Fakultät für Informatik
der Technische Universität München

Bachelor’s Thesis in Informatics

Implementation of a HomeMatic simulator using Android

Johannes Neutze

Fakultät für Informatik
der Technische Universität München

Bachelor's Thesis in Informatics

Implementation of a HomeMatic simulator using Android

Implementierung eines HomeMatic Simulators unter Android

Author:	Johannes Neutze
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	Nils T. Kannengießer, M. Sc.
Date:	April 15, 2013

I assure the single handed composition of this bachelor thesis only supported by declared resources.

Munich, 15th of April, 2013 Johannes Neutze

Abstract

This bachelor thesis is about the development of an Android application, simulating the HomeMatic system and the XML-API v1.2, designed to control the HomeMatic system, e.g. through the HomeDroid client.
Configuration of the system, handshake with HomeDroid and persistent storage of the entire system status is implemented by using standard components, like a HTTP server, a relational database and an Android graphical user interface.
As a result, the user can connect to the application via HomeDroid and interact with it, like with the real hardware.
First, the foundation of emulated system and the environment used to develop the application, are explained. Then the system requirements are outlined and the architecture is illustrated.
Finally, the implementation of each component, the graphical user interface and its testing are described.

Table of contents

Acronyms and terms

- ADB - **A**ndroid **D**ebug **B**ridge [10]
- ADT - **A**ndroid **D**evelopment **T**ools [11]
- API - **A**pplication **P**rogramming **I**nterface
- App - **App**lication
- CCU -**C**entral **C**omputing **U**nit
- CGI - **C**ommon **G**ateway **I**nterface
- Eclipse - Open Source Development Software [14]
- EditText - Text box in Android GUI [8]
- GUI - **G**raphical **U**ser **I**nterface
- HMEmulator - HomeMatic Emulator application
- HomeDroid - Android client for HomeMatic systems [31]
- HomeMatic - Home automation hardware and software [19]
- HTTP - **H**yper**T**ext **T**ransfer **P**rotocol
- MIME - **M**ultipurpose **I**nternet **M**edia **E**xtension
- SDK - **S**oftware **D**eveloper **K**it
- spinner - selection box with drop down menu [5]
- Tcl - **T**ool **C**ommand **L**anguage
- toast - small information pop-up [6]
- USB - **U**niversal **S**erial **B**us
- XML - **E**xtensible **M**arkup **L**anguage
- XML API - HomeMatic Plugin [13]

I Introduction and foundation

1 Introduction

1.1 Scope statement

Create an emulator for a HomeMatic CCU including the XML API, a system database and a configuration module in 4 months using Android standard packages in Eclipse. The XML API implements the communication with the HomeDroid client.

1.2 Problem statement

The emulator allows to demo the functionalities of the HomeMatic CCU with an Android device without buying the dedicated HomeMatic hardware.

1.3 Outline

The following work describes the chosen architecture, its individual modules, the usage of the Android standard packages and the code developed to emulate the relevant parts of the CCU.

2 Foundation

2.1 HomeMatic

The HomeMatic System is a proprietary system produced by the German company eQ-3 and distributed in Germany and Switzerland [17]. The HomeMatic System includes the CCU, sensors and actors.

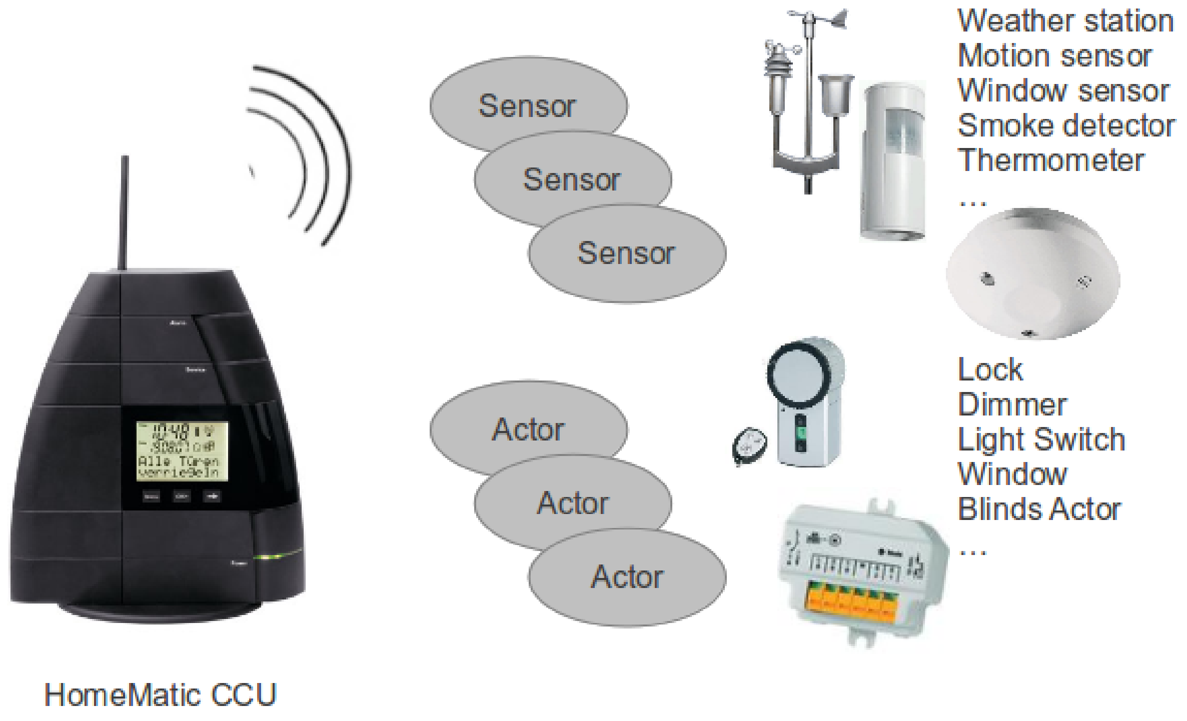


Figure 1. HomeMatic system components.

2.1.1 Logic structure

The CCU is a dedicated hardware running an operation system based on Linux with the HomeMatic software on top. It is responsible for the wired and wireless communication with the peripheral hardware and allows to automate tasks. The unit itself offers a small display with very rudimental functions. More complex tasks are supported via a web interface or PC software using direct USB connection. The HomeMatic software can be extended with plugins, e.g. for planning scheduled tasks or remote control.
Sensors and actors are external hardware. Sensors for example are smoke detectors, thermometers or door/window lock sensors. Switches, dimmers or valves are actors.
The whole system is modular and can be extended by adding hardware at any time. [21]

2.1.2 HomeMatic API

The hardware of the CCU is managed over the web interface which can be expanded via plugins using HomeMatic Script.
HomeMatic Script is an internal language for the HomeMatic CCU to access and modify the status information and system variables. It represents the internal base for the application interfaces.

