrollable list of contacts and application B needs to display a list of contacts, application B could simply start, if possessing the permission, the specific application component which implements the scrollable contacts list of application A in order to display the list of contacts without implementing this particular feature again. Native Android applications offer already many useful application components which can be reused in that way. In part two of this thesis it will be shown that Amarino uses the same concept by calling the configuration screen of an Amarino Plug-in to smoothly integrate plug-ins developed by third parties.

Each Android application package is described by one structured XML file (*AndroidManifest.xml*). It declares the components available within the application and defines permissions required by external applications to start components. Additionally, the manifest file specifies other useful meta-information needed by Android to install the application properly, such as icon, title, version, and so on. The manifest file is explained in detail further below.

4.3.1 Activity

The most important application component in Android is the activity. In Android a visible user interface is implemented by extending the *Activity* class. Usually, one

screen refers to one activity. An application might consist of only one activity or several. For instance, a contacts application consists of several activities. One displaying a list of all contacts, another might show the details of a particular contact and a third activity might be used to edit a contact. *Views*, visible user interface elements, are added to an activity to build the graphical user interface in order to show information and to provide interaction elements to the user. Compared to desktop application development, an activity corresponds to a *Form.*[28]

An activity is in either one of three states: **active**, **paused** or **stopped**. There is exactly one **active** activity at a time, the foreground activity. It is visible, has the focus and receives user actions. An activity is **paused** if parts of the activity are still visible even when another activity has become **active**. For example, an **active** activity is translucent and/or does not cover the entire screen. In such cases the underlying activity is **paused** but still present in the memory, preserving its state so that it can be reactivated quickly if necessary. A **stopped** activity is not visible to the user anymore. If the system has enough memory a **stopped** activity will not be killed immediately. Though, a **stopped** activity might be killed as soon as memory is needed by the system. [25]

Since Android entirely controls the life-cycle of an activity, it is essential for Android developers to understand the activity life-cycle in order to create correct and well working applications. Figure 5 illustrates the life-cycle of an activity and its callback methods which are invoked by the Android OS to inform the activity about certain state transitions. An application developer simply overwrites the hook (callback) methods inherited from the *Activity* class to react properly to state changes.

The first method called, when an activity is started, is the *onCreate* method. In *onCreate* all the initialization operations like creating views, initialize variables, and so forth should be done. The *onStart* method is invoked shortly before the activity becomes visible to the user, while *onResume* will be called right before the activity gets the focus, allowing the user to interact with it. After *onResume* has been processed, the activity is considered as **active**. In this state the user can interact with the activity.

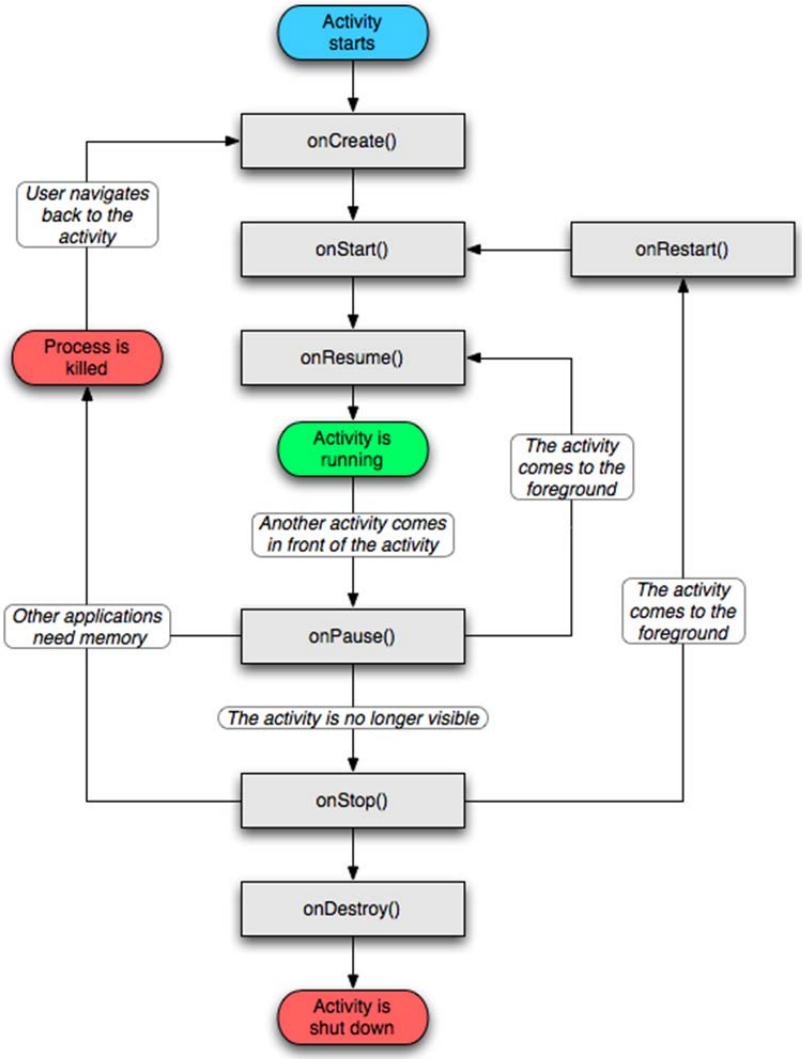


Figure 5: Activity life-cycle (copied from [25])

The call to *onPause* happens when another activity is about to be resumed or started. This is where developers usually put code to store persistent data and do other cleanup operations. The activity is now situated in state **paused**. From there it might be resumed if the activity becomes the foreground activity again, or *onStop* is called if the activity is not visible anymore. The state of the activity changes to **stopped** after *onStop* has been executed. Since the activity remains in memory until the operating system decides to kill the activity, usually due to memory shortage, it can be restarted without calling *onCreate*. In case of low memory conditions, the activity might be killed and removed from memory without calling *onDestroy*. Moreover, Android does not even guarantee to call *onStop* when an extreme low memory situation is present. [25]

The default order of method calls, when a new activity is launched, is that the *onPause* method of the current foreground activity is called, and then the methods in order *onCreate, onStart* and *onResume* from the newly started activity are executed. As a result the launched activity becomes the new foreground activity. Finally *onStop* of the previously active activity is called if it is no longer visible to the user. [25]

4.3.2 Service

A service is a component that runs in the background not having a visual user interface, very similar to a daemon in Linux. Services are used to download data from the Internet, to play music or to do any other background operation which does not require user interaction. Amarino uses a service to handle the communication with the microcontroller, so that other applications can be started without interfering with Amarino. An activity within the same application as the service is able to bind to the service and call directly methods through the interface of the background service. A service has similar but less callback methods as the activity. Indeed only three callback methods are supported per default: *onCreate*, *onStartCommand* and *onDestroy*. In addition, if the service allows other components to bind to it, three further hook methods for the binding process will be called: *onBind*, *onUnbind* and *onRebind*. [25]

4.3.3 Broadcast Receiver

A broadcast receiver is a lightweight component that just does one job, receiving broadcasted messages. A lot of messages are broadcasted by the operating system or native applications to inform other applications about system events, like changes of the battery charge level, incoming phone calls, received text messages and so forth. In Android all applications are allowed to broadcast messages and each application that registers for a specific message type (action) will receive the announcement. Amarino sends broadcasts about connection state changes or when it receives data from Arduino. This way, all interested applications can get this information if they extend the Android framework class *BroadcastReceiver* and register the receiver either during runtime or within the Android manifest file.

4.3.4 Content Provider

An application has to implement a content provider in order to share its data with other applications. In fact, there is no choice in Android if data need to be shared across third-party applications. A common content provider interface for reading, adding, changing and deleting data simplifies and standardizes the way shared data is accessed. Therefore, the actual storage implementation, whether it is a database, files or any other storage solution, must not be known by the client application and remains hidden. Android already offers some content providers to enable other applications to access contacts, pictures, videos and other type of commonly stored data on a mobile device. [25] Amarino does not require shared data and therefore does not implement a content provider.

4.4 Intents

Intents are used to activate application components and to deliver messages across and within applications. An intent is an asynchronous message sent from one application or component to another. The message is attached to an *Intent* object which is delivered by the operating system to the addressed component. Intents can be seen as a mechanism to trigger certain actions. For example there is an intent for dialing a number (ACTION_DIAL) in Android. When an application needs to call someone, all it has to do is sending an ACTION_DIAL intent and attach the number that should be dialed while leaving the hard work up to the phone application which will receive the intent and perform the desired action. If there are more components registered for a specific intent action, then the system will show a dialog to the user asking which application should handle the requested intention.

Intents are a fundamental concept of Android. They promote decoupling of application components. Intents allow developers: to seamlessly integrate components from other applications, to provide components to other applications and even to replace existing components [28]. Amarino heavily utilizes intents. On the one hand side to integrate plug-ins smoothly into the Amarino application, and on the other hand side, to broadcast its connection state as well as to forward messages received from the microcontroller to other applications.

4.5 Application Manifest

When an application is installed on a mobile device, the Android OS analyzes the attached *AndroidManifest.xml* file to get information about embedded components, required permissions, as well as other metadata about the application. A straightforward manifest file, the Amarino application manifest, is shown below.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="at.abraxas.amarino" android:versionCode="11" android:versionName="0.55">

  <application android:icon="@drawable/icon" android:label="@string/app name">

      <activity android:name="MainScreen" android:screenOrientation="portrait">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
      </activity>

      <activity android:name="DeviceDiscovery" android:screenOrientation="portrait"/>

      <activity android:name="EventListActivity" android:screenOrientation="portrait"/>

      <activity android:name="Monitoring"/>

      <service android:name="AmarinoService"/>

      <receiver android:name="RemoteControl">
         <intent-filter>
           <action android:name="amarino.intent.action.CONNECT"/>
           <action android:name="amarino.intent.action.DISCONNECT"/>
           <action android:name="amarino.intent.action.ACTION_GET_CONNECTED_DEVICES"/>
         </intent-filter>
      </receiver>

  </application>

  <uses-sdk android:targetSdkVersion="8" android:minSdkVersion="4"/>

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

  <supports-screens android:anyDensity="true"
                    android:resizeable="true"  android:smallScreens="true"
                    android:largeScreens="true" android:normalScreens="true">
  </supports-screens>

</manifest>
```

Code Snippet 1: Amarino's application manifest (*AndroidManifest.xml*)

This short example of Amarino's manifest should provide a quick overview of the structure of a usual manifest file. By looking at the *application* node it can be easily determined that Amarino consists of four activities, a background service and a broadcast receiver. A closer look at the example reveals that the structured XML file starts with the root node *manifest* declaring the namespace among the application package, which is used as a unique application identifier in Android, its version and version name. Next the application and its previously mentioned components are declared. The *icon* attribute of the *application* node links to an

image and the *label* attribute to a String resource specifying the application icon and its name.

The declaration of the *MainScreen* activity is more interesting. It says that the user interface of *MainScreen* only works in portrait mode and therefore does not provide a user interface description for the landscape mode. This causes the device not to redraw the screen when the phone is hold in landscape. The *intent-filter* node specifies which actions will launch the component. The **android.intent.action.MAIN** action marks this activity to be launched when a user hits the application icon on the home screen. The *category* node simply expresses, that this activity should be visible within the home application launcher.

The following activity nodes declare the other three screens available in Amarino. However since they do not specify an *intent-filter* they cannot be activated outside of Amarino. The service needs also to be declared in order to inform Android about its presence. The last component declared in this example manifest is a broadcast receiver which will be activated when actions such as **CONNECT**, **DISCONNECT** and **ACTION_GET_CONNECTED_DEVICES** are broadcasted.

It should be said that broadcast receivers are also allowed to specify an intent filter during runtime. However, when the intent filter is declared within the manifest, the operating system will take care of instantiating the receiver component and call its *onReceive* callback method upon receiving an action matching the intent filter.

Finally, the *uses-sdk* node tells Android which Android versions are supported and the *supports-screens* node says which screen sizes are supported by this application. The *uses-permission* nodes request two Bluetooth permissions for Amarino required for accessing the Bluetooth hardware.

The discussed nodes and attributes are only a very small portion of available declarations and properties. For a detailed description of the Android manifest please refer to the Android developers website [25], where a complete reference among many other examples can be studied.

Chapter 5

5 Arduino – Open Hardware Microcontroller Prototyping Platform

"Arduino is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. It's intended for artists, designers, hobbyists, and anyone interested in creating interactive objects or environments." [7]

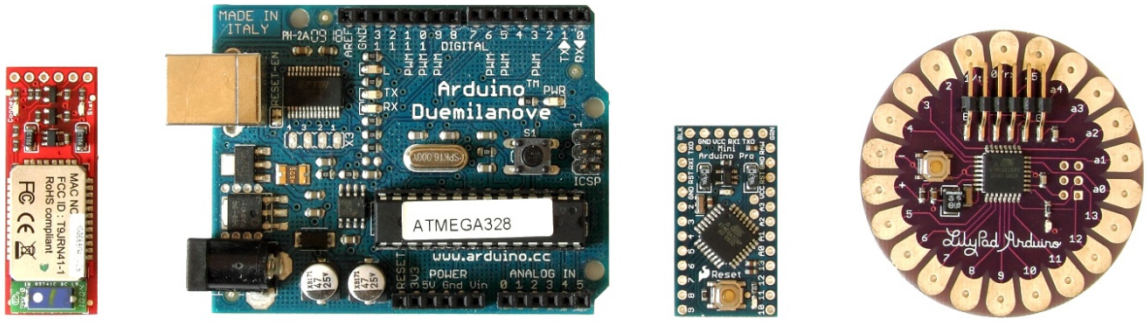


Figure 6: Selection of Arduino components
From left: Bluetooth shield, Arduino Duemilanove, Arduino Pro Mini, Lilypad Arduino

5.1 Fundamentals

In interaction design prototyping projects, where often short iterations and quick results are desired, easy to setup environments with a steep learning curve[8] can be very beneficial for the prototyping process. Microcontroller toolkits, like the

[8] quick progress in learning

Arduino, might be very supportive in such situations. Although there are some comparable platforms available besides Arduino, Arduino differs in some important aspects from its competitors [31]:

- It runs on multiple platforms, like Windows, Linux and Macintosh.
- A basic Arduino board can be purchased for about 20 Euros only, thus rather inexpensive compared to microcontroller kit alternatives.
- Arduino's IDE[9] relies on Processing [22], which is popular among artists and interaction designers for its simplicity.
- Instead of programming the board with a serial cable, it uses USB.
- As an open source product, everyone is allowed to adjust its software or even build a circuit board using Arduino's layout, without paying royalty fees to Arduino.
- There is a lively community behind Arduino providing many tutorials and offering quick answers to inquiries.
- Since Arduino was started as an educational project, novices can get used to it very quickly.

Developing microcontroller based products requires at least a programming environment as well as a microcontroller. Therefore, the Arduino platform consists of two components, the Arduino circuit board with a microcontroller on it and the Arduino IDE to write and upload programs to the board.

5.1.1 Arduino Board

The core of an Arduino board is its tiny microcontroller. Additionally, the board contains other electronics required for the microcontroller to operate correctly. Figure 6 shows a picture of three different Arduino boards; the most common Arduino Duemilanove, an Arduino Pro Mini, which is especially designed for experienced users heading for a small and inexpensive solution, and the LilyPad Arduino, aimed at wearable applications. Apart from these three randomly picked

[9] Integrated Development Environment

boards, many more variations in size, feature set and price are offered on the market. Nonetheless, they all have in common an ATmega microcontroller, albeit not necessarily the same microcontroller model in terms of speed and memory. The features of the Arduino Duemilanove, as the best selling board [23], are described in more detail.

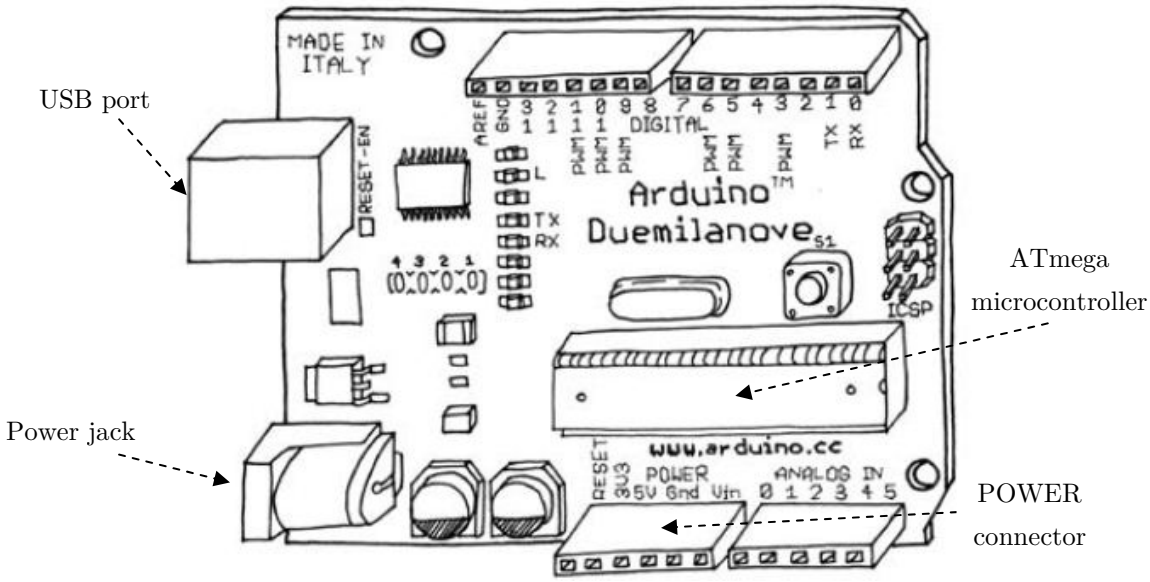


Figure 7: Sketch of the Arduino Duemilanove (copied from [31])

The Arduino Duemilanove comes either with an ATmega168 or an ATmega328, operates at 5 volts, has 14 digital I/O pins, 6 of them are pulse-wide modulated, and offers 6 additional analog input pins. The clock speed of the microcontroller is 16 MHz. Depending on the ATmega model, the flash memory has 16 or 32 KB in size for storing code, a 1 or 2 KB SRAM[10], and an EEPROM[11] with a size of 512 bytes, respectively 1 KB. The EEPROM library of Arduino allows a developer to read and write to the EEPROM in order to store persistent data even when the microcontroller is turned off. However, writing on it is rather slow, one write cycle lasts for about 3.3 ms, and the life span of an EEPROM is restricted to roughly one million write and erase cycles. [7]

[10] Static Random Access Memory

[11] Electrically Erasable Programmable Read-Only Memory

Some of the I/O pins have additional functions. Pin 0 (RX) and 1 (TX) can be used for TTL[12] serial communication. The Bluetooth shield normally connects to these pins for data transmission. Pin 2 and 3 might be used to raise interrupts and Pins 10, 11, 12 and 13 for SPI[13] communication, whereas pin 3, 5, 6, 9, 10 and 11 can be configured to be pulse-wide modulated output pins. At last, Pin 13 is directly connected to an onboard LED, convenient for testing, since no additional electronics is required to get simple visual feedback. [7]

The drawing in Figure 7 depicts a USB port and a power jack. This allows a user to power the board either via a USB cable or via an external power supply feeding between 7 and 12 volts. It is also possible to drive the board with a battery by connecting the leads from the battery to the ground (Gnd) pin and the voltage input (Vin) pin of the POWER connector located at the center bottom in Figure 7. [7]

A pre-burned bootloader allows the Arduino Duemilanove to be programmed directly via the USB port of a computer and does not require an extra hardware programmer. The Arduino IDE used to upload a program to the microcontroller is elucidated next.

5.1.2 Arduino IDE

One of the big advantages of the Arduino platform is its easy-to-use development environment which includes a text editor, a compiler and a serial monitor. After installing and starting the IDE on a computer, the window presented in Figure 8 will open. It shows a white blank text editor, a toolbar and a menu at the top. The toolbar buttons are rather minimalistic while providing the most common actions needed during development which are from left to right: verify, stop, new, open, save, upload and a button to open the serial monitor. The black area at the bottom is the output console.

The programming language in which programs (in Arduino often called sketches) are written is based on C and C++. By pressing the upload toolbar button, the

[12] Transistor-Transistor Logic

[13] Serial Peripheral Interface

IDE translates the program into C, the resulting code is compiled with the open source *avr-gcc* compiler and linked against *avr-libc* [32], which can be used on Atmel AVR[14] microcontrollers like the ATmega series. The generated assembler code is then uploaded to the Arduino board via USB.

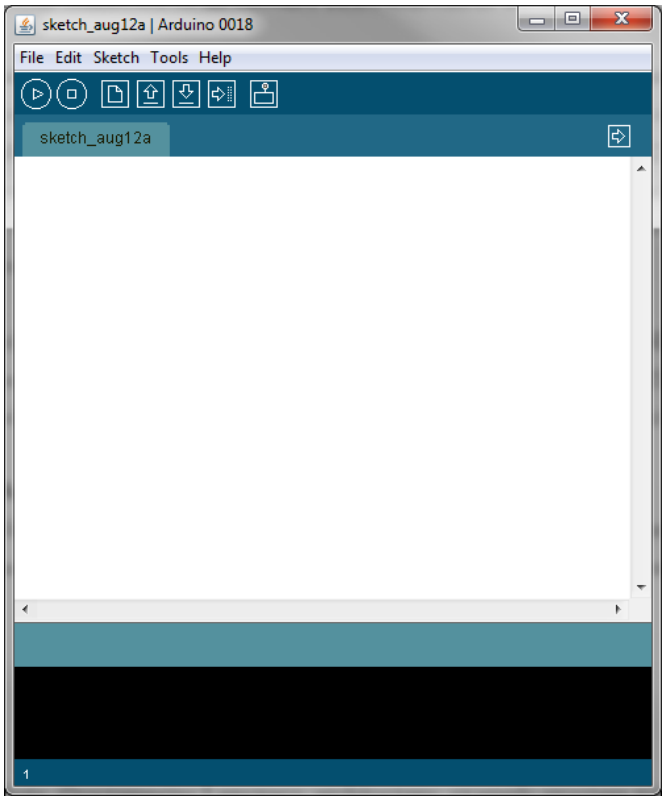


Figure 8: Arduino's IDE

The tiny example program below should help to understand the structure and code style of simple Arduino programs. The program actually sends an SOS Morse code to Pin 13.