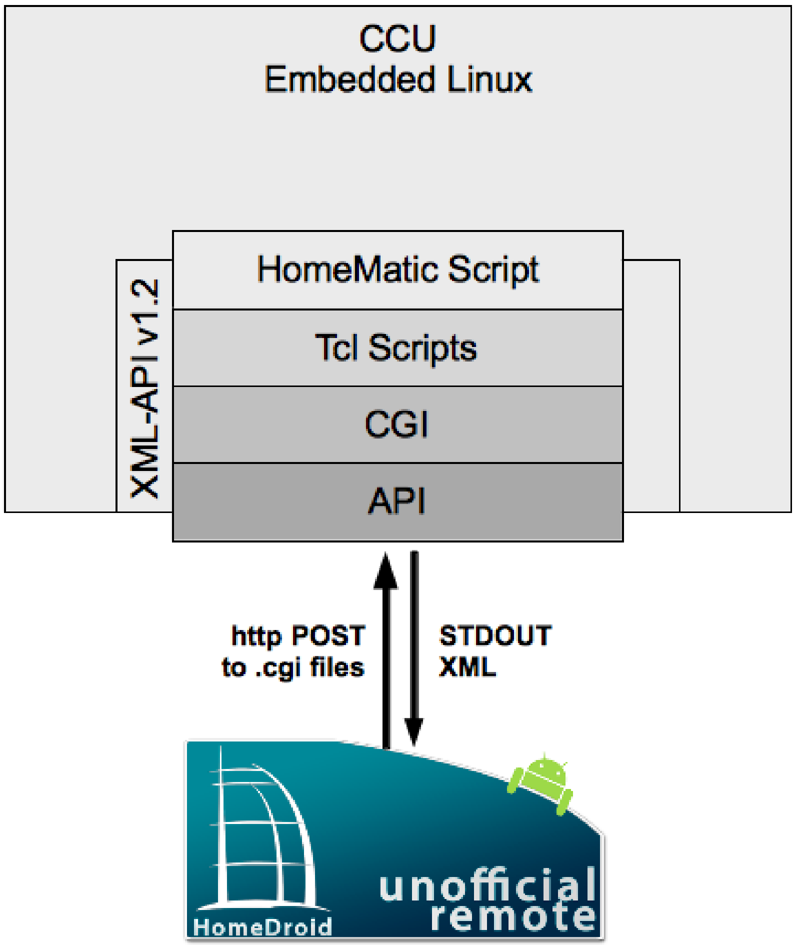


Figure 2. Structure and integration of the XML-API v1.2.

Tcl scripts are able to connect to the HomeMatic Script interpreter by including the tclrega.so library. Tcl scripts are then able to call HomeMatic Script functions via the *rega_script()* function and to read results via an array returned by *rega_script()*.

Tcl script enhanced by HomeMatic Script is used in *.cgi scripts to implement application interfaces (API). These scripts are accessible via standard HTTP calls.

The specific API (XML-API v1.2) emulated in this thesis provides a predefined set of *.cgi scripts with its functionalities. [20]

2.1.3 XML-API v1.2

The XML-API is installed as a firmware update via the web interface. It uses the stack of HomeMatic Script, *.tcl and *.cgi, described above, to provide the functionalities in the set of *.cgi scripts listed below. Applications access the scripts via HTTP calls, such as http://[CCU IP]/config/XMLAPI/[script name].cgi?ise_id=[channel id]&new_value=[value]. All data is returned in STDOUT, structured in XML format. [13]

Table 1. Available XMP-API v1.2 *.cgi scripts [13].

deviceList.cgi	returns list of all devices and its channels
functionList.cgi	returns list of all functional groups and its channels
roomList.cgi	returns all rooms and its channels
state.cgi	returns the current value of the requested channel
statechange.cgi	initiates the change of values for the specified channel
stateList.cgi	returns all values of channels
version.cgi	returns used version of the XML-API

The third party application used to verify the emulated system is HomeDroid. [31]

2.2 HomeDroid

HomeDroid is an Android remote application for the HomeMatic system. It is providing a GUI, assigning data to categories, like devices, rooms, functions and methods to manipulate it. This application uses the XML-API to communicate with the CCU.

2.3 Android

2.3.1 Android demographics

Android is the most popular mobile operating system. It is open-source and based on the Linux kernel. It is designed for touchscreen devices like mobile phones and tablets. [7]
The project to develop Android was financially backed and later bought by Google in 2005. Android was released synchronously to the founding of the Open Handset Alliance, a consortium of hardware, software and telecommunication companies, including Motorola, Qualcomm, T-Mobile and others. Their aim was to develop open standards for mobile devices. The consortium is led by Google and members of the alliance are not allowed to produce devices running incompatible versions of Android.
The Android code is released under the Apache Open Source License. This allows device manufacturers, wireless carriers and private developers to enhance the operating system. One reason for the large user and fan base. [29]
Android's popularity is reflected in the Play Store where most of the applications are distributed. Over 700.000 apps are available. [34]
Android applications are written in Dalvik Java, a modified version of Java, using the Android SDK.

2.3.2 Android SDK

The Android software developer kit is used for programming Android apps. It includes the necessary API libraries and developer tools to build, test and debug Android apps. It can be used on Linux, Mac OS and Windows, e.g. in Eclipse by using the Android Development Tools Plugin. [9]
Eclipse is an open source, multi-language software development environment which can be extended by plugins to provide additional languages and functions.
Android Debug Bridge, short ADB, is included in the SDK package. It is used for direct interaction between computer and Android device via USB.
The final version is compiled as a *.apk file on the computer by the Android SDK. Downloaded onto the device, it can be executed without the device being connected to a computer via ADB. Registered developers are allowed to upload the application to the Play Store.

2.3.3 Applications fundamentals

Android applications are written in Java and compiled to *.apk archives. An apk file is considered as one single program and used to install the application on an Android device.
If an application is installed on a device and started, it is executed in its own sandbox. A sandbox is an isolated computing environment where only this application runs in.
Basically Android OS is a multi-user Linux system in which each application represents a different user.
The system has a strict permission management allowing applications only to access its own resources and system functions.
Android starts the process when it is needed or shuts the process down when it is not used and memory has to be recovered. The memory management is done by Android in the background. This is realized in the application structure, described later.
This creates a very secure environment by using the “principle of least privilege”. [3]
All activities and privileges used by the application, like Internet, WiFi and SD card access, have to be defined in the manifest file of the application. These application rights have to be accepted on application installation by the user.
There are some ways to share data with other applications and for an application to access system services. One way is to use the same user id to access each other’s files. The applications have to be signed with the same application creator certificate to achieve this.

The graphical user interface consists of screens, represented by activities. Each screen is an activity on its own.
Activities have basically three states, resumed, paused and stopped. Resumed is also referred to as running and means the application activity is in the foreground. A paused activity is still alive and visible but another activity is shown. It still has its state and member information but it can be killed if extreme memory shortage occurs. If the activity is in the background and not visible by the user it is called stopped. It still holds state and member information but is more likely to be cleaned from the memory by the system. Activities can be stopped running by *finish()*, sent from the system or the app itself, or by the system forcing it. The application needs to have the data saving functions in the *onPause()* and *onStop()* methods to preserve its generated data.

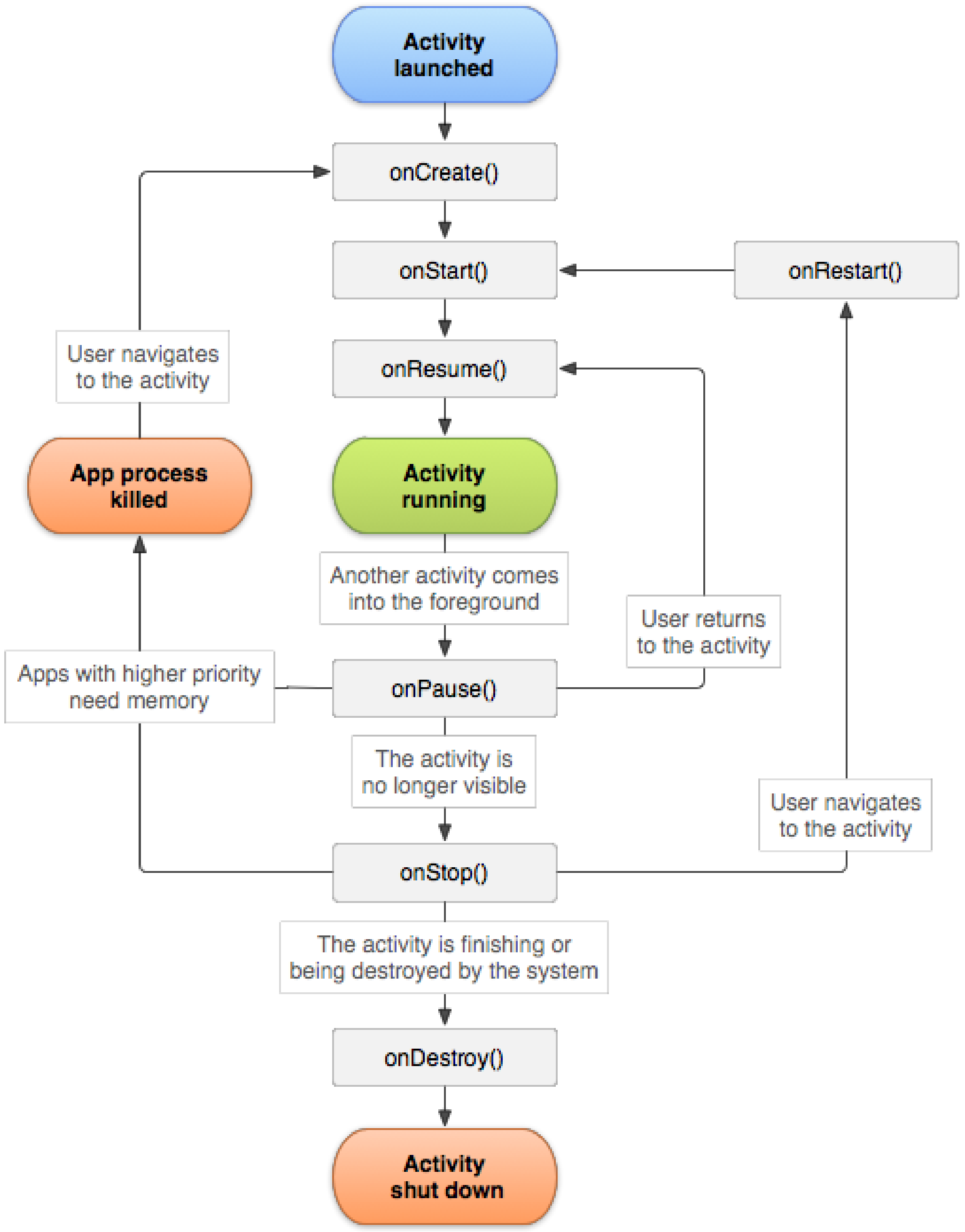

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 3. Activity lifecycle [2].

Beside the code an application also has resources, like photos, strings and audio files. These are declared in XML files. This helps to update different characteristics of the application without changing the code and to easily make it available to devices with different configurations, like screen size or language. The Android SDK assigns each resource a unique integer ID which is used to reference it. [3]

2.3.4 Memory model

There are different ways to store data in a persistent way so it does not get lost when the application is closed.

Primitive data is mostly stored in “sharedpreferences”, like the reference to a ringtone or settings. It is hidden from the user and is saved as long as the application is installed on the device.

Data can be saved in text files directly on the device’s internal storage. It is saved to the application folder and can only be accessed by the application. This is a very simple way of saving data because the data is outside of the app itself. It is persistent and save from outside access because it is in the application folder. All data is lost upon uninstalling the application.

Getting data into the application can be accomplished by choosing assets. Assets are files moved to the assets folder of the application before compiling the project. They can be accessed at runtime but are read only.

These are mostly rudimental ways of handling data. Complex operations on persistent data have to be more efficient than using text files or the “sharedpreferences”. That’s the reason why databases are used. They can provide data to all classes of the application but the data ca not be accessed from outside the application. Databases are realized by using SQLite databases. [4]

II Application implementation

3 System design

3.1 System requirements

REQ-1 Stable Android application

REQ-2 Emulating the CCU and the XML-API v1.2

REQ-2.1 Data exchange between application and HomeDroid
REQ-2.1.1 deviceList.cgi retrievable and working
REQ-2.1.2 roomList.cgi retrievable and working
REQ-2.1.3 functionList.cgi retrievable and working
REQ-2.1.4 state.cgi and stateList.cgi retrievable and working
REQ-2.1.5 statechange.cgi executable and working
REQ-2.1.6 version.cgi retrievable and working

REQ-2.2 Data storage
REQ-2.2.1 Creation of relevant entities
REQ-2.2.2 Persistency of data
REQ-2.2.3 Initial load of config database

REQ-2.3 Data modification
REQ-2.3.1 Interactive creation of real configuration
REQ-2.3.1.1 Editable devices
REQ-2.3.1.2 Editable rooms
REQ-2.3.1.3 Editable functions
REQ-2.3.1.4 Editable channels
REQ-2.3.1.5 Editable values
REQ-2.3.2 Visualization of positions and values of channels in the house

Not in scope is the functionality on mobile internet and the use real sensors and actors in connection with the application.