```
1   const int ledPin = 13; // the number of the LED pin
2   const int dit = 240;   // duration of a dot in ms
3   const int dah = 3*dit; // duration of a dash in ms (three times the lenght of a dot)
4
5   void setup() {
6     // set the digital pin as output
7     pinMode(ledPin, OUTPUT);
8   }
9
10  void loop() {
11    // send an SOS signal in Morse code ( · · · – – – · · · )
12    dot();  dot();  dot();    delay(dah);
13    dash(); dash(); dash();   delay(dah);
14    dot();  dot();  dot();    delay(dah);
15    delay(7*dit); // pause between words is 7 times dit in MORSE code
16  }
```

[14] 8-bit RISC microcontroller

```
17
18  void dot() {
19    digitalWrite(ledPin, HIGH);
20    delay(dit);
21    digitalWrite(ledPin, LOW);
22    delay(dit);
23  }
24
25  void dash() {
26    digitalWrite(ledPin, HIGH);
27    delay(dah);
28    digitalWrite(ledPin, LOW);
29    delay(dit);
30  }
```

Code Snippet 2: An Arduino Morse code example, which sends an SOS signal to a LED attached to Pin 13

Almost every Arduino program has a *setup* and a *loop* function. The *setup* function is called once right after the microcontroller comes to life and the *loop* function is executed over and over again until someone turns the power off. In *setup*, developers normally do all the initializations and configurations like setting each involved pin to operate either as input or output.

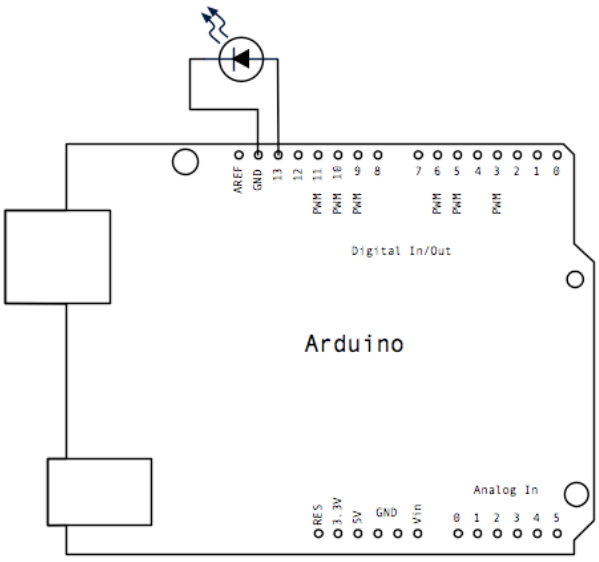


Figure 9: Schematic for the SOS Morse code example
(Schematic is equal to the blink example schematic and therefore borrowed from [7])

After the *setup* function is finished, the *loop* function is called. In the code example above, *loop* invokes the *dot* and *dash* functions defined below the *loop* function. The *dot* function sets Pin 13 to HIGH, means 5 volts, waits 240 milliseconds (ms), and sets Pin 13 back to LOW, zero volts. This has the effect of turning the attached LED on for 240 ms. Similarly does the *dash* function, however turning the LED on for 720 ms. At the end of either function, *dot* and *dash*, a pause of one dit (240ms) is added to separate each signal.

The *digitalWrite*, *pinMode* and *delay* functions, which were used within the Morse code example, are part of the Arduino programming language among many other useful functions like *digitalRead*, *analogWrite*, *min*, *max* or *random*, or constants like *HIGH, LOW* and *OUTPUT*, just to mention a few. The full reference of the Arduino language can be found online at [7].

5.2 Extensibility

The Arduino language reference offers some helpful functions per default. However for complex tasks such as controlling servo or stepper motors, LCDs[15] or for wireless communication, more elaborated functions are required. When controlling hardware, a library might also be referred to as a driver. A library's responsibility is to offer simple to use extra functions for specific duties while hiding intricate implementation details.

The Arduino IDE is delivered with a selection of commonly used libraries but also supports integration of third-party libraries. Figure 10 shows all preinstalled libraries of the latest Arduino version (v. 0018). At the end of the list, the MeetAndroid library is marked as a contributed (third-party) library.

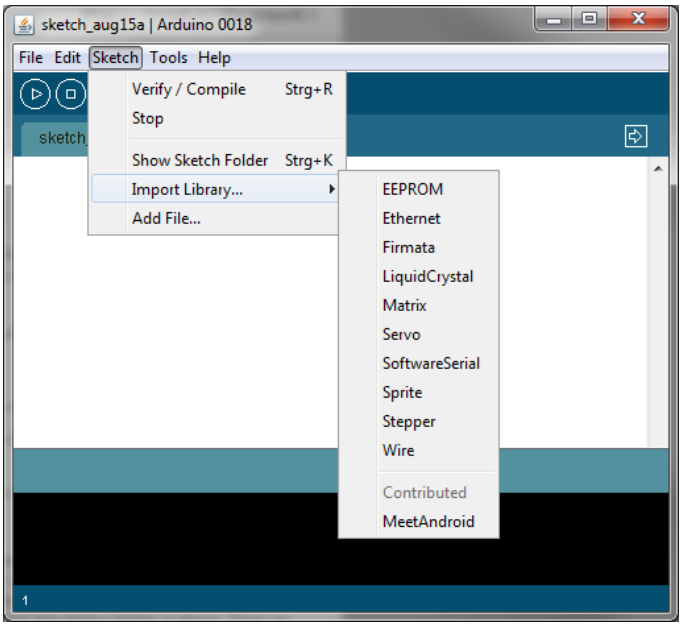


Figure 10: Arduino IDE Menu: "Sketch→Import Library…" lists available Arduino libraries

[15] Liquid Crystal Displays

To get an external library integrated into Arduino's IDE, at least one header and a source file (.h and .cpp file) have to be written and put into a directory named after the library. This directory must be copied into the *libraries* sub-directory of Arduino's user directory. Afterwards, the new library will appear in the "*Sketch→Import Library…*" menu and is ready to be included and used within an Arduino program.

In addition, a library developer can attach examples to the library in order to demonstrate the use of the library. If the Arduino IDE finds an *examples* directory inside the library directory with example sketches, the menu of the IDE will list the library examples at *"File→Examples"* just as shown in Figure 19 (on p. 46) for the MeetAndroid library. Moreover, a *keywords.txt* file defining the library's keywords could be put into the directory of the library to enable keyword highlighting for the library.

5.3 Community

One reason why Arduino has been chosen for Amarino over other microcontroller platforms is its active fast growing community. Indeed, in less than two month, from June to August 2010, the number of registered users increased by 9 percent to over 28,000 while more than 256,000 contributions were counted, a plus of 11 percent. Since most people of the Arduino community share the open source philosophy of Arduino, they are diligent about publishing do-it-yourself tutorials, libraries and example code.

Figure 11: Selection of Arduino projects [33]
From left to right: Laser Harp, UAV, Game Boy, Botanicalls and the Star Wars Force Trainer
(First three pictures taken from [33], next from [34] and the rightmost from [35])

The Arduino platform motivated designers, researchers and tinkerers to produce imaginative projects some of which turned into commercial products. The HacknMod website [33] lists a selection of forty outstanding Arduino projects published on the Web. Among a laser harp, which plays sounds by touching the

light beams, an UAV[16] spy plane and an open source game boy, also commercialized products like the Botanicalls [34] and the Star Wars Force Trainer [36] can be found.

The Botanicalls Kit is an internet-enabled electronic leaf which, plunged into a potting soil, will report via Twitter the plant's water level or its current feelings. The Force Trainer is a brain-controlled toy used to levitate a ball within a tube by measuring brainwaves through dry EEG[17] sensors.

[16] Unmanned Aerial Vehicle

[17] Electroencephalography – recording of electrical activity within the brain [http://en.wikipedia.org/wiki/Electroencephalography]

Part II

Implementation of a Toolkit for the Rapid Prototyping of Mobile Ubiquitous Computing

Chapter **6**

6 Amarino Software Toolkit

Amarino is a software toolkit with two main software components: the Android application and an Arduino library. In addition, Amarino provides an Amarino API and an online documentation. The toolkit might either be used in prototyping projects where sensors, tangible or wearable devices have to communicate with a phone, or where the phone is utilized to control everyday things wired to the microcontroller.

As illustrated in Figure 12, Amarino works as a communication channel allowing transparently sending and receiving data via Bluetooth between a phone and a microcontroller in order to communicate with devices attached to the microcontroller.

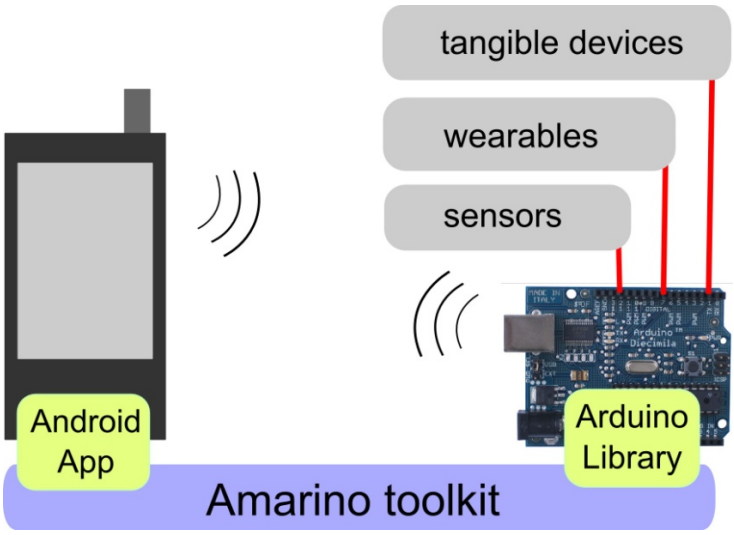


Figure 12: Smartphone talks to Arduino which talks to various other devices

Amarino hides raw data transmission by employing an event-based communication infrastructure. The Arduino library has a callback mechanism for incoming events. A developer registers one callback function for each expected event type. The Arduino library will call the registered function as soon as an event has been received and procures convenient functions for a developer to extract data attached to the event. The library offers also functions to send data from the microcontroller to the phone.

The Android application comes with a graphical user interface which allows a user to manage Bluetooth connections, create a set of events which should be sent to Arduino and monitor the data stream for debugging reasons. For many projects it is sufficient to get sensor readings from the phone's accelerometer, light sensor, compass or whatever sensor a phone provides. The Amarino Android application offers access to most available phone sensors plus several other internal phone events (Table 3, on p. 44). This means a user can select for instance the compass sensor event using the graphical user interface of Amarino and when the connection between the phone and the microcontroller has been established, the phone periodically sends compass events to the microcontroller.

The subsequent part of this chapter delineates Amarino from a user's point of view, starting with a use case scenario briefly unfolding the development steps necessary to realize a project with Amarino. This is followed by a detailed look into the Android application, the MeetAndroid Arduino library and the Amarino API. Finally, the documentation, as an important part of the Amarino toolkit, will be adumbrated.

6.1 Use Case Scenario

This following rather simple and yet idealistic use case scenario outlines the development process when prototyping with Amarino. It gives a rough schema of how Amarino might be approached.

Idea

Alice has an idea. She wants her handbag to react whenever she receives a text message on her phone. She thinks that it would be really cool if the handbag could vibrate and blink when one of her friends sends her a text. Alice searches the

internet for instructions how she could achieve her goal. She finds the Amarino website [37] and decides to use this toolkit to implement her idea. The website mentions that she has to have an Arduino based microcontroller in order to realize her idea. Alice searches on the Arduino website [7] for solutions, stumbling over the LilyPad Arduino [12], a microcontroller board especially designed for the use with e-textiles and wearables.

Preparation

She orders one LilyPad board, conductive yarn, a sewable vibration motor, some sewable LEDs, a battery and a Bluetooth shield; all in all it will cost her about 50 Euros. After Alice went through some online tutorials about how to use the LilyPad she feels comfortable to start developing her own project. She installs the Amarino application on her phone and adds the Amarino library for Arduino (MeetAndroid) to her Arduino development environment. By going through an online "getting started" tutorial for Amarino, which she found on Amarino's website, she becomes familiar working with the toolkit.

Development steps

The following short version of the "getting started" tutorial shows what needs to be done before Arduino can receive an event from her Android phone:

1. Discover and pair the Bluetooth shield with the phone
2. Select one or more events in Amarino to be transmitted to the microcontroller
3. Write an Arduino program
    a. Import Amarino's Arduino library called MeetAndroid
    b. Register a callback function for each event
    c. Implement the callback functions
4. Upload your program to Arduino
5. Hit the connect button in Amarino's Android application

Hands-On

Alice attaches the Bluetooth shield and the battery to her LilyPad and switches on the microcontroller. Next, she uses Amarino to discover and pair the Bluetooth

device. Alice wants her phone to inform the LilyPad whenever she receives a text message. Consequently, she selects the *Receive SMS* event in the Amarino application on her phone. Afterwards she writes some code in the Arduino programming environment for the LilyPad to light up some LEDs and to activate a tiny vibration motor when the events is received. After Alice uploaded her Arduino program to the LilyPad microcontroller she attaches the LEDs and the vibration motor to the LilyPad. Everything seems to be fine and Alice starts to connect the mobile phone and the LilyPad by hitting the connect button within the Amarino application. A status icon tells her that connection is up and the phone is ready to transmit events.

Test
Since Alice wants to see the LilyPad in real action, she asks her mom to send her a text. Just a second after her mom sent her a text message, lights go crazy and the vibration motor shakes. Alice is happy with the result and decides to sew the components onto her bag.

6.2 Android Application - Amarino

The Android application of Amarino can be seen as the window to Amarino, since it is the user interface of the Amarino toolkit. The Android application has three main modules to administer handling with Bluetooth devices:

- Bluetooth Device Manager
- Event Manager
- Monitoring

Each of the modules is demonstrated from a user's point of view as you would approach them when using Amarino for the first time. I like to mention that a user of Amarino is most likely also a developer, since Amarino is all about developing prototypes.

6.2.1 Bluetooth Devices Manager

Once a user has installed and started the Amarino application on the phone, the left screen as shown in Figure 13 is presented to the user. Before a user can start of, at least one Bluetooth device must have been added to Amarino. Touching the

Add BT Device button at the bottom of the screen immediately initiates Bluetooth device discovery, and a list of discoverable nearby Bluetooth devices is appearing on the screen (center screenshot Figure 13). In case Bluetooth is disabled on the phone, Amarino takes care of enabling it automatically before device discovery starts.

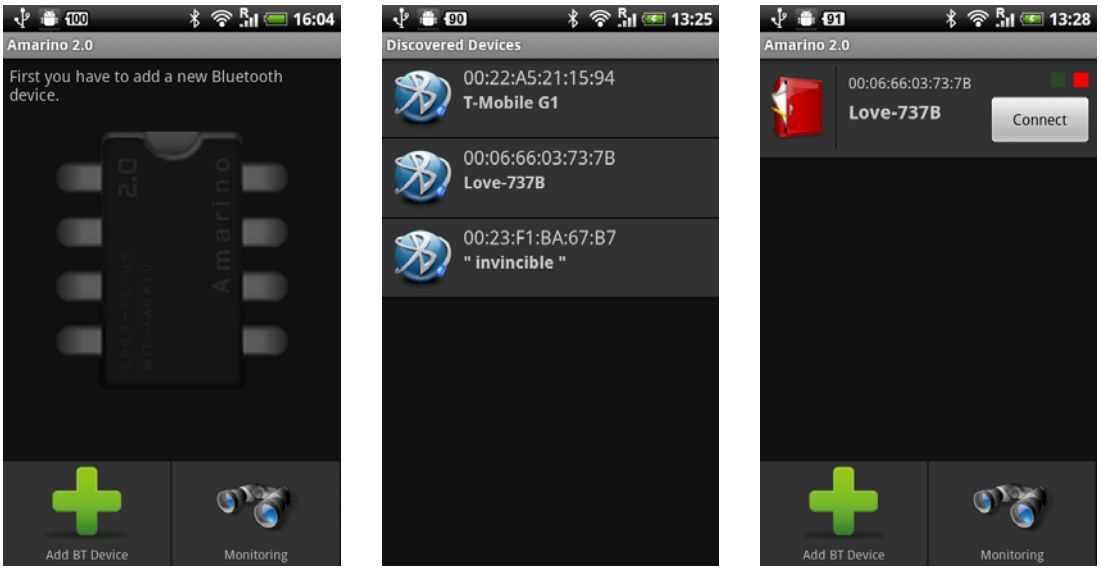


Figure 13: Screenshots of Amarino's user interface
left: virgin start screen | center: discovered Bluetooth devices | right: added Bluetooth device

If the user has correctly configured and assembled an Arduino with a Bluetooth shield, one of the discovered Bluetooth devices should be the Bluetooth module attached to the Arduino. By taping the corresponding list entry, the device gets added to Amarino. As a result the added device shows up on the start screen (Figure 13, right).

Amarino has been designed to be able to work with several Bluetooth devices in parallel. Thus, a user could repeat these steps in order to add more devices to Amarino. In Figure 14, three devices have been added. The first device is connected while the two on the bottom are disconnected. A user could connect the other two devices as well and Amarino would handle the communication to all connected devices virtually simultaneously.

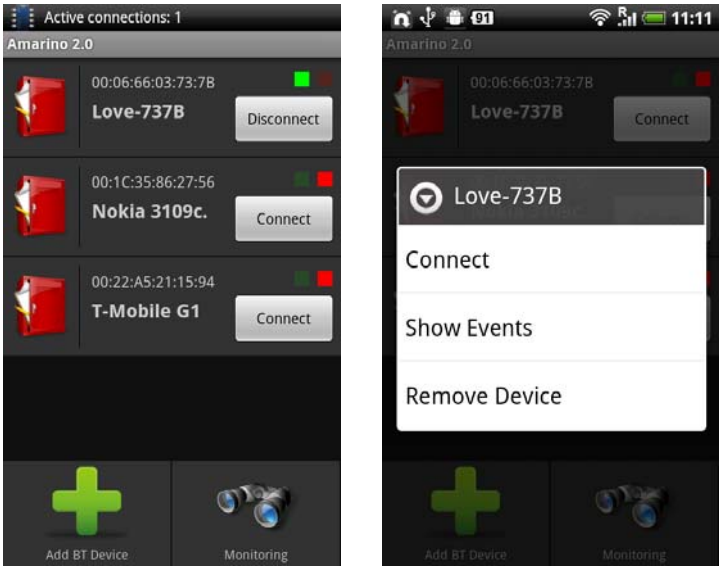


Figure 14: Amarino is able to manage several BT devices in parallel (left)
The context menu popping up when long-pressing a disconnected device (right)

A list entry on the start screen represents a unique Bluetooth device stating its address and its user-friendly name. In addition, each list entry indicates the connection state between Amarino and the Bluetooth device via a tiny green/red indicator on the top right corner. The connection state can be changed by hitting the connect button on the right. The outer left red document locker icon directs the user to the event manager for that device. Long-pressing a device brings up a self-explanatory context menu with three options: *Connect* respectively *Disconnect*, *Show Events* and *Remove Device* (Figure 14, right). Added devices are stored persistently in Amarino's internal database and will stay put when the application is restarted after it was exited. A device can be removed from Amarino by selecting *Remove Device* from the context menu as described previously.

An important feature of this toolkit is that Amarino maintains active connections even if the user interface has been closed. This is because connections to Bluetooth devices are solely managed by Amarino's background service, allowing a user to start other applications on the phone while Amarino's communication to connected devices remains active without interfering with third-party applications. In order to reduce unnecessary memory consumption and processor time, hence longer battery life, the background service runs only if there are active connections, otherwise it will be killed when leaving Amarino's user interface.

Since many scenarios intend Amarino to run in the background, Amarino employs notifications to inform the user about actions done by the background service. Due to the reason that communication is always managed by the background process, even when the application is visible, notifications are also in place when dealing with the user interface. To illustrate the use of notifications, Table 2 shows different notification messages when first connecting and then disconnecting from a device by user intention.

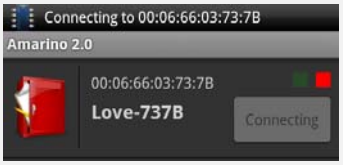
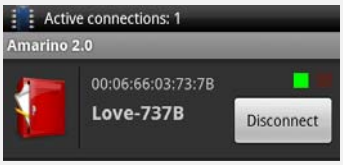
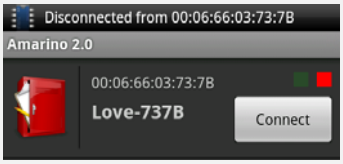
1. User hits the connect button. 2. The connect button becomes disabled and the text changes to *Connecting*. 3. A notification message appears in the status bar informing the user about the progress.	
4. Connection successfully established. 5. The connect button becomes enabled and the text changes to *Disconnect*. The green light is bright. 6. The status bar shows a notification indicating the number of active connections.	
7. User taps the disconnect button. 8. The tapped button reverts back to *Connect* and the red light signals the disconnected state. 9. A notification in the status bar informs the user about the disconnection.	

Table 2: Amarino's background service uses notifications to indicate state changes

It should be emphasized that notifications are also updated when Amarino's user interface is closed. In fact, this is the main purpose of the notification feature, to give users unobtrusive up to date information of connection states. Perhaps another process, whether it is a third-party user interface or another background service, will call the Amarino API to connect or disconnect from Bluetooth devices, then Amarino would also provide feedback about state changes to the user via its notification service.

Figure 15: When Amarino is running in the background, the Amarino icon is visible in the status bar (left) Extending the status bar exposes the last notification message from Amarino (right)

Once Amarino's background service is running, the Amarino icon is visible on the status bar (Figure 15, left). This way a user is always aware of a running background service. By pulling the status bar down, the last notification message of Amarino is revealed. For convenient access, tapping the status bar entry will start Amarino's user interface, no matter which application is in foreground at that moment.

Now we know how to discover, connect and disconnect external Bluetooth devices and which status messages to expect, next I introduce the event manager which is all about sending data from the phone to the microcontroller.

6.2.2 Event Manager

Let us assume a developer wants to send data from Android to Arduino, but does not want to write any code for Android (the developer has still to write some code for Arduino to receive data, though), the event manager module is where to go. As mentioned in the previous subsection, pressing the red locker icon directs the user to the event manager for the particular device. Alternatively, the user could also access the event manager via the context menu of a device. When the user enters the event manager for the first time, he will see an empty screen (Figure 16, left), because no events have been added to the associated device yet. Consequently, the

next step is to hit the *Add Event* button. In an instant, a dialog box pops up with a list of available events (Figure 16, center).

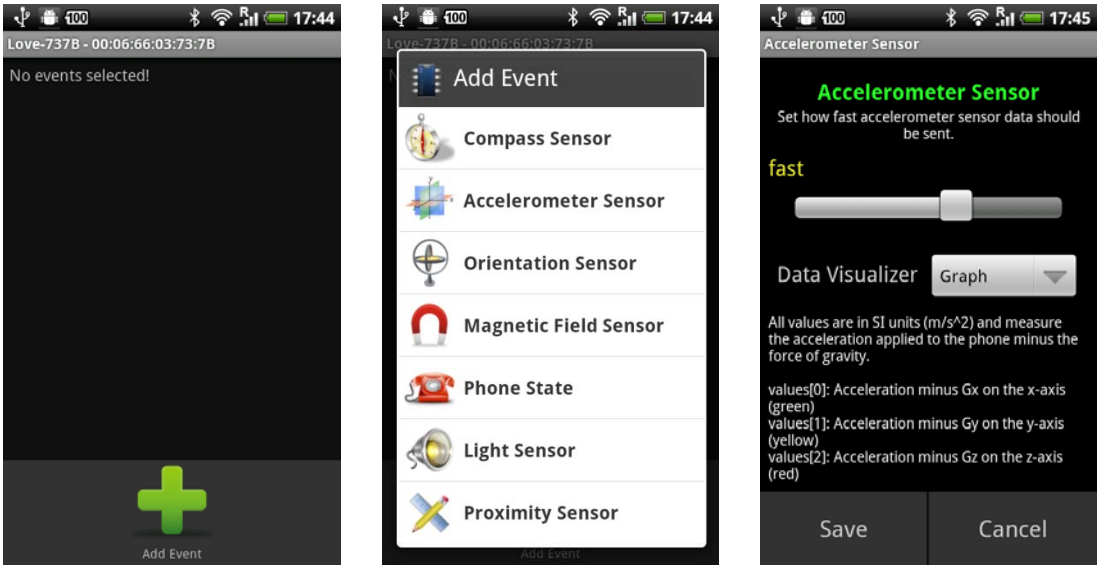


Figure 16: Screenshots of the event management module
left: empty event manager screen | center: dialog for adding an event | right: configuration screen for the accelerometer sensor event

When a user adds an event to a certain Bluetooth device, then the device will receive messages described by this event, once a Bluetooth connection has been established. Table 3 lists all events which are part of the *Amarino Plug-in Bundle* shipped with Amarino. The list of events can be extended by developing Amarino Plug-ins which is covered in detail in Chapter 8.2.

After a user has selected an event, the configuration screen opens up. In Figure 16 the *Accelerometer Sensor* event has been selected and its configuration screen is shown on the right hand side of Figure 16. The *Accelerometer Sensor* configuration screen offers the option to change the frequency of messages to be sent, the type of data visualization (graph, bars, text) and, in addition, it displays detailed information about the *Accelerometer Sensor* event.

Plug-In Bundle Events	Description
Compass Sensor	Sends heading in degrees [0-359]
Accelerometer Sensor	Sends acceleration data in m/s2 [x,y,z]
Orientation Sensor	Sends orientation data in degrees [azimuth, pitch, roll]
Magnetic Field Sensor	Sends magnetic field data in micro-Tesla [x,y,z]
Phone State	Sends a message when the phone state changes [IDLE, RINGING, OFFHOOK]
Light Sensor	Sends a message when the ambient light level changes in SI lux units
Proximity Sensor	Sends the distance to the proximity sensor in centimeters
Battery Level	Sends the current battery level as soon as the phone's battery level changes
Time Tick	Sends a message every minute containing the actual minute
Test Event	Sends a test message every 3 seconds with random data [0-255]
Receive SMS	Forwards a received SMS to Arduino as String [only first 30 characters are sent]

Table 3: Events available after installing Amarino's Plug-in Bundle

By confirming the configuration screen, the event is added to the event manager (Figure 17, left) and set up to transmit data to Arduino. But it does not start to send data immediately, because the toolkit controls the life cycle of added events. When a connection to a Bluetooth device is established, Amarino starts all associated events, respectively disables them when disconnected. Tapping the event entry will always bring the configuration screen back to permit configuration changes or just to read the description again.

Another feature of the event manager is to display active events in real-time. Amarino offers three different visualizers: graph, text and bars. The visualizers are demonstrated in the same order from top to bottom in Figure 17, the center screen shows disabled events and the right screen displays events in action. The left screen shows the state before the *Test Event* and the *Orientation Sensor* event were added.

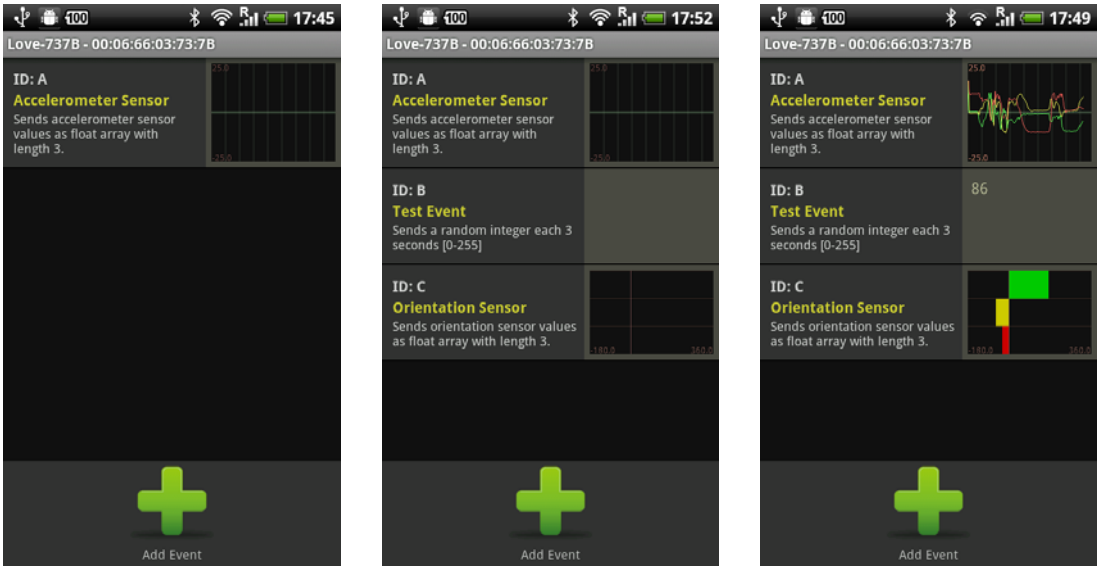


Figure 17: Event Manager with events added to it
left: Accelerometer Sensor event added | center: three events are added to device Love-737B | right: events are enabled displaying their data in real time using Amarino's visualizer

Visualization of transmitted data should provide feedback and aid users in debugging their projects, especially since debugging communication with microcontrollers might be hard due to their restricted user interface facilities.

In order to test event functionality, the event manager allows manually enabling and disabling events. These options become visible when long pressing an event entry. Enabling an event by hand will display the data in real-time, but Amarino will not forward the messages to Arduino, as the Bluetooth device might not be connected.

6.2.3 Monitoring

Another facility a user might consult for debugging reasons is the monitoring module (Figure 18), which can be accessed from the start screen by hitting the *Monitoring* button. The monitoring screen displays Amarino's logging data, error and debug messages as well as information about received and sent data.

The monitoring module can also be used to manually send data to Arduino. By selecting the flag (event ID) and entering a message, a user can construct a custom message and send it to a connected Arduino. The meaning of a flag becomes clearer during the forthcoming chapters.

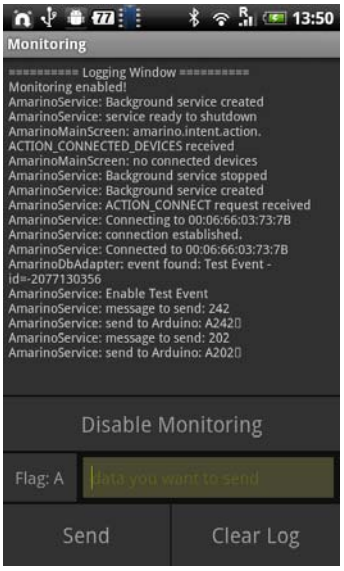


Figure 18: Behind the curtain - monitoring Amarino

6.3 Arduino Library - MeetAndroid

The MeetAndroid library for Arduino is the counterpart of the Android application. It provides convenient functions to develop programs for Arduino to receive and send data using Amarino's communication channels.

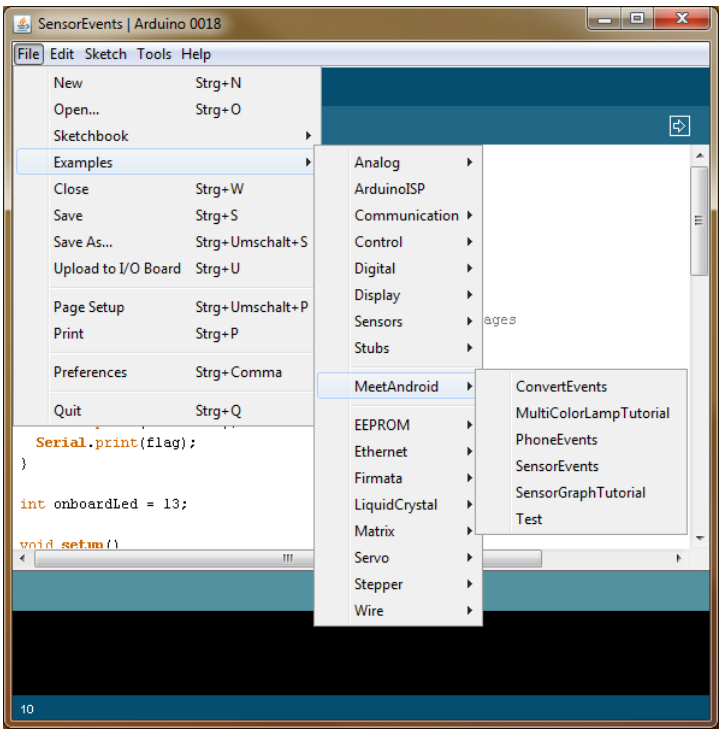


Figure 19: Once the MeetAndroid library is installed, its examples appear in the Arduino IDE example section

Moreover, it comes with some examples illustrating the use of the library. The examples can be found after installing the library within the regular Arduino IDE examples folder (Figure 19).

In Code Snippet 3 an Arduino example is shown which receives data from a compass event. The compass event sends exactly one integer value per message; the heading of the phone. In our example the compass event has the ID 'A', which would be the case if only one event has been added to a Bluetooth device. Messages sent from Amarino are flagged with an event type identifier (ID), in order to distinguish the origin of the data in MeetAndroid. The library ensures to invoke the callback function registered with the ID. The following example will illustrate this process in detail. The demonstrated sample code is also part of Amarino's documentation and comes pre-installed with the MeetAndroid library helping users to quickly understand the usage of library functions.