3.2 Architecture

The emulated system consists of five main parts. First part is the emulated the XML-API v1.2 providing the needed *.cgi commands. The second part is the web server, which handles the incoming requests from HomeDroid. Third part is the logic, responsible for processing input and output. The fourth part is the database, handling the persistent storage of the system data. And last but not least the graphical user interface with its possibilities to configure the system.

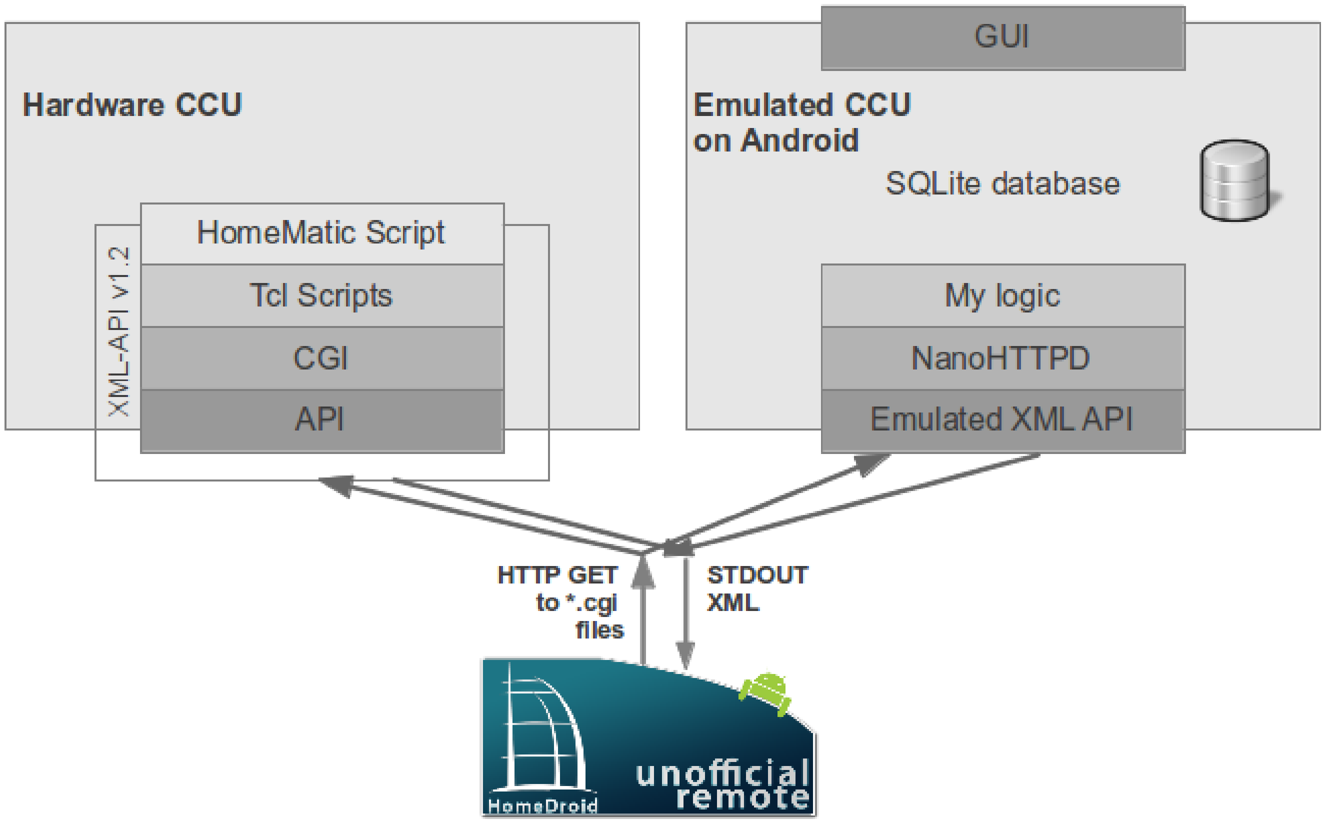


Figure 4. Real vs. emulated configuration.

3.2.1 NanoHTTPD

HomeDroid raises requests via HTTP and expects XML structured data as an answer. For this purpose a web server has to be implemented because the client requested format is in HTTP. The web server answers on these HTTP call with the requested data.
An integration into the application code is necessary so it should be preferably written in Java. Instead of using *.cgi as in the original hardware the server has to handle requests in combination with a database. Also multi-user support should be available.

To fit all of this requirements, NanoHTTPD is used. It is an open-source web server with a small resource usage, embeddable and modifiable. It was developed to be small and uncomplicated to embed into Java projects. It features only one Java file and is compatible from Java 1.1 up. It supports GET and POST methods as well as dynamic content and file serving but does not support *.cgi. [25]

HomeDroid raises requests as HTTP requests in the following scheme [33]:

```
GET /config/XMLapi/[script name] HTTP/1.1
User-Agent: Dalvik/1.6.0 (Linux; U; Android 4.2.1; Nexus 4 Build/JOP40G)
Host: 192.168.178.51:8765
Connection: Keep-Alive
Accept-Encoding: gzip
```

- Request line: The request line contains the path to the requested source and the HTTP version
- User-Agent: The user agent header contains the software name acting on behalf of the user. It contains information about the client. With this information the server can return special content for the different clients, like mobile sites for mobile devices or adjusted ads for the different operating systems. In this case the Nexus 4 with Android 4.2.1 Build JOP40G can be identified.
- Host: The host header contains the domain name of the server. It has to be included since HTTP/1.1 for name based hosts. If it is not present the server replies with a “400 Bad Request”.
- Connection: The connection header defines the connection type. In this case the connection should be “Keep-Alive that it has to be established only once.
- Accept-Encoding: The accept encoding header submits which encode formats are supported. Encoding is used to minimize the data size.
- Content-Type: On response, the server adds the MIME-type of the body to the header to inform the client that XML is used.

The real HomeMatic uses *.cgi scripts to return the data. These *.cgi scripts are written in perl and use HomeMatic Script to retrieve the data.
There are some reasons why this methodic was not transferred to the emulator.
Using *.cgi script would require to use an interpreter and would increase the footprint unnecessarily.
NanoHTTPD is not a stand alone web server but is embedded into the application. Being part of the application the NanoHTTPD is able to access the database directly so *.cgi scripts are not necessary to create dynamic content. Parameters and data can be simply passed inside the application and no external scripts are needed.

3.2.2 HomeMatic Emulator

In this implementation the *.cgi scripts are replaced by the web server transmitting the requests and parameters directly to the logic. The logic is part of the application code and uses SQL statements to gather the requested data.
The web server passes the request line to the ListHelper which does the work. The requests, the ListHelper can handle, are defined. The web server knows these GET requests and whether they are available. If the request is unknown, the NanoHTTPD replies with an error.
When the request is passed to the ListHelper, the matching method is called and executed. The method queries the required data from the database and then transfers into the XML code for the HomeDroid application.