```
/*
 * Receives compass sensor events from your phone.
 */
#include <MeetAndroid.h>

MeetAndroid meetAndroid(error);

void error(uint8_t flag, uint8_t values){
  meetAndroid.send("ERROR: ");
  meetAndroid.send(flag);
}

void setup() {
  Serial.begin(57600);
  meetAndroid.registerFunction(compass, 'A');
}

void loop() {
  meetAndroid.receive(); // you need to keep this in your loop() to receive events
}

/*
 * Compass events are sent several times a second.
 * therefore this method will be called constantly.
 *
 * note: flag is in this case 'A' and numOfValues is 1
 * since compass event sends exactly one single int value for heading
 */
void compass(byte flag, byte numOfValues) {
  // we use getInt(), since we know only data between 0 and 359 will be sent
  int heading = meetAndroid.getInt();

  doSomethingUseful(heading);
}

```

Code Snippet 3: An Arduino example using the MeetAndroid library to receive compass sensor events

By going briefly step by step through the code it will be exemplified what needs to be done in order to receive compass events from the phone.

After the library has been included (4), the *MeetAndroid* object is declared and constructed (6) while an error function (8) is passed to it, which will be called when messages are received for which no callback function has been registered. The *setup()* function (13), which is called only once just before the *loop()* function (18), initializes the serial port, respectively the Bluetooth module, with baud rate 57,600 (14) and registers a callback function (15), named *compass* (29), to be called when events with ID 'A', compass events, are received. All callback functions registered to receive events must match the signature as shown in (29). The *loop()* function (18), called over and over again until the Arduino is reset or switched off, has only one function to execute, the *receive()* function (19) of the MeetAndroid library. This is where all the magic happens. The *receive()* function checks if there are serial data available, analyses the received raw data, if there are any, and invokes the registered callback function for a given message from Amarino. To retrieve the compass data within a callback function, the MeetAndroid library offers some useful functions. The example uses the *getInt()* function (31) to extract the received integer value, the heading, and assigns it to a variable called heading.

Besides the mentioned *getInt()* library function, the MeetAndroid library offers similar functions for all other data types to be expected from Amarino (Figure 20). Examples included in the MeetAndroid library demonstrate the use of the other functions, too.

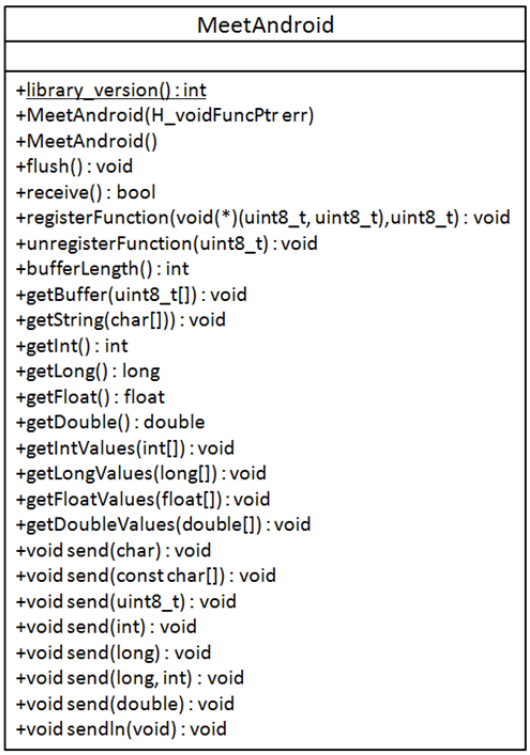


Figure 20: Available functions of the MeetAndroid library

Furthermore, the library provides functions to send data to the phone as well as functions to retrieve the serial byte buffer, if a developer wants to access received raw data.

6.4 Amarino API

Events as discussed in Section 6.2.2 have some limitations; they allow background services to send messages but they do not have access to the screen of the phone in order to display received data or to enable the user to interact with the touch screen. Therefore, whenever interaction with a user interface is required, or if a developer wants to display data received from Arduino on the screen, a separate application needs to be developed. Luckily, for this reason Amarino offers an Application Programming Interface (API), supplying various methods to third-party Android applications in order to support communication with an Arduino microcontroller.

Communication with an Arduino Bluetooth device involves four different actions:

- Connect to Arduino

- Disconnect from Arduino
- Send data to Arduino
- Receive data from Arduino

Figure 21 shows the two components constituting the Amarino API. An *Amarino* class with a selection of static methods and the *AmarinoIntent* interface which is a bunch of constants required to use the API.

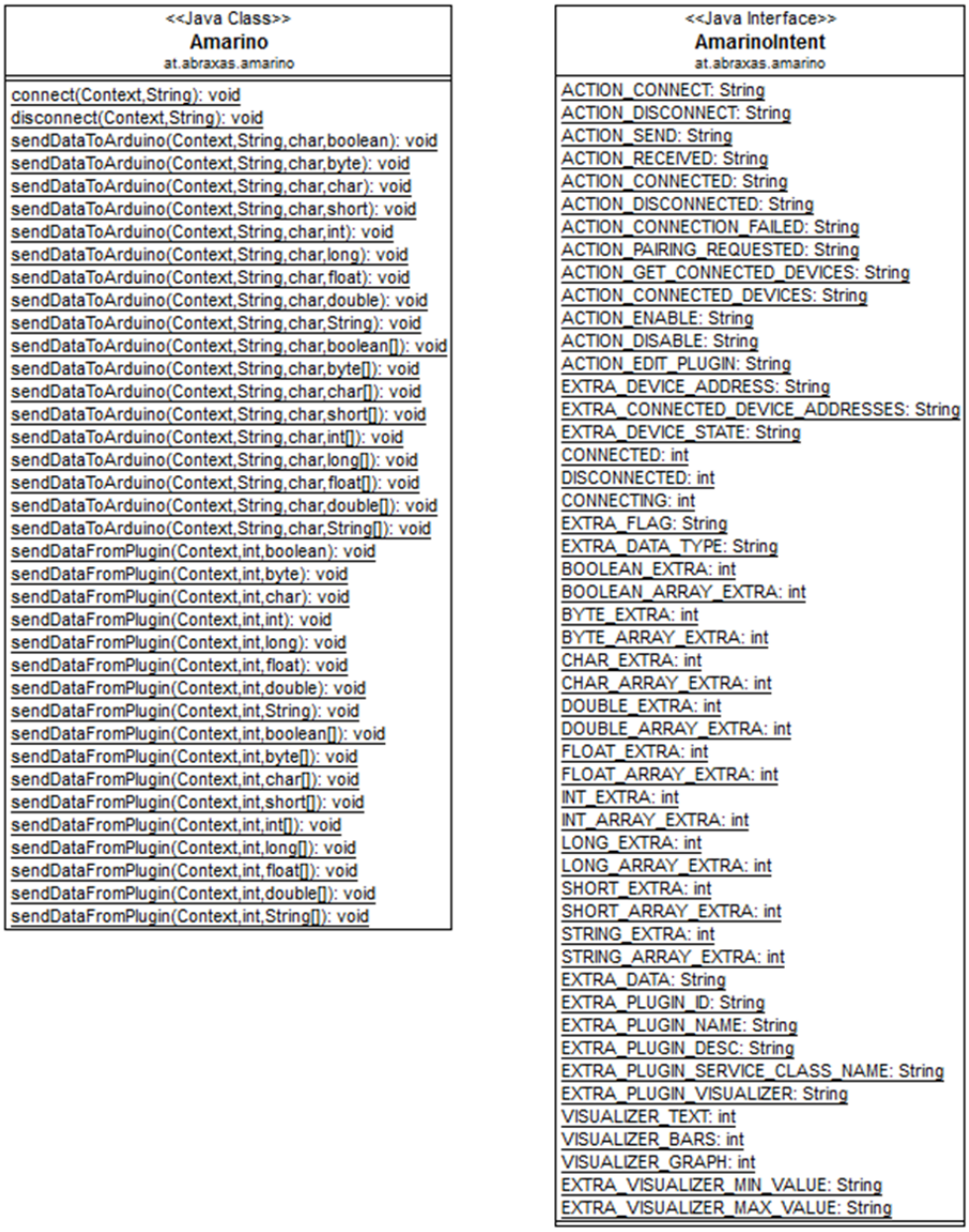


Figure 21: Amarino API

How to connect and disconnect

To connect or disconnect from or to a Bluetooth device, a developer simply calls the methods *connect* or *disconnect* of the Amarino class and passes the context object and the device address into it (Code Snippet 4 line 7 and 15).

```
1  private static final String DEVICE_ADDRESS =  "00:06:66:03:73:7B";
2
3  @Override
4  protected void onStart() {
5    super.onStart();
6
7    Amarino.connect(this, DEVICE_ADDRESS);
8  }
9
10 @Override
11 protected void onStop() {
12   super.onStop();
13
14   // if you connect in onStart() you must not forget to disconnect in onStop()
15   Amarino.disconnect(this, DEVICE_ADDRESS);
16 }
```

Code Snippet 4: Connect and disconnect using the Amarino API

As you can see, calling *connect* and *disconnect* is rather easy, however how does a third-party application know if a launched command really resulted in a connection state change of the Bluetooth device? It can catch Amarino's broadcasted intents to get this information, since Amarino signals connection changes by broadcasting intents to third-party applications.

How to get feedback about connection state changes

The following Code Snippet 5 demonstrates a *BroadcastReceiver* template which receives intents from Amarino about connection state changes.

```
1  public class ConnectionStateReceiver extends BroadcastReceiver() {
2
3    @Override
4    public void onReceive(Context context, Intent intent) {
5      final String action = intent.getAction();
6
7      // this is how to retrieve the address of the sender
8      final String address = intent.getStringExtra(AmarinoIntent.EXTRA_DEVICE_ADDRESS);
9
10       if (AmarinoIntent.ACTION_CONNECTED.equals(action)){
11         // connection has been established
12       }
13       else if (AmarinoIntent.ACTION_DISCONNECTED.equals(action)){
14         // disconnected from a device
15       }
16       else if (AmarinoIntent.ACTION_CONNECTION_FAILED.equals(action)){
17         // connection attempt was not successful
18       }
19       else if (AmarinoIntent.ACTION_PAIRING_REQUESTED.equals(action)){
20        // a notification message to pair the device has popped up
21       }
22   }
23 }
```

Code Snippet 5: Template of a BroadcastReceiver receiving connection state messages from Amarino

How to send data to Arduino

Sending data to Arduino can be done in one single line of code (Code Snippet 6 line 4), assuming that a connection has been properly established before.

```
1  final char flag = 'a';
2  final String message = "Amarino rocks!";
3
4  Amarino.sendDataToArduino(this, DEVICE_ADDRESS, flag, message);
```

Code Snippet 6: Send data to Arduino

The *sendDataToArduino(…)* method asks for a context object, the device address, the flag which is used to identify the sender, and the data to be sent. As apparently can be recognized in Figure 21 (on p. 50), the Amarino API heavily uses polymorphism to ease sending of different data types, even arrays.

How to receive data from Arduino

Receiving data works very similar to receiving connection state changes, again a *BroadcastReceiver* is needed. But this time not to catch connection state intents, but Amarino's **ACTION_RECEIVED** intents.