3.2.3 Database

A database system is used to store data. It consists of a database management system and the stored data that should be managed. A database system guarantees persistence and consistency of data and provides an interface to insert, delete and manipulate information.
The stored data is physically saved on the hard disc or flash memory, as opposed to the volatile program variables only residing in RAM during process runtime.
The database uses tables to store the data. A table is a two dimensional representation of the data using columns and rows and has a unique name. [30]
Android provides full support for SQLite databases. SQLite is a relational database management system. Its “software library implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is an integral part of the client and not a separate process like other databases”. [32]
SQLite is an embedded database system and has an optimized size. Unlike common database systems it does not use separate processes for the server and the client. It might be the most widely used embedded database because of its use in application software such as web browsers and on most mobile operating systems, like Android or iOS.
It is published under public domain license.

3.2.4 SQLite on Android

SQLite is fully supported by Android and is available on every Android device. SQLite itself supports data types TEXT, INTEGER and REAL, similar to String, long and double in Java. All data has to be converted to these types in order to be saved in the database. SQLite has no mechanism to check whether the right format is inserted into a column.
The database itself does not require a setup and is managed by the Android system. For the database itself, *onCreate()* and *onUpgrade()* must be defined. Upon creation, the database is saved on the internal storage in the directory DATA/data/[application name]/database/[filename].
To use and work with databases on Android, the package android.database is needed. The package android.database.sqlite contains the SQLite specific classes.
While the management of the database is done by the Android system, the create and upgrade methods for the database need to be provided in the application code. Create, read and update methods are written when their functionality is needed.
The database is created with a class that is extended by SQLiteHelper. This class handles the

declaration and creation of tables and creation, updating and loading of the database. In the class constructor of the SQLiteHelper class the *super()* method is invoked. *super()* executes the default method with the parameters of database name and the database version to define the database object. The two functions *onCreate()* and *onUpgrade()* overwrite the methods of the SQLiteHelper class. *onCreate()* is called by the framework to create the database. In case it does not exist, o*nUpgrade()* is called if the database version, defined in the code, changed. It provides a way to change the database schema when the database or a table was edited in the code. These functions receive the database they should handle as parameter.
To read and write from and to the database, the methods *getReadableDatabase()* and *getWritableDatabase()* are used.
In addition to the respective data, a primary auto increment key column is inserted into each table. The key column is used to uniquely identify each record and used by most methods to access the data.
The content of the database is handled with another class. The DataSource class provides *insert()*, *update()* and *delete()* methods as well as *execSQL()* to directly execute SQL statements. ContentValues are used to insert and update database entries. Each attribute is added to the ContentValue and inserted into the database as one column with a generated key. Data can be accessed by using *rawQuery()* which accepts SQL statements, like SELECT * FROM table WHERE a=b, as input. Whereas *query()* has a defined interface for passing query components. Java and the environment is object orientated so even the query data is returned as an object. The cursor object saves the id of the first result in the table and can be moved to the next result row. The whole data can be accessed by *moveToFirst()* and *moveToNext()*. Finally with *isAfterLast()*, it is checked whether all elements were provided. By calling *getCount()* the number of rows can be received. With getter methods single columns and its attributes can be read. Before using the next Cursor object, it has to be released by calling *close()*. [1, 12, 27]

4 Implementation

4.1 Definitions

Below data types reflect the entities in the HomeDroid system and are stored in specific database tables. The tables are used to create the different lists. The HomeDroid app requests the data and the ListHelper processes the data. [15, 16]

Device - A device is a sensor or actor. It provides the basis for its channels and the channel's datapoints. The device can be seen as a container for its components, e.g. a blinds device has many channels, the blinds channels.
Channel - A channel is a part of a device. It contains the datapoints that define the type of the channel, e.g. a thermometer has two datapoints, temperature and humidity.
Datapoint - A datapoint contains a value of the the channel, e.g. a thermometer's temperature channel has 27.0 as value.
Room - A room is an area inside or outside the house. A Channel can be attached to a room to provide orientation where it is located. For instance a thermometer's channel has the attribute "Garden" as room, to indicate it somewhere in the garden.
Function - Channels can be grouped into functions. They will be summed up under one word, like all smoke detectors in the house can be united in the function "fire alarm system" . Each channel can be assigned to one function.

4.2 NanoHTTPD

The NanoHTTPD server starts on application startup. It is responsible for handling the HomeDroid requests. The server provides a port for the client to which it can connect.