```
1  public class DataReceiver extends BroadcastReceiver {
2
3    @Override
4    public void onReceive(Context context, Intent intent) {
5
6      if (AmarinoIntent.ACTION_RECEIVED.equals(intent.getAction())){
7
8        // the device address from which the data was sent,
9        // we don't need it here but to demonstrate how you retrieve it
10       final String address = intent.getStringExtra(AmarinoIntent.EXTRA_DEVICE_ADDRESS);
11
12       // the type of data which is added to the intent
13       final int dataType = intent.getIntExtra(AmarinoIntent.EXTRA_DATA_TYPE, -1);
14
15       switch (dataType){
16
17         case AmarinoIntent.STRING_EXTRA:
18         String data = intent.getStringExtra(AmarinoIntent.EXTRA_DATA);
19         // do something useful with the received data
20         break;
21       }
22     }
23   }
24 }
```

Code Snippet 7: BroadcastReceiver to receive data from Arduino

When Amarino receives data from Arduino it attaches the received data to an **ACTION_RECEIVED** intent and broadcasts it so that interested parties can catch the intent and extract the piggybacked data. Code Snippet 7 represents a *BroadcastReceiver* that extracts data from an **ACTION_RECEIVED** intent.

Many methods and constants of the Amarino API are dedicated to plug-in developers and will be discussed farther down in Section 8.2. A complete documentation of the Amarino API is published online at Amarino's website [37].

6.5 Documentation

For any toolkit, documentation is obligatory and Amarino is not different in that respect. I would even go one step further and say documentation is the crucial part for the success of a toolkit and certainly developers will appreciate well documented toolkits.

Amarino's documentation is based on five columns:

- Online tutorials
- Code Examples
- Javadoc
- Source Code
- Discussion forum

The entire documentation is available on Amarino's website [37], whereas the source code is hosted by Google code [38] under GNU General Public License v3 [39] and the discussion forum is on Google groups [40].

Online tutorials

On Amarino's website [37] users find a "getting started" tutorial and a tutorial about plug-in development. The "getting started" tutorial leads a newcomer in ten steps to a successful data connection between an Android phone and an Arduino microcontroller. In contrast, the plug-in development tutorial addresses advanced developers already familiar with Android, guiding them also in a step-by-step manner through the implementation of a fully functional Amarino Plug-in.

Code examples

Beside online tutorials, a complete working project can be downloaded from the website. Developers will also find pre-installed practical Arduino code examples within the Arduino library as mentioned already in Section 6.3.

Figure 22: Home page of Amarino's website

Javadoc

For the Amarino API discussed in Section 6.4, a Javadoc documentation has been generated exhaustively describing and exemplifying the use of the Amarino API. The Javadoc documentation is available online on Amarino's website [37].

Source code

Amarino is open source; hence its sources are accessible to all interested developers. The license [39] under which Amarino is published enables developers to modify and reuse the toolkit if they like. Often the source code can provide deep understanding of the software and therefore it is a very useful part of the documentation.

Discussion forum

Many questions regarding the use of the toolkit are of the same nature. Instead of answering each inquiry directly, I set up a discussion forum where users of Amarino can post issues or contribute by providing solutions for Amarino related issues. It is the first place to ask for support when questions arise with regard to Amarino. Issues related to the sources of Amarino can be opened and discussed via a Google hosted issue tracker [38].

Chapter 7

7 Architecture Overview

The preceding chapter described Amarino from a user's point of view. This chapter digs deeper and reveals the architecture Amarino is based on. The role of each major component is explained and its relation to other components of Amarino is depicted.

7.1 Design Rationale

Before going into details about the architecture of Amarino, the design rationale for Amarino is discussed. Jarczyk et al. [41] emphasize the importance of having a design rationale in software engineering. According to the researchers a design rationale should at least list which decisions drove the design process and it should explain the motivation behind those design choices.

When I started designing Amarino, my first decision was to build a prototype in order to elaborate if the proposed concept is technically feasible. Upon completion of the first usable Amarino prototype I held workshops to gain feedback from workshop participants and I talked to other users who tested the Amarino prototype without guidance. Thanks to the insights during the prototyping phase, I had a much clearer picture in mind about the user experience people would like to have, which features they use most, the technical possibilities and what kind of stumbling blocks to expect during the development of the final product. The initial prototyping process led to five important design decisions. Each listed decision (in arbitrary order) had a significant impact on the architecture of the toolkit. The motivation behind these decisions is briefly discussed below.

Decision 1 – Amarino shall be able to run as a background task (unobtrusive)
Smartphones are used for different tasks, calling, texting, browsing and so on, for that reason Amarino should not interfere with the regular operation of a smartphone, hence should run in the background most of the time.

Decision 2 – Interfaces for third-party applications shall be provided (extensible)
Other developers might want to extend Amarino's collection of events or even implement foreground applications to interact with microcontrollers. Therefore, the toolkit should offer well-documented interfaces to third-party applications and Amarino's architecture should allow seamless extension of built-in events.

Decision 3 – Comprehensive monitoring shall be available (transparent)
Most often, debugging can be frustrating without proper tools. During the workshops people spent most of their time debugging code. System-wide monitoring of actions processed by Amarino, especially since many actions are done in the background, should be visible for users.

Decision 4 – Parallel communication to multiple devices shall be possible (powerful)
Considering Amarino as a toolkit with the purpose of supporting the prototyping process of mobile ubiquitous computing, it should be able to manage communication with more than one microcontroller at a time. This decision leads inevitably to a more complex design but results in a wider range of usage scenarios.

Decision 5 – Bluetooth related code shall be decoupled (portable)
During the prototyping process, Android's underlying Bluetooth stack changed due to a major version update of the Android operating system. This rendered the prototype useless on newer Android devices. The final version of Amarino should separate Bluetooth related code in order to reduce maintenance effort on the off chance of further modifications of the underlying Bluetooth stack. An external Android Bluetooth library should be used to ensure backward compatibility and to outsource necessary code adjustments triggered by potential modifications of Android's Bluetooth implementation. This strategy has yet another positive side effect; it will be less complex and laborious, due to its loose coupling to the Bluetooth layer, to extend the toolkit with additional communication infrastructures (e.g. Wi-Fi).

7.2 Components

The upcoming part of this chapter focuses on Amarino's architecture illustrated in Figure 23. It shows the toolkit distributed among Android and Arduino. The yellow blocks are Amarino components, whereas the orange blocks stand for external applications installed on the phone utilizing Amarino to talk (and listen) to the microcontroller. An external application might be a completely independent third-party application which uses just the Amarino's API to communicate with Arduino. Or it might be an Amarino Plug-in which uses also the API, but on top of that, is smoothly integrated into Amarino's user interface.

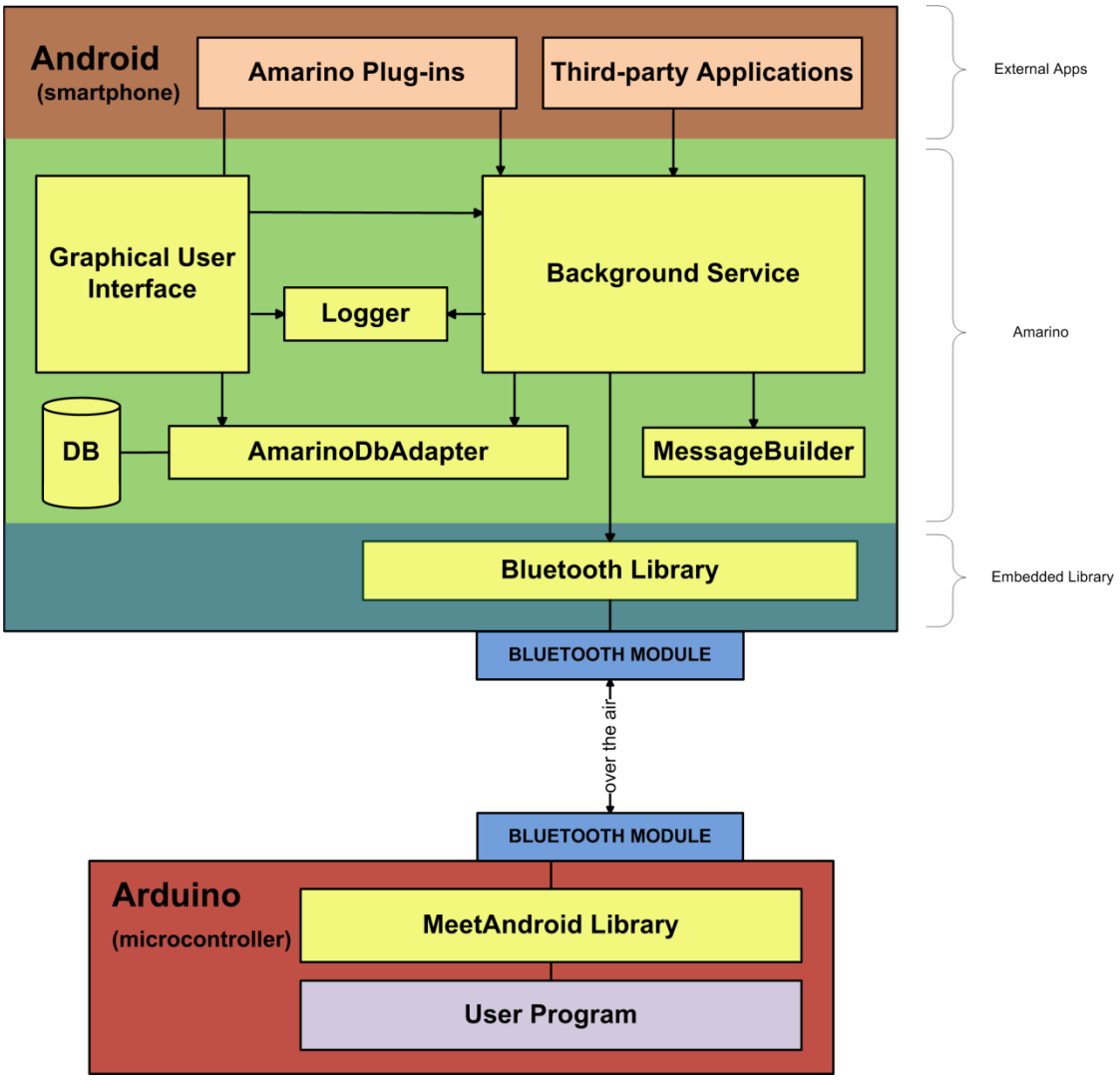


Figure 23: Amarino's architecture

With respect to Decision 1, a *Background Service* component handles all the communication with the microcontroller with the aid of a *MessageBuilder* and a

Bluetooth Library which in turn is designed as a separate entity according to Decision 5. The *Bluetooth Library* implements the hardware abstraction layer and takes care of being backward compatible to Android 1.x devices. Both, the *Background Service* and the *Graphical User Interface (GUI)* components are sending debug information to a synchronized *Logger* component. The *GUI* fetches the entire log when it needs to display it to the user as requested by Decision 3. Finally the persistence layer is realized by the *AmarinoDbAdapter* transparently giving access to the database.

Before we go into details about sole components, a sequence diagram is shown to illustrate the interaction between the components (Figure 24). The sequence starts when a user hits the connect button of a paired, added Bluetooth device on the main screen. This sends an **ACTION_CONNECT** intent, with the device address attached, to the *Background Service*.

If the service does not run yet, it starts and immediately checks, with the aid of regular expressions, if the given address is in correct Bluetooth address format. Next, the service asks the *Bluetooth Library* to initialize the local Bluetooth device. When the Bluetooth device has been initialized the library calls the previously registered ready listener which triggers a connector thread to be started by the *Background Service*. The connector thread tries to open a socket using the *Bluetooth Library*. If the socket could be opened successfully, the *Bluetooth Library* returns a socket directly connected to Arduino's Bluetooth module, otherwise it would raise a connection error and the *GUI* would be informed with an **ACTION_CONNECTION_FAILED** intent. However, since the *Background Service* component now has an open socket to communicate with, it starts a separate connection thread in order to handle communication synchronously. The life cycles of all connection threads (we might have more than one connection thread due to Decision 4, the ability to manage multiple microcontrollers at a time) are maintained by the *Background Service,* which knows the mapping between a connection thread and a microcontroller.

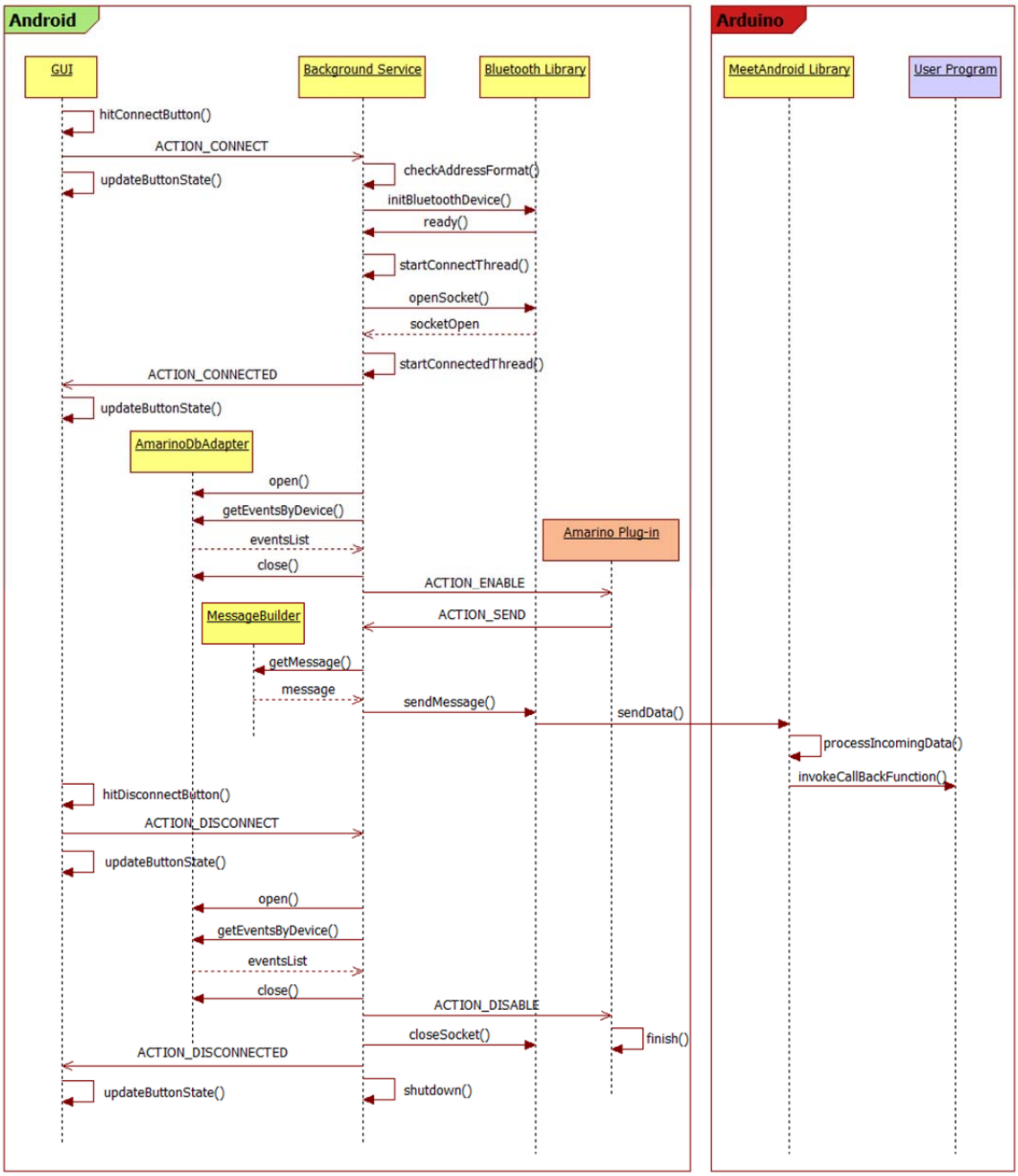


Figure 24: Sequence diagram of connecting and sending data via an Amarino Plug-in

Once the thread is started, the *Background Service* notifies the *GUI* by sending an **ACTION_CONNECTED** intent. Furthermore, in order to determine which events are added to the connected Bluetooth device, the service requests the list of added events from the database. As we know already, each event is implemented as an *Amarino Plug-in*, thus to activate an event, the *Background Service* needs to inform the matching *Amarino Plug-in* that it should begin to send its event data. After an *Amarino Plug-in* received the **ACTION_ENABLE** intent it starts its own

service, hence runs in the background, and sends ACTION_SEND intents whenever it has an event to populate. The *Background Service* receives the ACTION_SEND intents and consults the *MessageBuilder* to convert data attached to the intents to a message with appropriate protocol flags added (details about the protocol are discussed in Section 8.1). Next, it sends the compiled message via the *Bluetooth Library* to Arduino. Noteworthy, the *Bluetooth Library* does not know anything about the underlying protocol or the data type of the message; it only gets a byte array and transmits bytes to Arduino.

On Arduino, the MeetAndroid library analyzes the incoming bytes, cleaning the data from overhead and passes the processed event to the registered call back function within the *User Program*.

As soon as an *Amarino Plug-in* is enabled, it keeps sending events until it receives an explicit ACTION_DISABLE intent from the *Background Service*. This happens as a successive result of a user pressing the disconnect button on the *GUI*. The *GUI* generates an ACTION_DISCONNECT intent and sends it to the *Background Service*. This triggers the service to fetch again the list of events from the *AmarinoDbAdapter* and send ACTION_DISABLE intents to all involved *Amarino Plug-ins*, followed by closing the socket to Arduino. At last the *Background Service* informs the GUI about the new state and shuts down if no more active connections are held.

The following subsections describe each component individually by discussing their responsibilities more detailed and by highlighting some implementation details as well.

7.2.1 Background Service

The core element in Amarino is its *Background Service* component. Its main responsibility is to handle and maintain the communication between microcontrollers while processing commands coming from the GUI, plug-ins or external third-party applications. Furthermore, the *Background Service* accounts responsible for enabling and disabling plug-ins.

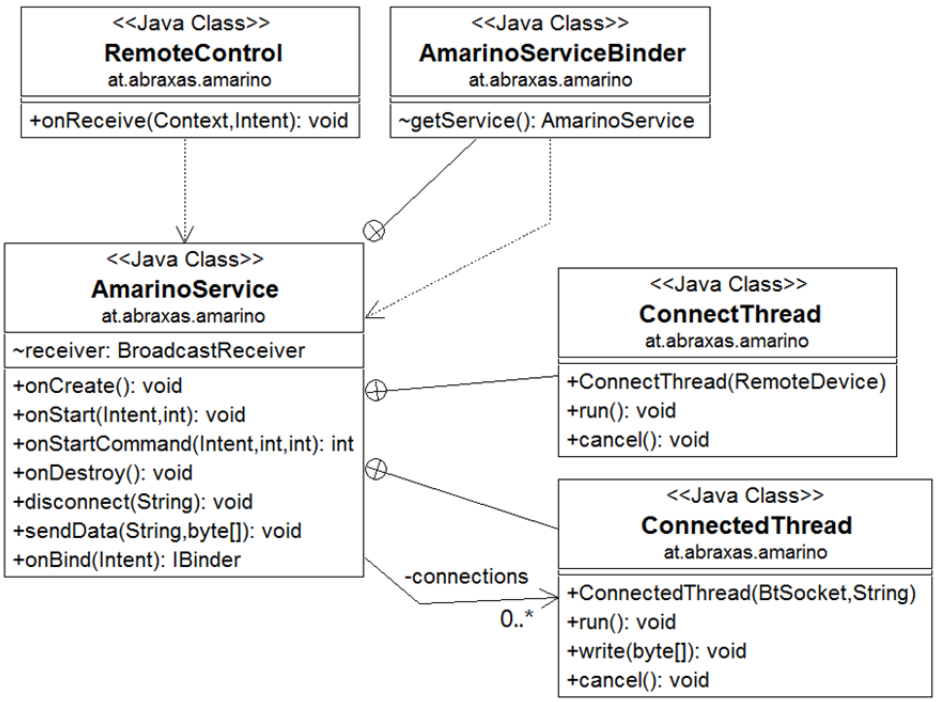


Figure 25: Classes of the Background Service component

Figure 25 displays all classes making up the *Background Service* component. The *RemoteControl* realizes parts of the component interface for external applications. As a *BroadcastReceiver* declared in Amarino's application manifest, the *RemoteControl* enables the *Background Service* to receive command intents such as:

- **ACTION_CONNECT**
- **ACTION_DISCONNECT**
- **ACTION_GET_CONNECTED_DEVICES**

Because of its declaration in the manifest, those intents can be received even when the service is not started. The *RemoteControl* starts the service upon receiving one of the listed intents and passes the intent on to the service. The actual service is implemented in *AmarinoService*, which additionally registers a *BroadcastReceiver*, yet at runtime, to receive **ACTION_SEND** intents. **ACTION_SEND** intents are not registered within *RemoteControl* since **ACTION_SEND** intents are not expected, indeed undesired, to be received without getting an **ACTION_CONNECT** intent beforehand. Therefore, the service does not start when **ACTION_SEND** intents are broadcasted while no active connections are online.

In general, communication with *AmarinoService* is done via intents. However, intents, although rather flexible, lightweight and powerful, are not for free and do have some overhead due to their involvement of the operating system as explained in Section 4.4. Nevertheless, external applications have no choice; they have to send intents to talk to the service. In contrast, components within the same context as the service, just like components of the Amarino application, are able to bind to it in order to receive an instance of the service via an *IBinder* object (*AmarinoServiceBinder*). For the sake of performance, granularity of access and also simply for convenience, the *GUI* binds to the service to receive a persistent connection to it. This allows the *GUI* to call directly methods of *AmarinoService*, hence the GUI has full control over its operation.

Connecting

As mentioned earlier, two types of threads are in place to create and use communication sockets. Upon receiving an ACTION_CONNECT intent or by calling the connect method of *AmarinoService*, a *ConnectThread,* which is an inner class of *AmarinoService*, is instantiated to request a bi-directional communication channel (input and output sockets) between Android and the microcontroller. In case the channel could be opened successfully, it is passed to a *ConnectedThread* responsible for sending and receiving raw data. Moreover, an ACTION_CONNECTED intent is broadcasted to inform interested parties about the established connection.

Sending

ConnectedThreads are put into a hash map of *AmarinoService* where they can be retrieved via the address of the connected device. When data need to be sent, *AmarinoService* gets the proper thread mapped to a device address and uses its *write* method to send data to the Bluetooth device.

Receiving

Each *ConnectedThread* is able to receive data since it is listening constantly to its input socket. Data sent from the microcontroller are processed and packed into an ACTION_RECEIVED intent, then broadcasted to the operating system, so that other applications can catch the intent and extract the data within it using Amarino's API.

Disconnecting

If the disconnect method is called or *AmarinoService* receives an ACTION_DISCONNECT intent, the corresponding *ConnectedThread* is removed from the service's hash map and its *cancel* method is invoked to close the connection and stop the thread. Afterwards, *AmarinoService* broadcasts the ACTION_DISCONNECTED intent.

With respect to a continuous operation of the *Background Service*, *AmarinoService* is set to be a foreground process in Android, raising its priority to hinder the operating system to shut it down due to heavy memory pressure. Lifting a service, which runs in the background, to such a high priority level requires putting a notification icon onto the status bar. Consequently, a user recognizes that Amarino is running in the background, which is a desired functionality for Amarino anyway. *AmarinoService* also uses notifications to inform the user about connection state changes as exemplified in Table 2 (on p. 41).

7.2.2 MessageBuilder

Protocol specifics are decoupled from the *Background Service* and separately processed through the *MessageBuilder* component. For this reason, changes or extensions of the communication protocol will be easier to maintain or even could be replaced without any side effects to other components apart from the MeetAndroid library.

The *MessageBuilder* takes an ACTION_SEND intent, extracts the piggy-backed data, determines the type of the data and, according to the data type, constructs an Amarino conform String message. This String message has a leading event identifier flag and a termination flag, both required by the MeetAndroid library to detect the start and the end of a message while the start flag is also used to decide on the callback function to be invoked.

7.2.3 Bluetooth Library

The Bluetooth Library, written by Stefano Sanna and Emanuele Di Saverio [20], is embedded into Amarino to transparently access the Bluetooth hardware. The

library is backward compatible, hence can be used on Android 1.x and Android 2.x devices, even if there is no official Bluetooth API for Android 1.x.

In order to achieve backward compatibility, the library makes heavy use of the Java Reflection API to access present but hidden Bluetooth functions on Android 1.x implementations. In applying the Factory design pattern, the library returns to a client a proper instance of the local Bluetooth device based on the underlying Android implementation while hiding compatibility issues.

7.2.4 Graphical User Interface

Most aspects of Amarino's GUI are straightforward in terms of Android development. In fact, apart from the *Monitoring* activity each Amarino activity extends the *ListActivity* class of the Android SDK. *ListActiviy* is a special activity with the purpose of displaying a list of items. Normally, a *ListActivity* requires a data source which it can bind to in order to retrieve list items. Luckily, the Android SDK provides already various standard adapter classes encapsulating data sources, such as database cursors or arrays, where *ListActivity* can get its items from. By definition, an adapter compiles and returns *View* objects of data items it represents. However, instead of using one of the provided Android SDK adapters, Amarino usually extends *BaseAdapter* to feed the *ListActivity* objects with *View* objects. By extending the *BaseAdapter* class, Amarino has full control over *View* generation even during runtime. Therefore, Amarino is able to deliver more customized and complex *Views* to its *ListActivity* classes as the default adapters of the Android SDK would be able to.

One GUI aspect which should be highlighted is the implementation of the event management user interface, since it is capable of displaying real-time data coming from plug-ins. Section 6.2.2 already discussed the Event Manager from a user's point of view. Now we take a look at its implementation structure as outlined in Figure 26.

Once the *EventListActivity* is started it fetches *View* items from the *EventListAdapter* to display events added to a Bluetooth device. In addition, it registers a *BroadcastReceiver* to receive **ACTION_SEND** intents, necessary to update the visualizers accordingly.

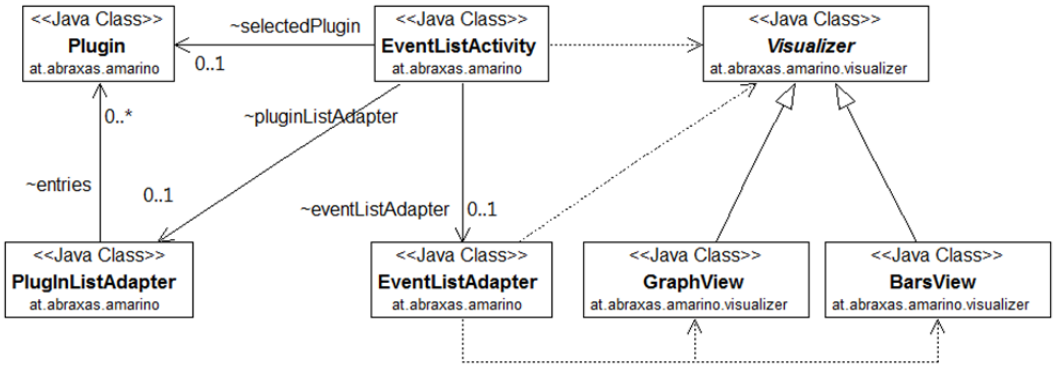


Figure 26: Class diagram of the Event Manager

Show list of installed events

Before a user hits "Add Event" to bring up the list of available events (Figure 16, on p. 43), *EventListActivity* calls *buildPluginList*, shown in detail in Code Snippet 8, to retrieve a list of installed Amarino Plug-ins from the operating system by querying the *PacketManager* about activities registered for the **ACTION_EDIT_PLUGIN** (line 3). I explain later in Section 8.2 that it is essential for an Amarino Plug-in to have an activity registered for the **ACTION_EDIT_PLUGIN** intent. This strategy enables *EventListActivity* to determine which plug-ins are installed and to call its configuration activity when needed. Lines 8 to 15 of Code Snippet 8 show the meta-information about an activity extracted from an activity's *ResolveInfo* object. The meta-data is put into the plug-in object *p* and added to the collection of plug-in objects (line 14). The *PlugInListAdapter,* which is responsible for generating *View* objects to be displayed within the dialog of available events, is instantiated by passing the *plugins* collection as its data source into it.

```
1  private void buildPluginList() {
2    PackageManager pm = this.getPackageManager();
3    List<ResolveInfo> editActivites = pm.queryIntentActivities(
4                                      new Intent(AmarinoIntent.ACTION_EDIT_PLUGIN), 0);
5
6    ArrayList<Plugin> plugins = new ArrayList<Plugin>(editActivites.size());
7
8    for (ResolveInfo ri : editActivites){
9      Plugin p = new Plugin();
10     p.label = ri.loadLabel(pm).toString();
11     p.icon = ri.loadIcon(pm);
12     p.packageName = ri.activityInfo.packageName;
13     p.editClassName = ri.activityInfo.name;
14     plugins.add(p);
15   }
16   pluginListAdapter = new PlugInListAdapter(this, plugins);
17 }
```

Code Snippet 8: Query the list of installed Amarino Plug-ins

In Code Snippet 9 line 5 we see that the instance of *PluginListAdapter* is set as the data adapter for an *AlertDialog*. The *showPlugins* method in Code Snippet 9 is called when the "Add Event" button was hit. It creates a new dialog showing all available events, respectively plug-ins. When a user selects an event, its configuration screen will be shown to the user.

The *PluginListAdapter* provides a list of *View* objects to the dialog which shows a label and an icon. In addition, it stores information about how to start the configuration activity. The resulting dialog is shown at the center screenshot of Figure 16 (on p. 43).

The click listener of a *PluginListAdapter* item builds the intent necessary to start the configuration activity of a plug-in. Besides the **ACTION_EDIT_PLUGIN** action, the exact class name and its package name is set to ensure a unique hit when the operating system delivers the intent. Otherwise the system would not know which of the many **ACTION_EDIT_PLUGIN** activities to start. The attached extras are required for the configuration activity to function properly according to the plug-in specification discussed in Section 8.2.

```
1  private void showPlugins(){
2    new AlertDialog.Builder(EventListActivity.this)
3      .setTitle("Add Event")
4      .setIcon(R.drawable.icon_very_small)
5      .setAdapter(pluginListAdapter, new DialogInterface.OnClickListener()  {
6
7         @Override
8         public void onClick(DialogInterface dialog, int which) {
9             // start EditActivity of selected plugin
10            selectedPlugin = pluginListAdapter.entries.get(which);
11
12            Intent intent = new Intent(AmarinoIntent.ACTION_EDIT_PLUGIN);
13            intent.setClassName(selectedPlugin.packageName,
14                                selectedPlugin.editClassName);
15
16            intent.putExtra(AmarinoIntent.EXTRA_FLAG, getNextFlag());
17            intent.putExtra(AmarinoIntent.EXTRA_DEVICE_ADDRESS, device.address);
18            intent.putExtra(AmarinoIntent.EXTRA_PLUGIN_ID,
19                            selectedPlugin.label.hashCode());
20
21            startActivityForResult(intent, REQUEST_CREATE_EVENT);
22        }
23     })
24     .create()
25     .show();
26 }
```

Code Snippet 9: Create and show the dialog of available events

Once the user has finished the configuration of the event, the configuration screen returns to *EventListActivity*, due to calling the activity using

startActivityForResult (line 21). The configuration activity returns an intent with useful information about the event to the *EventListActivity*. The returned intent specifies min and max values for the visualizer, the class which implements the background service to generate event messages and the name and description of the event. *EventListActivity* uses this data to create an event object and to store it into the database. After *EventListActivity* updates its data adapter the newly added event appears on the screen.

Since *EventListActivity* registered a *BroadcastReceiver* for **ACTION_SEND** intents, it will receive each message which is about to be delivered to Arduino. Catching the send intent allows *EventListActivity* to extract its data and its originator while updating the corresponding visualizer in real-time. *EventListActivity* is only interested in messages sent by plug-ins, hence **ACTION_SEND** intents not coming from plug-ins are ignored.

7.2.5 AmarinoDbAdapter

Some information in Amarino needs to be stored persistently: the list of added devices and their associated events. Amarino implements a SQLite database for its persistent data. Figure 27 illustrates the database schema of Amarino's database.

Table Device

A dataset in table *Device* represents one added Bluetooth device with its friendly *name* and a Bluetooth *address* as well as a primary key called *_id*.

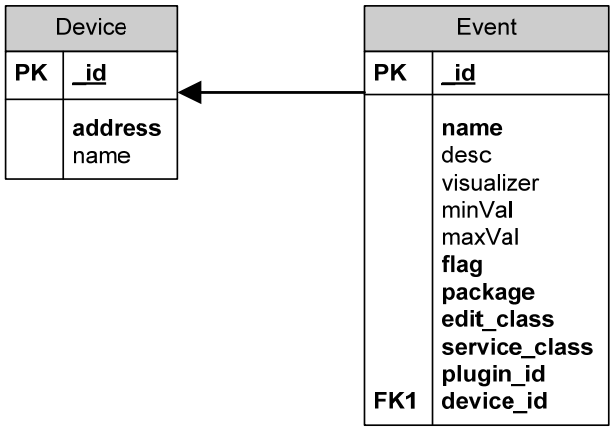


Figure 27: Amarino's database structure

This table is consulted when the *GUI* needs to know which devices to show in the device list.

Table Event

Each entity in table *Event* belongs to exactly one device; hence the table specifies a foreign key called *device_id*. Beside the primary key (*_id*), an event has a *name*, a *description*, a *visualizer* type, attributes to define the range for the visualizer (*minVal*, *maxVal*), and a *flag* which is used as an external event identifier. The other attributes are necessary to communicate with the *Amarino Plug-in* realizing the event. The *package* name of the plug-in to send broadcasts more accurately, the name of its *edit* and *service class* as well as a *plugin_id* used to map messages coming from a plug-in to a certain event.

The *AmarinoDbAdapter* creates the database upon installing Amarino and offers CRUD[18] operations for devices and events stored in Amarino's database. Supposing that the database needs to be upgraded in later versions of Amarino, the *AmarinoDbAdapter* component has been designed to incorporate upgrade scripts straightforwardly. It uses a *DatabaseHelper* to figure out if the database version has changed and applies the upgrade script if mandatory.

7.2.6 Logger

The *Logger* component is used by other components to store debugging messages during Amarino's operation; however the logger does not store the messages persistently.

One important aspect of a logging component is to maintain the order of incoming messages. This requirement has been realized by using a class instance (static) list of strings to store the log messages, while adding new messages to the list solely is done using static synchronized methods. The log behaves like a ring buffer; the oldest message is removed when the buffer maximum is reached and a new message is about to be added. The implementation of the *Logger* component as shown in Figure 28 is based on the Observer pattern introduced by Gamma et al.[42].

[18] CRUD stands for create, read, update and delete

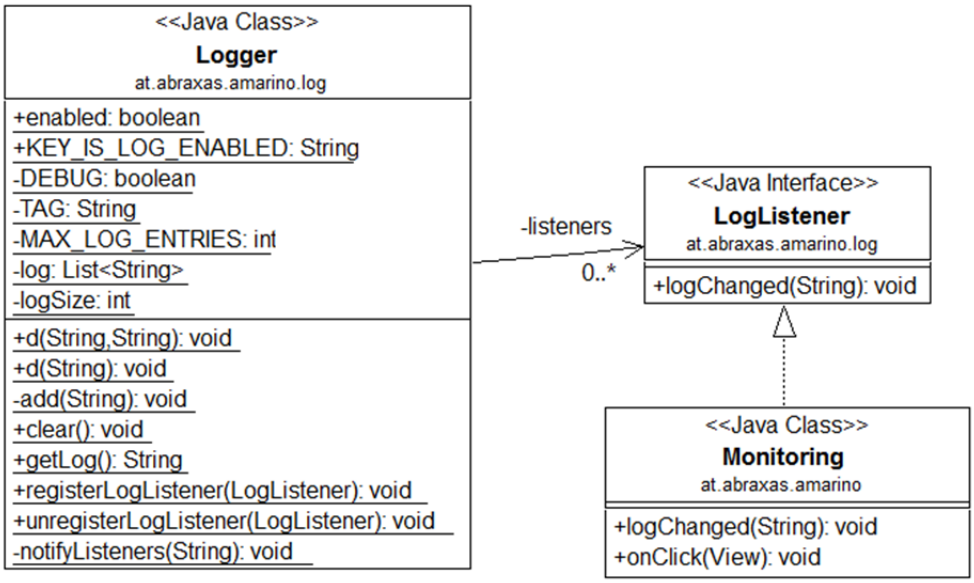


Figure 28: Logger implementation

Interested parties (observers) have to implement the *LogListener* interface and must register using *registerLogListener(LogListener)* to get notification about changes within the *Logger* (model).

Addressing performance, the latest message is passed into the *logChanged(String)* method as a parameter. This way the listeners do not have to fetch the entire log upon each update, and communication between the model and its observers is reduced. In Amarino, the only observer is the *Monitoring* class, the activity implementing the monitoring screen.

7.2.7 MeetAndroid Library

In Section 6.3 it was described how to use the MeetAndroid library as a developer and Figure 20 showed the interface of the library. This subsection explains the processing the library does and discusses its role in Amarino.

The MeetAndroid library, responsible for managing incoming and outgoing messages on the microcontroller side, is implemented as a C++ class called *MeetAndroid*. The library accesses the serial port of an Arduino to send and receive data. The attached Bluetooth module communicates with the serial port of an Arduino transmitting the byte stream over the air directly to a connected Bluetooth device, in our case the Android phone.

Incoming messages from the phone will have a start (event id) and an end (ACK) flag to enable the library determining the start and the end of a message. This requires the library to buffer incoming bytes until it receives the ACK flag, thus *MeetAndroid* implements a byte buffer, 64 bytes in size. The length of accepted incoming messages is therefore limited to 62 bytes (64 bytes - 2 byte flags).

Additionally, *MeetAndroid* implements a function pointer map (ID → function pointer) to store function pointers used to invoke registered callback functions. A function pointer is determined by the ID, an ASCII character, for which the callback function was registered for. Up to 75 function pointers can be held, covering ASCII characters ranging from '0' to 'z', which are used in Amarino to represent different event identifiers. As a result, the MeetAndroid library is able to distinguish between 75 different event types. The *registerFunction* of the library simply puts the provided function pointer into the map and removes it when *unregisterFunction* is called.

Receive message

The process of receiving incoming bytes is illustrated by the activity diagram in Figure 29. The *receive* method of the MeetAndroid library implements the demonstrated process.

First, the library asks the serial port if there are any data in the serial buffer waiting to be read. If so, the oldest byte of the serial buffer is consumed and analyzed. An abort byte would immediately clear the buffer, while an acknowledge byte determines the end of a message and initiates the message to be processed (Figure 30). In case the buffer is not full yet, the consumed byte is appended to the *MeetAndroid* byte buffer.

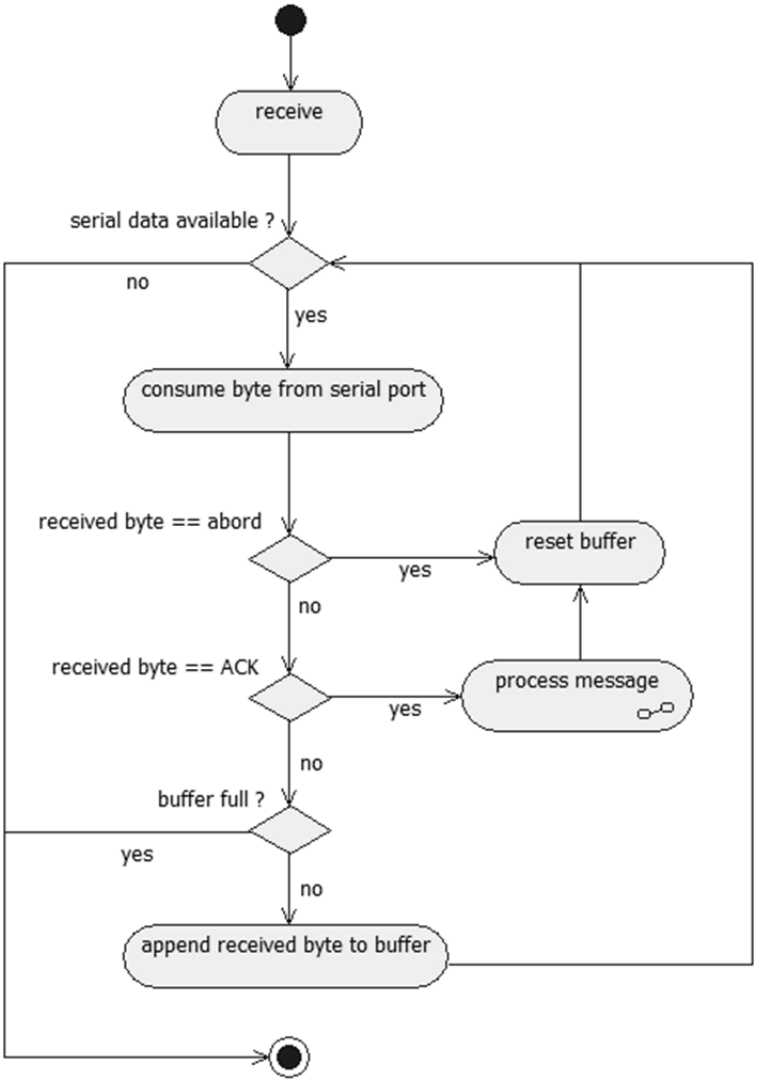


Figure 29: Activity diagram illustrating the extraction of a message from incoming bytes

Process message

When a message has been entirely received, the library tries to figure out which callback function has to be called for the received event. Remember, a user registers a callback function for each event identifier in his or her Arduino program (explained in Section 6.3). The actual message processing is shown in Figure 30.

At the beginning the process message function checks if the first byte of the message is a valid event identifier; characters between '0' and 'z' are valid IDs. The users-defined error function will be called if the event ID is not in the expected range. Did the user not define an error function in the Arduino program the library sends a default error message over the Bluetooth connection.

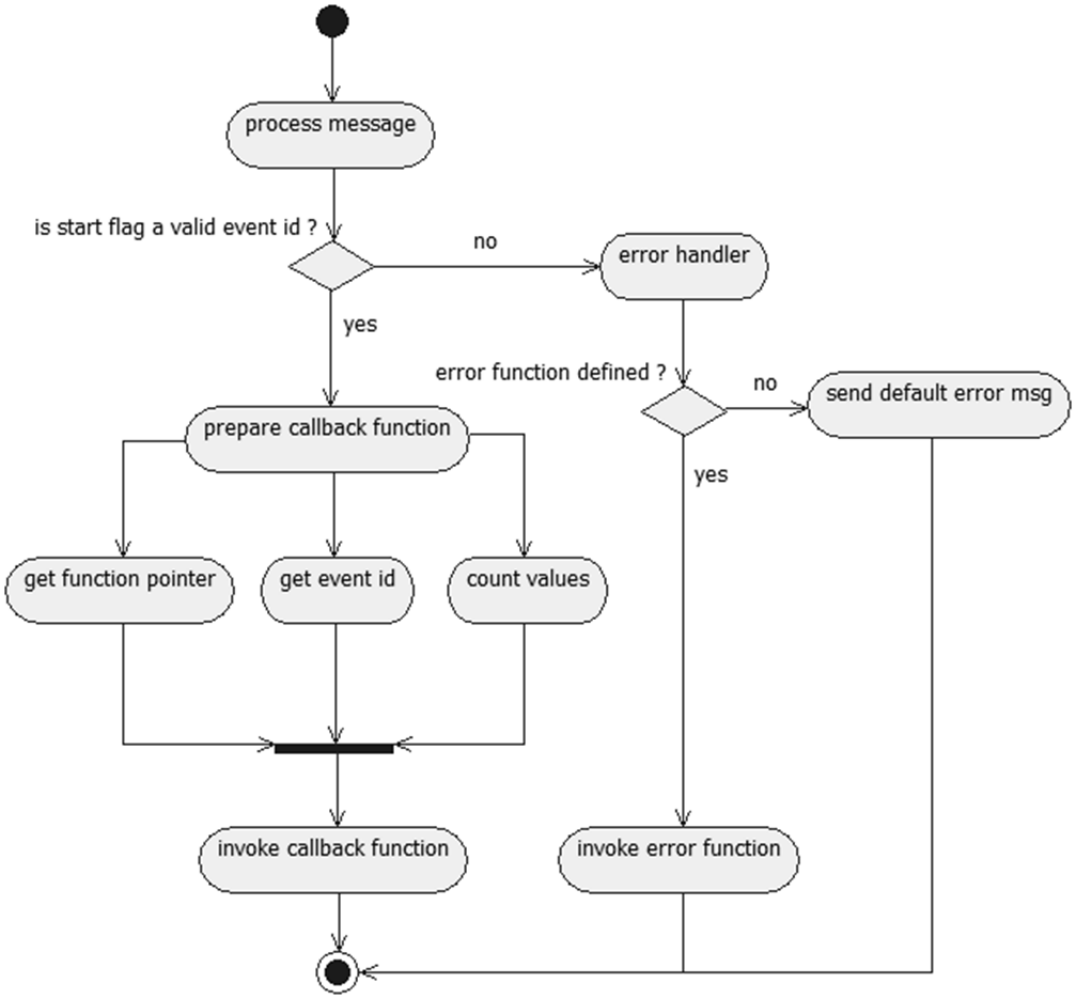


Figure 30: Activity diagram showing message processing in the MeetAndroid library

In case the event identifier is valid, *MeetAndroid* gets the function pointer registered for this event ID, extracts the event ID from the buffer and counts the values contained in the message. Normally there is zero (event is a signal only) or one value (Integer, Boolean, etc.) attached to an event, but for arrays there might be more than one. The number of values is needed by a user to iterate over a received array. Last, the process method invokes the registered callback function matching the event ID, while the event ID and the number of values are passed into it for convenience.

Chapter 8

8 Implementation Details

The foregoing two chapters gave a general overview of Amarino; one concentrated on the user's point of view and the other one on architectural design. The following implementation details provide insights into technical details of the communication protocol and the plug-in technology deployed by Amarino. At the end, a short manual about how to design a new Amarino Plug-in is given.

8.1 Communication Protocol

The main purpose of Amarino is to bridge the gap between a smartphone and a microcontroller in terms of interoperability. As a result, Amarino aims to provide a transparent communication channel for toolkit users. Consequently, the underlying communication protocol, conceptually shown in Figure 31, is an essential aspect of the toolkit worth to be discussed in more detail.

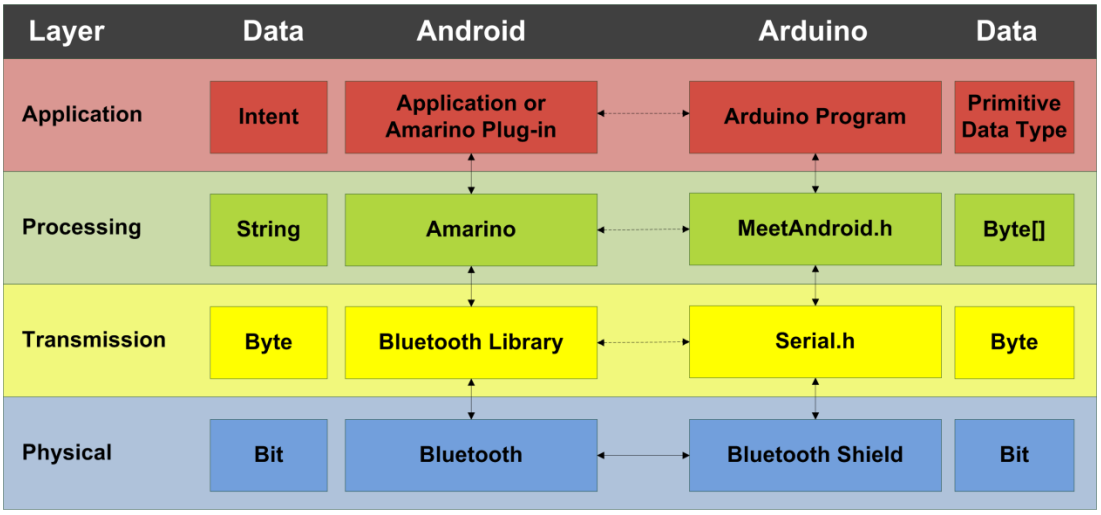


Figure 31: Communication model

8.1.1 Application Layer

At the top layer, the application layer, both realms' communication, Android's and Arduino's, is based on high-level data structures and strategies. The phone's application layer uses intents to send and receive messages, whereas the Arduino program sends and receives primitive data types. In order to send data from an Android application to an Arduino, an **ACTION_SEND** intent has to be created with the following extras attached to it:

- **EXTRA_DEVICE_ADDRESS** – the address of the recipient (String)
- **EXTRA_FLAG** – the event identifier (char)
- **EXTRA_DATA_TYPE** – the type of **EXTRA_DATA** (int)
- **EXTRA_DATA** – the data to send (depends on **EXTRA_DATA_TYPE**)

The intent will cause the registered callback function for the given event identifier in **EXTRA_FLAG** to be invoked at the Arduino which matches the address set in **EXTRA_DEVICE_ADDRESS**. The Arduino program extracts the actual data by calling the *MeetAndroid getXXX* methods according to the data type set in **EXTRA_DATA_TYPE**. Slightly different is the intent composition when a plug-in is sending data, because plug-ins normally do not know the device address or the event identifier when sending data. Instead of an **EXTRA_DEVICE_ADDRESS** the plug-in would put its plug-in identifier as **EXTRA_PLUGIN_ID** into the intent and omit the **EXTRA_FLAG**, since Amarino can determine the flag and the device address based on the provided plug-in ID. Generally, a developer, both application and plug-in developer, might use the functions provided in the Amarino API to transmit messages. The Amarino API hides the described composition of the intent and sends the data via Amarino's background process to Arduino.

Sending data from Arduino requires calling one of *MeetAndroid*'s *send(…)* methods. It allows the Android program to send primitive data types. The Android application has to register a *BroadcastReceiver* for **ACTION_RECEIVED** intents to receive the data from Arduino. An **ACTION_RECEIVED** intent has the following extras attached which can be accessed via the *getXXXExtra* methods of the intent:

- **EXTRA_DEVICE_ADDRESS** – the address of the sender (String)

- **EXTRA_DATA_TYPE** – the type of **EXTRA_DATA** (int)
- **EXTRA_DATA** – the received data (depends on **EXTRA_DATA_TYPE**)

An example of how to receive data from Arduino has already been shown in Code Snippet 7 (on p. 52).

8.1.2 Processing Layer

When the application layer, an application or a plug-in, passes the send intent to the processing layer, Amarino extracts the data added to the intent and converts it into a String message which will be sent to Arduino. The resulting String will be led by the event identifier and terminated by an acknowledge flag (ACK). A complete specification in BNF[19] notation of the composed String (<string>) is shown in Figure 32.