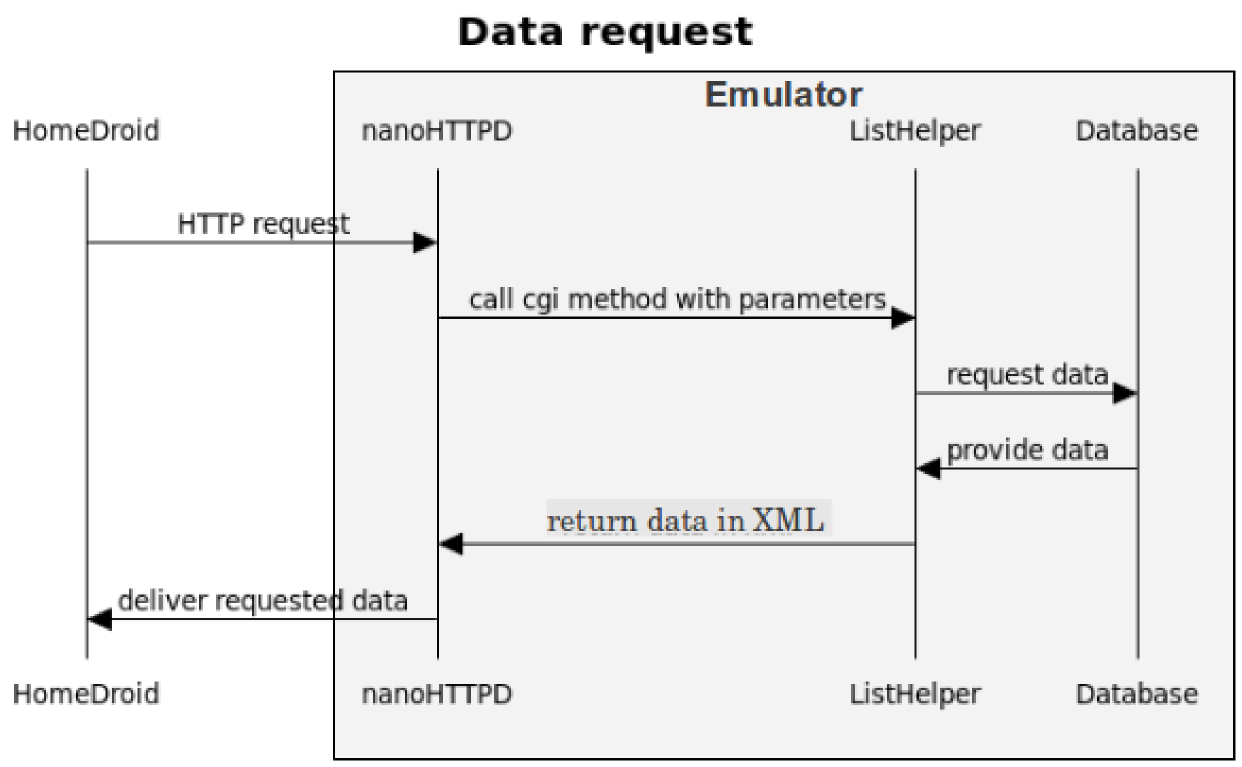


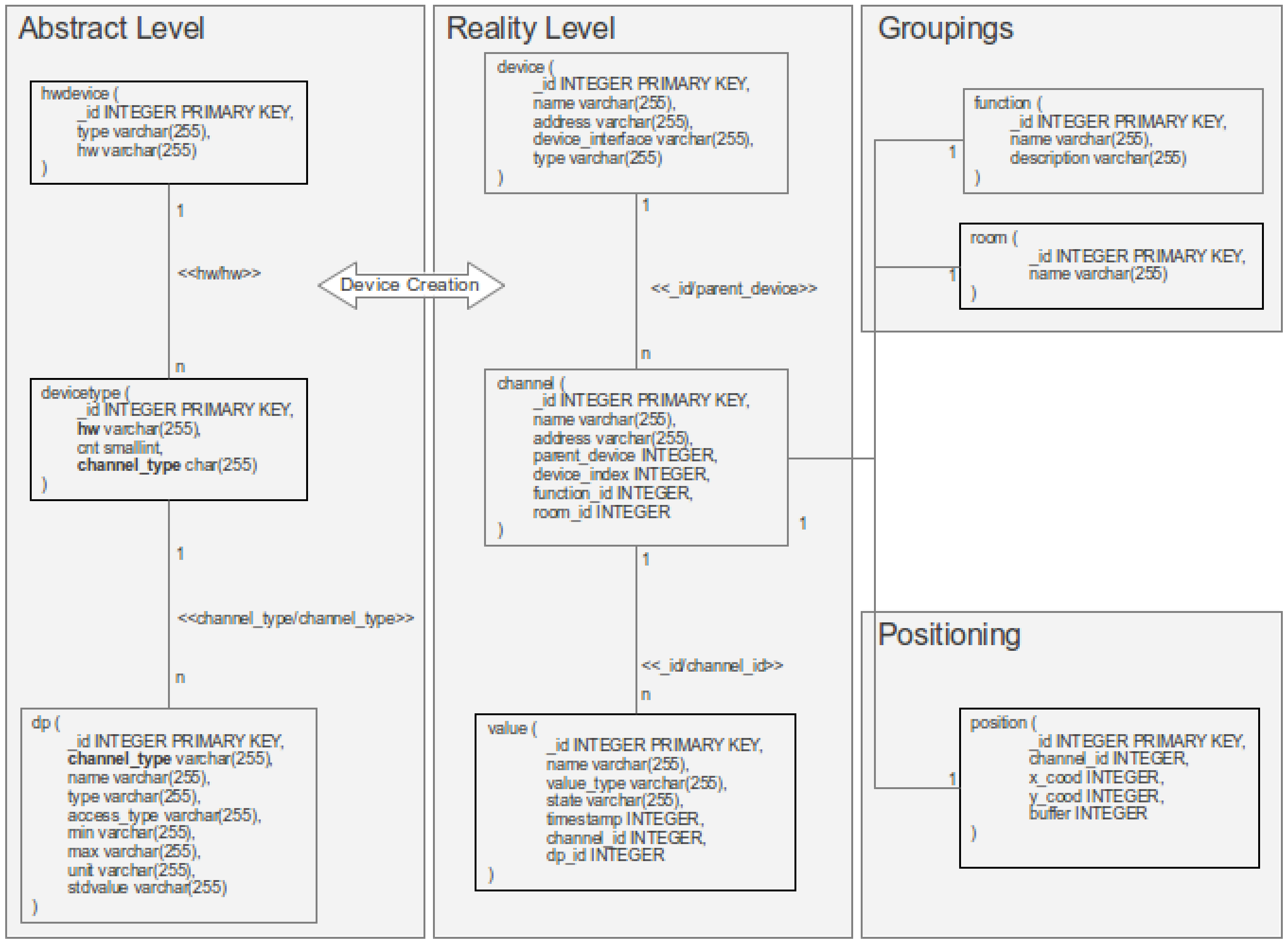
Figure 5. Sequence diagram of a HomeDroid request.

When the connection is established, the client sends requests in HTTP GET, as described above, to the server's IP address and port. The server buffers the request and parses it into the demanded *.cgi name and if available, the given parameters as well. HomeDroid requires the answer in a XML format. The ListHelper creates the demanded format and collects the needed data from the database by calling the *.cgi related method with the corresponding parameters. After the ListHelper created the XML list, the necessary HTTP header information will be added by the NanoHTTPD and responded to the HomeDroid client.

4.3 Data model

Before the function of the data handling and output can be explained, a closer look at the data model has to be made. The data is persistently saved in a database to which the the GUI and the ListHelper have access. The database is used to store the initiated devices and its components to make it editable and reachable for the clients. The data is split into three big categories, the Abstract Level, the Reality Level and the third level, which can be divided into Groupings and Positioning.

The initial database is located in the assets folder of the application. It contains the predefined tables which are transferred from the documentation of the HomeMatic system [15, 16]. Only devices and their components, provided as templates in the Abstract Level of the database, can be created. If the portfolio of devices should be expanded, new hardware devices have to be added and the database has to be recompiled. Then the database has to be transferred into the assets folder of the application in Eclipse and the apk file has to be generated.



```
hwdevice (
        _id INTEGER PRIMARY KEY,
        type varchar(255),
        hw varchar(255)
)

devicetype (
        _id INTEGER PRIMARY KEY,
        hw varchar(255),
        cnt smallint,
        channel_type char(255)
)

dp (
        _id INTEGER PRIMARY KEY,
        channel_type varchar(255),
        name varchar(255),
        type varchar(255),
        access_type varchar(255),
        min varchar(255),
        max varchar(255),
        unit varchar(255),
        stdvalue varchar(255)
)

device (
        _id INTEGER PRIMARY KEY,
        name varchar(255),
        address varchar(255),
        device_interface varchar(255),
        type varchar(255)
)

channel (
        _id INTEGER PRIMARY KEY,
        name varchar(255),
        address varchar(255),
        parent_device INTEGER,
        device_index INTEGER,
        function_id INTEGER,
        room_id INTEGER
)

value (
        _id INTEGER PRIMARY KEY,
        name varchar(255),
        value_type varchar(255),
        state varchar(255),
        timestamp INTEGER,
        channel_id INTEGER,
        dp_id INTEGER
)

function (
        _id INTEGER PRIMARY KEY,
        name varchar(255),
        description varchar(255)
)

room (
        _id INTEGER PRIMARY KEY,
        name varchar(255)
)

position (
        _id INTEGER PRIMARY KEY,
        channel_id INTEGER,
        x_cood INTEGER,
        y_cood INTEGER,
        buffer INTEGER
)
```

Figure 6. Structure of the database tables.