```
1   <string>     ::= <event_id> <data> <ack>
2   <event_id>   ::= <digit> | <letter>
3   <digit>      ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
4   <letter>     ::= "a"-"z" | "A"-"Z" (all characters between a and z or A and Z)
5   <data>       ::= <chars> | <array>
6   <chars>      ::= <character> | <character> <chars>
7   <character>  ::= all ASCII characters 0-255 without <ack>
8   <array>      ::= <chars> | <chars> <delimeter> <array>
9   <delimeter>  ::= ";"
10  <ack>        ::= ASCII char no. 17
```

Figure 32: BNF of a String message composed by Amarino in the processing layer

The MeetAndroid library receives the String as a byte array and stores it into its byte buffer, since the data type String is not natively available in Arduino. Data sent from Arduino are terminated with ASCII character number 12. When Amarino receives the data, it cuts off the termination character and puts the remaining data into a String object which is attached to an **ACTION_RECEIVED** intent in order to be forwarded to third-party applications.

8.1.3 Transmission Layer

The transmission layer is responsible for direct communication with the Bluetooth hardware. In Android the transmission layer is realized by an external Bluetooth library [20]. The library uses the Android SDK to access Bluetooth features of Android 2.x devices, and Java's Reflection API to call hidden Bluetooth system functions on Android 1.x devices, since the Android 1.x SDK does not support

[19] Backus-Naur Form

Bluetooth. Thanks to the Bluetooth library, Amarino is compatible to almost all available Android devices. It should be said, that there are more layers in between the Bluetooth library and the Bluetooth hardware of a phone, however those layers are part of the Android operating system and not relevant to understand Amarino's implementation. From Amarino's point of view, the Bluetooth library offers a serial communication socket to another Bluetooth device. In order to send data over an established RFCOMM[20] channel, the String message coming from the processing layer has to be converted to a byte array.

The Serial library, integrated and thus available on all Arduino boards, implements the serial port hardware abstraction layer for the microcontroller. It accesses the digital RX and TX pins of an Arduino to receive and send data. Received bytes are put into a byte ring buffer. In order to get the received data, the Serial library offers a *read* method which returns and consumes the oldest byte present in the ring buffer and an *available* method returning the number of available bytes in the ring buffer. For sending data the library provides various *print* methods.

8.1.4 Physical Layer

The bottom layer is implemented in hardware. The Serial Port Profile (SPP) as specified in [43] forms the ground base for RS-232 serial cable emulated data transmission between two Bluetooth enabled devices. Therefore, either Bluetooth device, Android and Arduino, needs to implement SPP. But before a Bluetooth device can be used to transmit data, it has to be enabled and initialized. The transmission layer provides functions to turn on and initialize the Bluetooth hardware. In Arduino it is important to initialize the Bluetooth shield with the correct baud rate in order to synchronize the communication between the microcontroller and the attached Bluetooth shield. A baud rate of 57,600 could be proved to work best for external Bluetooth shields, while 115,200 baud is optimal for the Arduino BT board, which has an embedded Bluetooth chip.

[20] Radio Frequency Communication – Bluetooth transport protocol to emulate up to sixty RS-232 serial ports

8.2 Amarino Plug-in Technology

Amarino supports extensions through its plug-in technology. Developers are motivated to extend the list of available events shown in Table 3 (on p. 44) by building plug-ins. Each plug-in normally represents and implements exactly one event. An exception is the *Amarino Plug-in Bundle*. The plug-in bundle is shipped with Amarino and once installed gives the user access to all events listed in Table 3 (on p. 44).

Principally, a plug-in is a separate Android application, without a main activity though, hence is not listed in Android's launcher application and therefore cannot be started manually. In fact, Amarino controls the life-cycle of a plug-in. When a user selects an event and adds it to a device, Amarino takes care of starting the plug-in upon connection to the device and disabling it just before disconnecting from the device. Amarino requires a plug-in developer to follow some conventions in order to be able to control the life-cycle of the plug-in and to integrate seamlessly into Amarino's Android application. The Android API provides a framework for plug-in development and the Amarino website offers a how-to of plug-in development.

In Figure 33, the class diagram of the Amarino Plug-in Time example, available for download on Amarino's website, is presented. It demonstrates the use of framework classes and indicates the structure of a simple plug-in. A typical plug-in consists of four essential components:

- PluginReceiver
- BackgroundService
- EditActivity
- AndroidManifest

In short, the *PluginReceiver* enables and disables the *BackgroundService*. The service's role is to produce events. The *EditActivity* implements the plug-in configuration screen and the *AndroidManifest* file specifies meta-information about the plug-in.

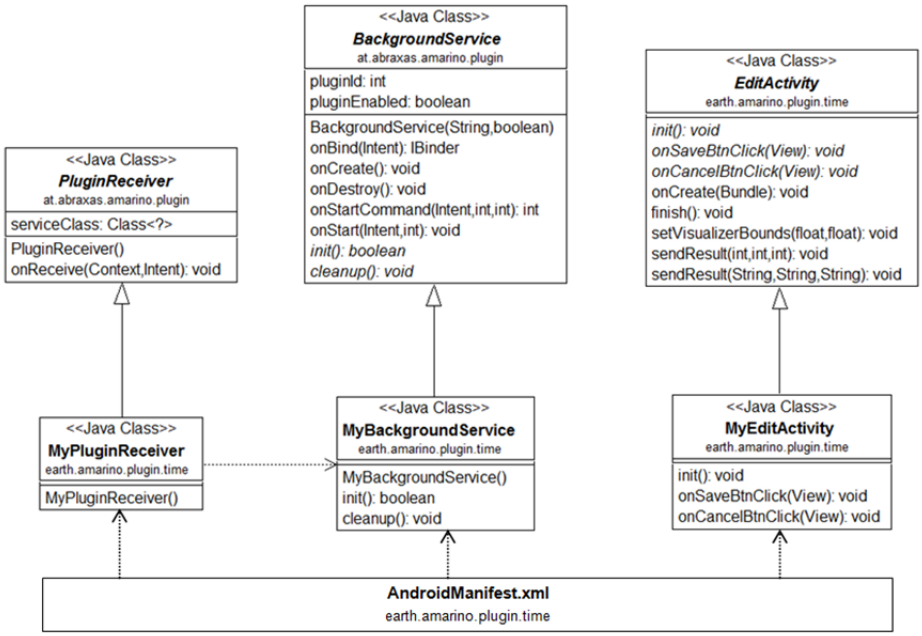


Figure 33: Class diagram of a plug-in example implementation

Before each single component is discussed in detail, the interaction between the components is exemplified with the aid of the sequence diagram in Figure 34, which illustrates the communication between Amarino and a plug-in.

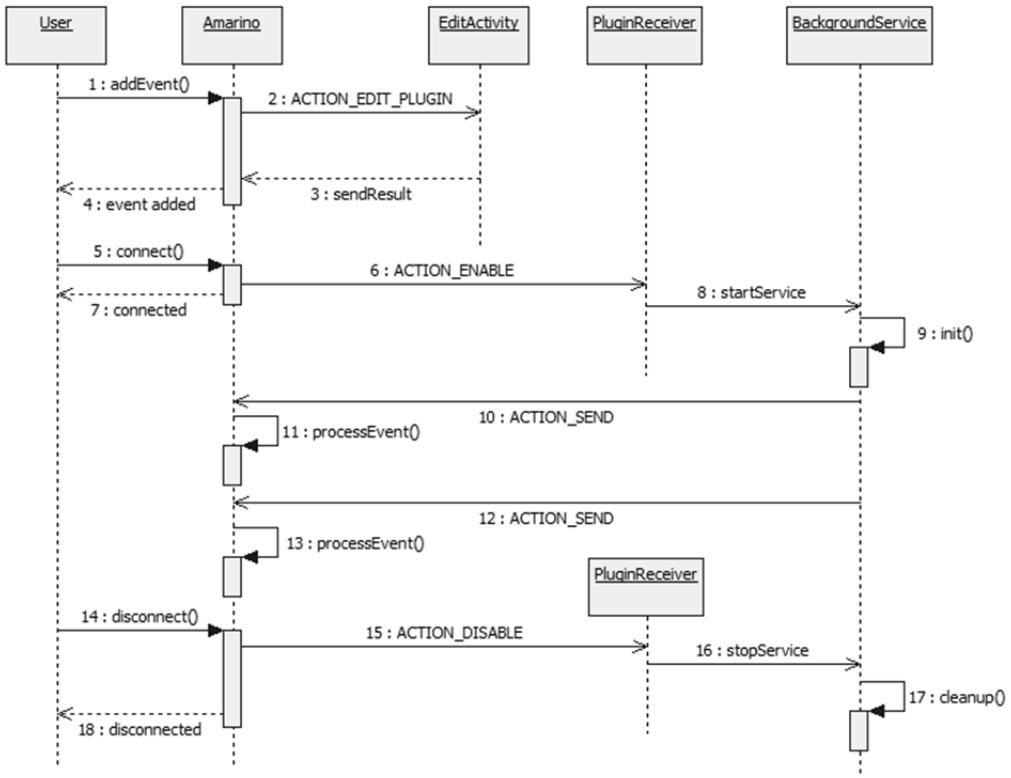


Figure 34: Sequence diagram demonstrating the role of a plug-in

The plug-in configuration screen is started when a user selects a certain event. Amarino consults the operating systems package manager as shown in Code Snippet 9 (on p. 67) and starts the *EditActivity*. After the user has finished the configuration, *EditActivity* returns to Amarino. The result is that the event has been added to the device. Next, Amarino gets a connect request, which will start the connection sequence already discussed in Section 7.2. Upon successful connection, Amarino sends an **ACTION_ENABLE** intent to all plug-ins, representing events added to the connected device. The operating system knows, according to the information in the *AndroidManifest* file of the plug-in, that the *PluginReceiver* is interested in **ACTION_ENABLE** intents and thus instantiates the *PluginReceiver* while passing the intent into its *onReceive(Intent intent)* method. The *PluginReceiver* then starts the background service. Afterwards the *BackgroundService* starts and calls the *init* method to allow implementation specific code to be initialized. Since the *BackgroundService* has been started, it is ready to send events or receive messages from Arduino (receiving data is not covered by this diagram). How often the *BackgroundService* sends messages, depends on its actual implementation and purpose, however as soon as the user requests disconnect, Amarino disables the plug-in by sending the **ACTION_DISABLE** intent.

8.2.1 PluginReceiver

The *PluginReceiver* is a *BroadcastReceiver* registered in *AndroidManifest* to receive **ACTION_ENABLE** and **ACTION_DISABLE** intents. All a developer has to do, is to extend the abstract *PluginReceiver* offered by Amarino's framework and set the implemented background service class within its constructor.

```
1  public class MyPluginReceiver extends PluginReceiver {
2
3    public MyPluginReceiver() {
4      // TODO change name of your background service class if necessary
5      serviceClass = MyBackgroundService.class;
6    }
7  }
```

Code Snippet 10: Customized PluginReceiver

Setting the name of the implemented background service class is required by the super class *PluginReceiver* to gain control over the component that starts and stops the service of the plug-in. Basically, the *PluginReceiver* component can be seen as the interface of a plug-in.

8.2.2 BackgroundService

A plug-in is designed to run as a background service on the phone. Occasionally it sends events to Amarino to be forwarded to Arduino. It should be said that a plug-in might also receive data from Arduino by implementing a *BroadcastReceiver* for **ACTION_RECEIVED** intents. A default implementation of a background service is provided by Amarino. The developer has to extend the abstract *BackgroundService* class and overwrite two abstract methods, *init* and *cleanup*. The function of the service is entirely up to the developer; the service might call web services, listen to sensors or even do much more complex tasks such as arithmetic calculations, navigation, starting several threads or communicating with other applications. At the end, the result should be an event that is sent via Amarino to the microcontroller.

8.2.3 EditActivity

Since plug-ins can be arbitrary complex, it might be desired to customize a plug-in before it does its job. In fact Amarino requires a plug-in to have a configuration screen, the *EditActivity*. When a user adds an event to a device, Amarino starts the *EditActivity* of the selected plug-in sending the assigned event ID (flag), the device address and a generated unique plug-in ID to the edit activity. A developer is relatively free regarding the implementation of the *EditActivity*, however it is essential to return a result intent to Amarino when the activity is finished. The result intent has to have mandatory extras attached to it as shown in the code example below.

```
1   final Intent resultIntent = new Intent();
2
3   resultIntent.putExtra(AmarinoIntent.EXTRA_PLUGIN_NAME, pluginName);
4   resultIntent.putExtra(AmarinoIntent.EXTRA_PLUGIN_DESC, description);
5   resultIntent.putExtra(AmarinoIntent.EXTRA_PLUGIN_SERVICE_CLASS_NAME, serviceClassName);
6   resultIntent.putExtra(AmarinoIntent.EXTRA_PLUGIN_ID, pluginId);
7   resultIntent.putExtra(AmarinoIntent.EXTRA_VISUALIZER_MIN_VALUE, visualizerMin);
8   resultIntent.putExtra(AmarinoIntent.EXTRA_VISUALIZER_MAX_VALUE, visualizerMax);
9   resultIntent.putExtra(AmarinoIntent.EXTRA_PLUGIN_VISUALIZER,
10                        AmarinoIntent.VISUALIZER_GRAPH);
11
12  setResult(RESULT_OK, resultIntent);
```

Code Snippet 11: Example of a result intent set in EditActivity

Developers are free to use the Amarino Time plug-in from the website as a starting example and to reuse its *EditActivity*, which offers a nice configuration screen

similar, in terms of function and look-and-feel, to those found in the *Amarino Plug-in Bundle*.

8.2.4 AndroidManifest

The *AndroidManifest.xml* file of a plug-in defines the mapping between intents and components as well as specifies other important meta-data, like the icon and the name of the application, needed by the operating system to install the application and to access it.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3        android:versionCode="1"
4        android:versionName="1.0"
5        package="earth.amarino.plugin.time">
6
7    <application android:icon="@drawable/icon" android:label="@string/app_name">
8
9         <!-- ###   EditActivity   ### -->
10        <activity android:name=".MyEditActivity"
11                  android:label="@string/plugin_name"
12                  android:icon="@drawable/icon"
13                  android:exported="true">
14            <intent-filter>
15                <action android:name="amarino.intent.action.EDIT_PLUGIN" />
16                <category android:name="android.intent.category.DEFAULT" />
17            </intent-filter>
18        </activity>
19
20
21        <!-- ###   BackgroundService   ### -->
22        <service android:name=".MyBackgroundService" />
23
24
25        <!-- ###   PluginReceiver   ### -->
26        <receiver android:name=".MyPluginReceiver">
27             <intent-filter>
28                 <action android:name="amarino.intent.action.ENABLE" />
29                 <action android:name="amarino.intent.action.DISABLE" />
30             </intent-filter>
31        </receiver>
32
33     </application>
34     <uses-sdk android:minSdkVersion="3" android:targetSdkVersion="8"/>
35
36 </manifest>
```

Code Snippet 12: AndroidManifest.xml file of the Android Time plug-in example

Important for Amarino to be able to start the configuration screen, the manifest file has to set the intent filter (Code Snippet 12, line 14 to 17) for the *EditActivity* class to match Amarino's **ACTION_EDIT_PLUGIN** (15) and the **DEFAULT** category (16). The *BackgroundService* simply needs to be defined (22), while the *PluginReceiver* must specify an intent filter (line 27 to 30) for **ACTION_ENABLE** (28) and **ACTION_DISABLE** (29).

One question remains: how does Amarino know which plug-ins are installed on the phone to display them in the list of available events? Well, Amarino queries Android's package manager to return all activities which have registered for the intent **ACTION_EDIT_PLUGIN**. Code Snippet 8 on page 66 demonstrates the implementation for this query.

Chapter **9**

9 Showcases

Amarino has already been used in a couple of projects outside the scope of this thesis. However, I present three proof of concept examples I created to illustrate the practical use of Amarino: an ambient light which can be controlled by a smartphone, a shirt continuously displaying its wearer's phone state and an armband counting push-ups displaying the counted number on the phone screen. These examples demonstrate different aspects of Amarino. In AmbientLight the Amarino API has been utilized to develop an Android application which controls a multi color lamp. CallMyShirt just uses built-in events shipped with Amarino and Workout receives sensor readings from a wearable and displays the data on the smartphone screen.

9.1 AmbientLight

For the AmbientLight example an Android application has been developed to control an Android driven multicolor lamp. The Android application offers a user interface to change the color of the lamp by moving red, green and blue sliders, which enables a user to produce the entire RGB color model with the lamp. The schematic in Figure 35 reveals that three LEDs (red, green and blue) are connected to pin 9, 10 and 11 of the Arduino, while each LED is secured with a resistor. By using pulse-wide modulated pins, a resolution of 256 steps is available to control the brightness of a LED. Therefore, theoretically more than 16.7 million colors can be produced.