4.3.1 Abstract Level tables

The Abstract Level contains the information needed for the creation of individual devices with their channels and values. The Abstract Level tables contain information about hardware available for the HomeMatic system, like the associated number of channels and the matching value type for channels. The HMEmulator can only handle these predefined hardware components. [15, 16]

4.3.1.1 HWDevice table

This table contains the information about hardware models provided by HomeMatic and is initially loaded from the *assets*. The data has been taken from the HomeMatic documentation [16]. When initiating an actual device based on the hardware model, the device type is used to determine the channel attributes. That’s the reason why only hardware models in this table can be instantiated into actual devices.

Table 2. HWDevice table.

_id	The unique identifier. It is automatically created by database and incremented for each new table record. e.g. 42
type	The device type. e.g. temperature-humidity-sensor
hw (key)	The hardware model name given by HomeMatic. e.g. HM-WDS10-TH-0

The hw column is used to join the table with the Devicetype table’s hw column.

4.3.1.2 Devicetype table

The Devicetype table contains the different channel types and the number of associated channels. The count indicates how many channels of the given channel_type a device has. For example blinds actor come with one to two channels to control the blinds. On device creation in “Edit Devices”, the necessary channels are created. Depending on the channel_type, the relevant datapoints are created in turn [16]. Device types are e.g. Weather or Dimmer. The table is loaded from the *assets*.

Table 3. Devicetype table.

_id	The unique identifier. It is automatically created by database and incremented for each new table record. e.g. 42
hw (foreign key)	The hardware model name. e.g. HM-WDS10-TH-0
cnt	The number of channels of channel_type for the device. e.g. 2
channel_type (key)	The type reference for the device. e.g. WEATHER

The Devicetype table is linked to the HWDevice table via the hw columns and to the datapoint via channel_type.

4.3.1.3 Datapoint table

The Datapoint table is also loaded from the *assets* and contains the data defined in the HomeMatic documentation [16]. The datapoints are the variables and values a channel has. On device creation all datapoints belonging to the device type are created in the Value table.
The table specifies the value type, its format, its access type, min and max value, unit and the standard value.

Table 4. Datapoint table.

_id (key)	The unique identifier. It is automatically created by database and incremented for each new table record. e.g. 42
channel_type (foreign key)	Devicetype using this datapoint. e.g. WEATHER
name	The name of the value. e.g. Humidity
type	The value type as number. As not documented and not needed by HomeDroid, i is populated with a dummy. e.g. float
access_type	Direction of communication. e.g. sender or receiver
min	Minimum value of the state.
max	Maximum value of the state.
unit	Type of the value. e.g. percentage
stdvalue	Default value. e.g. 0.0

The Datapoint table is linked to the Devicetype table via channel_type.

4.3.2 Reality Level tables

The Reality Level instantiates the “real” devices. In contrast to the abstract level tables , these tables contain all devices actually present in the current configuration, not the archetype devices . These would be the devices present in “real life” in the HomeMatic system, the ones which are existing and not only known to the system.

4.3.2.1 Device table

The Device table contains the actually existing devices of the emulated system with a name given by the user. Device creation instantiates a hardware device in the table. Devices can be created and deleted in the “Edit Devices” menu in the top right corner of the Main Activity of the GUI.

Table 5. Device table.

_id (key)	The unique identifier. It is automatically created by database and incremented for each new table record. e.g. 42
name	Name given by the user. e.g. Garden Thermometer
address	The device address for clients. It is starting with “JEQ” followed by a seven digit number consisting of zeros and the device _id at the end. e.g. JEQ0000042
device_interface	The hardware interface name. As this address is not documented and not needed for the HomeDroid, it is populated with a dummy. e.g. BidCos-RF
device_type	Hardware references name. e.g. HM-WDS10-TH-0

The device is linked to the Channel table via the (device) _id and the (channel’s) parent_device.

4.3.2.2 Channel table

The Channel table instantiates the channels of an actual device, e.g. a thermometer has one channel which provides the temperature and humidity and a blinds device can have several blinds channels. The number and type of channels created for a device is defined in the Devicetype table.

Table 6. Channel table.

_id (key)	The unique identifier. It is automatically created by database and incremented for each new table record. e.g. 42
name	Name of the channel. It is generated on creation and consisting of type, address and index. It can be renamed by user. e.g. HM-WDS10-TH-0 JEQ00000042:1
address	The channel address for clients. It consists of device address plus the index of the channel. The address is used by the HomeDroid client to assign channel and device. e.g. JEQ00000042:1
parent_device (foreign key)	Id of the device the channel belongs to. e.g. 42
device_index	A running number of the channel for the device. Starts with 1. The number of channels of the device is defined in Devicetype table e.g. 1
function_id (foreign key)	Id of the function the channel is assigned to. e.g. 2
room_id (foreign key)	Id of the room the channel is assigned to. e.g. 5

The channel is linked to its device via parent_device and the room as well as the function are linked via room_id and function_id.

4.3.2.3 Value table

In the Value table the actual status of the channels are instantiated. Each channel type has its own datapoints. They are created on channel declaration and saved in the Value table. The status of a datapoint can be remotely modified by using *statechange* or direct via the GUI of the HMEmulator.

Table 7. Value table.

_id (key)	The unique identifier. It is automatically created by database and incremented for each new table record. e.g. 42
name	Name of the value. e.g. temperature
value_type	The type of the value. Boolean, integer float are supported and depending on the datapoint. e.g. 4
state	Current value of the state. e.g. 25 (°C)
timestamp	System generated timestamp of the last edit.
channel_id (foreign key)	Id of the channel the value record belongs to. e.g 42
dp_id (foreign key)	Id of the datapoint the value record inherits from. e.g. 42

The Value table is linked to the Channel table via the channel_id.

4.3.3 Groupings and Positioning Level

Groupings are present in the real HomeMatic model. They consist of Functions and Rooms. Functions summarize different channels under one term. Whereas rooms are a spatial grouping of the channels, functions are functional grouping.
Positioning realizes the visualization for the emulator and is not present in the original HomeMatic.

4.3.3.1 Function table

The Function table contains the names and descriptions of the different functional groupings of sensors. Functions can be edited in the “Edit Functions” menu. The function can be named and given a description for humans to identify it more easily. The function can be assigned to the channel on “Assign Channel”.

Table 8. Function table.