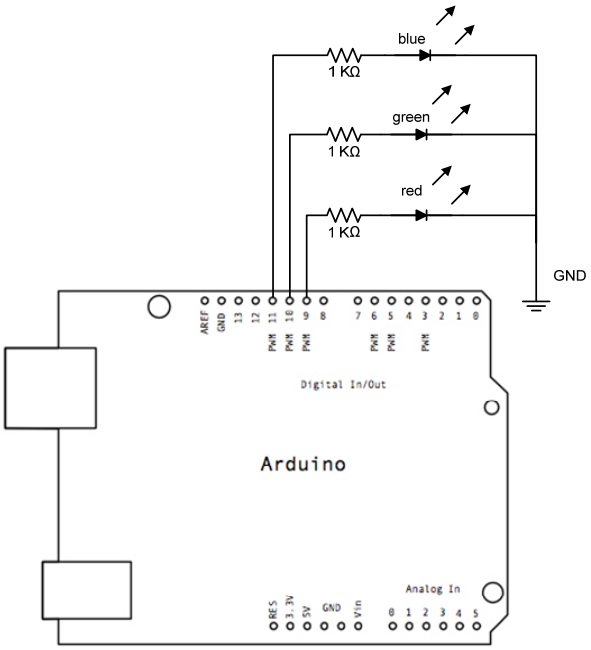


Figure 35: Arduino schematic for the AmbientLight example

The Arduino program shown in Code Snippet 13 is straightforward. It registers three callback functions, one for each color, changing the output voltage of the corresponding pin (9=red, 10=green, 11=blue) based on the integer value received from the phone.

```
#include <MeetAndroid.h>

MeetAndroid meetAndroid;

// we need 3 PWM pins to control the leds
int redLed = 9;
int greenLed = 10;
int blueLed = 11;

void setup() {
  Serial.begin(57600);

  // register callback functions, which will be called when an associated event occurs
  meetAndroid.registerFunction(red, 'o');
  meetAndroid.registerFunction(green, 'p');
  meetAndroid.registerFunction(blue, 'q');

  pinMode(redLed, OUTPUT);
  pinMode(greenLed, OUTPUT);
  pinMode(blueLed, OUTPUT);

  // just set all leds to high so that we see they are working well
  digitalWrite(redLed, HIGH);
  digitalWrite(greenLed, HIGH);
  digitalWrite(blueLed, HIGH);
}

void loop() {
  meetAndroid.receive();
}

void red(byte flag, byte numOfValues) {
  analogWrite(redLed, meetAndroid.getInt());
```

```
34  }
35
36  void green(byte flag, byte numOfValues) {
37    analogWrite(greenLed, meetAndroid.getInt());
38  }
39
40  void blue(byte flag, byte numOfValues) {
41    analogWrite(blueLed, meetAndroid.getInt());
42  }
```

Code Snippet 13: Arduino program for AmbientLight

In addition to the user interface, the Android application offers an interface for third-party applications. This interface allows other applications to change the color of the light by sending an **amarino.multicolorlamp.SET_COLOR** action intent with an integer representing the RGB color added as a "**value**" extra to the intent.

```
1  Intent intent = new Intent("amarino.multicolorlamp.SET_COLOR");
2  Intent.putExtra("value", Color.rgb(255,0,0)); // sets the color to red
3  sendBroadcast(intent);
```

After receiving the broadcasted intent, the AmbientLight app will connect to Arduino, change the color of the LEDs and disconnect immediately in case the connection is not needed. By providing an interface for third-party applications to the AmbientLight application, a developer could write for example a tiny Android application which changes the color of light by receiving a text message.

Figure 36: The Ambient Light

The AmbientLight is shown in Figure 36. The top left pictures displays the Arduino board with the LEDs soldered on it and a Bluetooth module connected to

the board. The cardboard box (top right) is used as a socket for the heart-shaped lamp shade hiding the electronics.

9.2 CallMyShirt

The second example, CallMyShirt, uses Amarino to communicate phone events to a wearable device. The wearable, shown in Figure 37 is a shirt with one big circle on its front consisting of ten luminescent pads. These individually controllable pads are used to display different patterns based on the phone's state. When the phone is ringing, all of the pads are blinking. If a call is accepted, the lights rotate in a circle until the user hangs up and when the phone is idle, the shirt generates random patterns. The shirt is also able to show the current battery level of the phone whereupon each pad stands for ten percent of battery capacity left.

The shirt was constructed from a LilyPad microcontroller and ten LilyPad LEDs which were sewn into the shirt with conductive thread. The microcontroller is connected to a Bluetooth shield for communication with the phone. Three main steps were necessary to realize this shirt: Step one is installing Amarino on an Android handset, pairing the LilyPad Bluetooth shield, and adding two events to it: *Phone State* and *Battery Level* as shown in Figure 37.

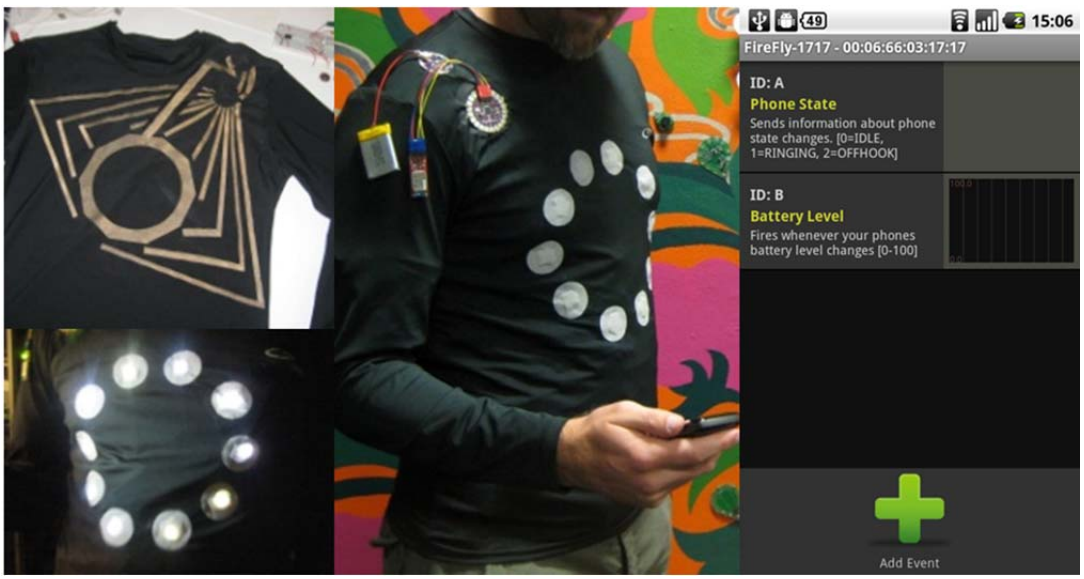


Figure 37: The inside of CallMyShirt with conductive traces (top left), all LEDs light up (bottom left), testing all functions before sewing remaining parts on the shirt (middle) and the corresponding event collection (right)

The second step is to write the Arduino program and upload it to the LilyPad. An extract of the code running on the LilyPad is shown in Code Snippet 14. The last step is making the shirt. I used stretchable conductive fabric ironed on the inside of the shirt to wire LilyPad output pins to the LEDs behind the semi-transparent pads. Once connection has been established between the LilyPad and the phone, the Amarino application on the phone can be closed. The connection is maintained and events are sent through Amarino's background process, thus not interfering with the user experience during daily phone operation.

```
#include <MeetAndroid.h>

//declare and initialize variables and constants
#define IDLE 0;
#define RINGING 1;
#define TALKING 2;
#define BATTERY 3;
#define NUM LEDS 10;

MeetAndroid meetAndroid;
int state;
int batteryLevel;
int leds[] = {3,4,5,6,7,8,9,10,11,12};

void setup() {
  for (int i=0; i<NUM_LEDS;i++) pinMode(leds[i], OUTPUT);
  // initialize the Bluetooth modem with 57600 baud
  Serial.begin(57600);
  // register Amarino callback functions
  meetAndroid.registerFunction(phoneStateCB,'A');
  meetAndroid.registerFunction(batteryCB,'B');
}

void loop() {
  meetAndroid.receive();
  switch (state) {
    case IDLE:    idle();         break;
    case RINGING: blinking();     break;
    case TALKING: circling();     break;
    case BATTERY: showBattery();  break;
  }
}

/*
 * Callback function for phone events
 *   called when a phone event is reported
 */
void phoneStateCB (byte flag, byte numOfValues){
  state = meetAndroid.getInt();
}

/*
 * Callback function for the battery level event
 *   called when the battery level has changed
 */
void batteryCB(byte flag, byte numOfValues){
  state = BATTERY;
  batteryLevel = meetAndroid.getInt();
}
```

Code Snippet 14: Arduino program for CallMyShirt

CallMyShirt proves that it is not required to write programs on Android at all to create compelling results using Amarino's built-in events only. Although experts definitely need more flexibility than built-in events can provide, for novices it is a good starting point to build interesting phone interfaces. The next project demonstrates the other direction of communication: sending sensor readings from an Arduino board to an Android application.

9.3 Workout

The final example, "Workout", was developed to help exercisers to keep track of their routines. It uses a motion sensing wearable and an Android application that allows wearers to visualize and analyze their activity level. The wearable is a knitted armband, shown on the left in Figure 38 that contains an embedded stretch sensor. The sensor was constructed knitting a piezo resistive yarn (visible as a silver gleaming middle part in Figure 38) into the fabric of the armband. When placed over an elbow, the armband can reliably detect when the elbow is bent. A LilyPad Arduino, again sewn into the band with conductive thread, reads sensor data from the band and relays it to a phone via Bluetooth.

Figure 38: Workout example made with the aid of Amarino
Left: the armband | right: Workout interface

The Android Workout application plots the raw sensor data it receives, extracts a number-of-arm-bends count from this data, and displays this count to the wearer.

A snapshot of this application in action is shown on the right hand side in Figure 38. In this example, Amarino is used to receive data from the armband and passes it along to the workout application.

Noteworthy, the complete prototype, both the wearable and the application, could be produced in a few hours. Amarino allowed it to develop and demonstrate the Workout proof of concept much more quickly than it could have been without it.

Chapter **10**

10 Conclusion

The software toolkit Amarino, which was developed over the course of this thesis, was released to the public in June 2010 with the aim of motivating and supporting researchers and hobbyists in creating compelling new interfaces for smartphones, tangibles and ubiquitous devices. As an open source project, Amarino was designed to be community-centric and therefore it is well documented, enabling users, from beginners to experts, to quickly produce working results. Shortly after the sources of Amarino were released, the Amarino website had more than 10,000 visits from over 100 countries. Nevertheless, it is still too early to draw a conclusion of how well Amarino will be picked up by the community. By analyzing relevant discussion forums, however, it seems that the Arduino community found a better way in approaching the toolkit compared to Android developers, while the Android developer community seems to be less interested in tinkering in general.

My personal appreciation is, that to this date the predominant portion of Amarino users are hobbyists. Inquiries by users of Amarino suggest that there is an interest for projects where things are being controlled with the phone, like steering a radio controlled hovercraft. In contrast, some developers focused on extending the smartphone interface. For instance, one of them reported about a device implemented with Amarino in less than a day. The device is a big red button on his table, similar to an emergency stop, which he hits to dismiss phone calls when he is not in the mood for talking. Clearly, these sample projects are made out of fun and are not likely to change the world we interface with computers. Given that Amarino users publish their work, which is very likely among the Arduino community, the creativity of people at their homes might be exhilarating for

product designers and researchers alike and might influence future directions of interface design and interaction modalities. Beside creations by tinkerers, a few interesting projects realized with the aid of Amarino by university students and researchers came to my attention; a glucose meter combined with a diabetes management application on Android; a small satellite project for NASA[21]; a glove helping people with arthritis to track joint temperatures; a weather station displaying its data directly on the phone screen. Apparently, Amarino is used in a wide range of research domains and has demonstrated convincingly its ability to support prototyping of mobile ubiquitous computing.

Apart from Amarino's ability to do most of its work in the background not interfering with other applications running on a phone, one of Amarino's strengths is its proposed plug-in technology and therefore seamless integration of third-party applications into Amarino. It can be expected that this technology will drive reuse of implemented code among device developers, since its straightforward integration into Amarino's user interface is as easy as installing an application on the phone. But still, Amarino has a lot of potential for improvement, both in performance and flexibility. By implementing support for Wi-Fi communication in addition to Bluetooth, many more prospective prototyping projects could be supported by Amarino. It was also thought about integrating Amarino into Google's recently released Android App Inventor, a visual programming language for Android. Furthermore, quite obvious, porting Amarino to other mobile operating systems and microcontroller platforms would increase its compatibility allowing more freedom when creating new interfaces.

In closing, I hope to see mobile ubiquitous computing further approaching in a way that computers become more seamlessly integrated into our daily life and environment, both by being assistive and unobtrusive or ideally even invisible, while offering easy-to-use and comprehensive services to human beings.

[21] National Aeronautics and Space Administration

Bibliography

[1] Weiser M., "The computer for the twenty-first century," *Sci. Am*, pp. 94-104, Sept. 1991.

[2] Amft O. and Lukowicz P., "From Backpacks to Smartphones: Past, Present, and Future of Wearable Computers," *IEEE Pervasive Computing*, vol. 8, no. 3, pp. 8-13, July-September 2009.

[3] Ishii H. and Ullmer B., "Tangible bits: towards seamless interfaces between people, bits and atoms," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Atlanta, Georgia, United States, March 22 - 27, 1997. S. Pemberton, Ed. CHI '97. ACM, New York, NY, 234-241.

[4] Sharlin E., Watson B., Kitamura Y., Kishino F., and Itoh Y., "On tangible user interfaces, humans and spatiality," *Personal Ubiquitous Comput.*, vol. 8, no. 5, pp. 338-346, Sep. 2004.

[5] Shaer O. and Jacob R. J., "A specification paradigm for the design and implementation of tangible user interfaces," *ACM Trans. Comput.-Hum. Interact.*, vol. 16, no. 4, pp. 1-39, Nov. 2009.

[6] Shaer O., Leland N., Calvillo-Gamez E. H., and Jacob R. J., "The TAC paradigm: specifying tangible user interfaces," *Personal Ubiquitous Comput.*, vol. 8, no. 5, pp. 359-369, Sep. 2004.

[7] Arduino, *http://arduino.cc*.

[8] Weiser M., "Some computer science issues in ubiquitous computing," *Commun. ACM*, vol. 36, no. 7, pp. 75-84, Jul. 1993.

[9] Scholtz J., "Ubiquitous computing goes mobile," *SIGMOBILE Mob. Comput. Commun.*, vol. 5, no. 3, pp. 32-38, Jul. 2001.

[10] Williams A., Farnham S., and Counts S., "Exploring wearable ambient displays for social awareness," in *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, Montréal, Québec, Canada, April 22 - 27, 2006. CHI'06. ACM, New York, NY, pp. 1529-1534.

[11] Winkler T., Ide M., Wolters C., and Herczeg M., "WeWrite: 'on-the-fly' interactive

writing on electronic textiles with mobile phones," in *Proceedings of the 8th international Conference on interaction Design and Children*, Como, Italy, June 03-05, 2009. IDC'09. ACM, New York, NY, pp. 226-229.

[12] Buechley L., Eisenberg J., Catchen M., and Crockett A., "The LilyPad Arduino: Using Computational Textiles to Investigate Engagement, Aesthetics, and Diversity in Computer Science Education," in *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI)*, Florence, Italy, April 2008, pp. 423-432.

[13] Werner J., Wettach R., and Hornecker E., "United-pulse: feeling your partner's pulse," in *Proceedings of the 10th international Conference on Human Computer interaction with Mobile Devices and Services*, Amsterdam, The Netherlands, September 02-05, 2008. MobileHCI'08. ACM, New York, NY, pp. 535-538.

[14] Labrune J. and Mackay W., "Telebeads: social network mnemonics for teenagers," in *Proceedings of the 2006 Conference on interaction Design and Children*, Tampere, Finland, June 07 - 09, 2006. IDC '06. ACM, New York, NY, pp. 57-64.

[15] Costanza E., Inverso S. A., Pavlov E., Allen R., and Maes P., "eye-q: eyeglass peripheral display for subtle intimate notifications," in *Proceedings of the 8th Conference on Human-Computer interaction with Mobile Devices and Services*, Helsinki, Finland, September 12-15, 2006. MobileHCI'06, vol.159. ACM, New York, NY, pp. 211-218.

[16] Fujiki Y. et al., "NEAT-o-Games: blending physical activity and fun in the daily routine," *Comput. Entertain.*, vol. 6, no. 2, pp. 1-22, Jul 2008.

[17] Kanjo E., Bacon J., Roberts D., and Landshoff P., "MobSens: Making Smart Phones Smarter," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 50-57, Oct 2009.

[18] Tokuhisa S. et al., "xtel: a development environment to support rapid prototyping of "ubiquitous content"," in *Proceedings of the 3rd international Conference on Tangible and Embedded Interaction*, Cambridge, United Kingdom, Feb 16-18, 2009. TEI'09. ACM, New York, NY, pp. 323-330.

[19] Oliver E., "A survey of platforms for mobile networks research.," *SIGMOBILE Mob. Comput. Commun.*, vol. 12, no. 4, pp. 56-63, Feb. 2009.

[20] Sanna S. and Saverio E. D., *Bluetooth API 1.0.*, http://code.google.com/p/android-bluetooth/.

[21] *Wiring.* http://wiring.org.co.

[22] Fry B. and Reas C., *Processing.* http://www.processing.org.

[23] Mellis D., *personal correspondence.*, January 2010.

[24] Boarduino, *http://www.ladyada.net/make/boarduino.*, last visited: August 12, 2010.

[25] Android developers, *http://developer.android.com*.

[26] The Official Google Blog, *http://googleblog.blogspot.com/2010/06/celebrating-android.html*., last visited: August 24, 2010.

[27] Androidlib, *http://www.androlib.com/appstats.aspx*., last visited: August 24, 2010.

[28] Meier R., *Professional Android 2 Application Development*, Wiley Publishing, Ed. Indianapolis, Indiana, US, 2010.

[29] Meier R., *Professional Android Application Development*, Wiley Publishing, Ed. Indianapolis, Indiana, US, 2009.

[30] Burnette E., *Hello, Android - Introducing Google's Mobile Development Platform.*: Bookshelf, Pragmatic, 2008.

[31] M. Banzi, *Getting Started with Arduino*, 3rd ed., O'Reilly Media - Make:Books, Ed.: O'Reilly, 2008.

[32] AVR Libc - C Library for use with GCC on Atmel AVR microcontrollers, *http://www.nongnu.org/avr-libc/*.

[33] HacknMod, *http://hacknmod.com/hack/top-40-arduino-projects-of-the-web/*., last visited: August 15, 2010.

[34] Hartman K., Faludi R., London K., and Bray R., *Botanicalls: The Plants Have Your Number.*: http://www.botanicalls.com.

[35] ThinkGeek, *http://www.thinkgeek.com*.

[36] Uncle Milton, *http://unclemilton.com/starwarsscience/*., last visited: August 15, 2010.

[37] Amarino, *http://www.amarino-toolkit.net*.

[38] Google Code - host for Amarino sources, *http://code.google.com/p/amarino/*.

[39] GNU General Public License, *http://www.gnu.org/licenses/gpl.html*.

[40] Google groups - discussion forum for the Amarino software toolkit, *http://groups.google.com/group/amarino-toolkit*.

[41] Jarczyk A. P. J., Löffler P., and Shipmann III F. M., "Design Rationale for Software Engineering: A Survey," in *25th Hawaii International Conference on System Sciences*, 1992, pp. 577-586.

[42] Gamma E., Helm R., Johnson R. E., and Vlissides J., *Design Patterns. Elements of Reusable Object-Oriented Software.*: Addison Wesley, 1995.

[43] Bluetooth SPP Specification, *http://www.bluetooth.com/SiteCollectionDocuments /SPP_SPEC_V12.pdf.*, last visited: August 12, 2010.