_id (key)	The unique identifier. It is automatically created by database and incremented for each new table record. e.g. 42
name	User given name of the function. e.g. all blinds
description	User given description of the function. e.g. easy and fast access to all blinds

The channel is connected via the function_id.

4.3.3.2 Room table

The Room table contains the different rooms available in the simulated house. Rooms can be added to the system in the “Edit Rooms” menu of the Main Activity. In “Assign Channel”, channels can be tied to rooms and their _ids. Please note, rooms are a logical grouping of channels whereas positions are virtual locations for single channels.

Table 9. Room table.

_id (key)	The unique identifier. It is automatically created by database and incremented for each new table record. e.g. 42
name	User given name of the room. e.g. living room

The channel is connected via the room_id.

4.3.3.3 Position table

The coordinates of the channel in the virtual house are saved in the Position table. On position assignment the channel_id, coordinates and size are stored in the table.

Table 10. Position table.

_id	The unique identifier. It is automatically created by database and incremented for each new table record. e.g. 42
channel_id (foreign key)	Id of the channel the position represents. e.g. 42
x_cood	x-coordinate of the location of the channel in the two dimensional map. e.g. 133,7
y_cood	y-coordinate of the location of the channel in the two dimensional map. e.g. 133,7
size	Size of the channel location in pixels. e.g. 100

The position is linked to its channel via channel_id.

4.4 Database

All these tables are part of the database building the foundation of the HomeMatic emulator. The Abstract Level includes the basic data needed to create the devices and its components. The Reality Level is used to save the user created and modified content and initially empty. The Groupings and Positions are also initially empty.
The abstract level tables are predefined upon the installation of the app. The data needed for the HMEmulator was extracted from the documentation [15, 16]. It was written into a *.csv file. In order to make the creation reproducible, a shell script was created with all the SQLite statements needed to create the tables and populate them with the data from the *.csv file.
This database is added to the assets folder of the Android application. A pre created database was chosen because it is very uncomplicated to create a database out of this data and to enter a big amount of data very quickly via a *.csv file. It is more comfortable to edit and add configuration data to the database than doing it directly in the Android application's code.
On first startup, the database is read and copied from the assets folder into the application's own database. The database already contains the empty tables of the Reality and Grouping level which are needed for the application. If adding new hardware models or device types becomes a recurring process and user created data should not be lost, the upgrade method has to be modified. [26]

4.4.1 Database implementation

The implemented database consists of two classes and one class for each table object. The DatabaseHelper extends the SQLiteHelper class and is responsible for the creation, editing and updating of the database itself. The DataSource class contains the methods to handle the tables. The table object classes provide the setters and getters to handle the attributes of the objects created by the DataSource from the tables. The setters are used to assign attributes to an object, while getters are used to receive the requested attribute.
The tables names and columns are known as strings in the DatabaseHelper class to get access to them. It checks if there's already a database of the app with the same version number in the application folder on the device. If it does not exist or is of an older version, the database from the asset folder is copied into applications database.
The DataSource contains all the methods needed to access the tables of the database. To edit the database, it has to open it first to gain write rights. This is achieved by using a DatabaseHelper object. The methods included in this class are the read, insert and update functions which are allowed for each table, e.g. there is no delete or insert method for the HWDevice table because it stays unmodified.
The Java code uses cursors to point to a database record which are then converted to an object. Each table has its own object and queries are returned as a list of results created with the cursor. To create objects, classes with setter and getter methods for the object are needed. The cursor is converted to the object via a *cursorToObject()* method which sets the position of the cursor in the record to the related attribute in the object. These objects are used throughout the Java code.
DatabaseHelper and DataSource are the basis for the GUI and database access. [28]

4.4.2 ListHelper

The ListHelper is the functionality which emulates the *.cgi scripts. Its job is to gather the requested information for the NanoHTTPD to pass to the client via STDOUT.
Thus the ListHelper class has a replacement method for each needed *.cgi used by HomeDroid. The methods query the database to receive the necessary data for the lists. The data is merged with the XML format the original XML API delivers.

Only the information necessary for HomeDroid were added to the table, so some parts of the responses are static. This data was re engineered from the HomeMatic documentation, the XML-API, the real hardware and by using HomeDroid.
First, the XML-API was analyzed. The scripts were downloaded and the XML related content was extracted [18]. This theoretical way was used, because in the beginning no real hardware was available. With this information, the structure, like the different *.cgi lists, were modeled. They were tested as dummy lists to check the NanoHTTPD functionality in combination with the HomeDroid client. Information about the HomeDroid requests was gathered by printing the HTTP GETs it sends. With this information the methods for the NanoHTTPD to reply with the right XML code were created.
The lists were phrased from the extracted code and by using existing lists from other users posted in forums [22, 23, 24].
Later the hardware was available and lists were directly accessible and could be inspected in connection with the real hardware.
Second the implementation of the data model created from gathered information was performed. The lists were split up into the different tables. The tables have been inserted into the database and the dynamical creation of the *.cgi request answers was implemented.
After the foundation was working, the Abstract Level tables were added. The data for these tables has been taken directly from the documentation [16]. This was done after being sure the HomeDroid accepted the data that had been taken from the available hardware.
With these tables it was possible to implement the statechange.cgi method and its functionalities.
The Position table was added with the graphical user interface because it has only HMEmulator specific information.
All *.cgi’s, except state and statechange, require no parameters. State requires an _id to return the channel’s current status. Whereas statechange needs a datapoint _id to change the current status of a datapoint. Parameters are specified in a name = value fashion standard in HTTP GET requests. Script request with parameters are looking like this:

```
[script name].cgi?ise_id=[_id]&new_value=[value]
```

The responses always start with the XML header “<?XML version="1.0" encoding="ISO-8859-1" ?>” followed by the XML content. The content is generated in each *.cgi replacement method. Not all fields of the lists are relevant to the HomeDroid thus some fields have no content.

When deviceList.cgi is requested, the deviceList method is executed. It collects all instantiated devices from the Device table with its related channels. All devices and channels are requested from the database, matched and returned in the proper XML format.

```
<deviceList>
        <device name="[device.name]" address="[device.address]" ise_id="[device._id]"
        interface="[device.device_interface]" device_type="[device.type]" ready_config="true">
                <channel name="[channel.name]" type="" address="[channel.address]"
                ise_id="[channel._id]" direction="" parent_device="[device._id]" index="[channel.index]"
                group_partner="" aes_available="false" transmission_mode="DEFAULT" visible="true"
                ready_config="true"/>
                <channel  ... />
        </device>
        <device ... />
</deviceList>
```

Figure 7. deviceList example.

The functionList.cgi contains all Functions from the Function table with its related channels. It is created by the roomList method.

```
<functionList>
        <function name="[function.name]" description="[function.description]" ise_id="[function._id]">
                <channel address="[channel.address]" ise_id="[channel._id]"/>
                <channel ... />
        </function>
        <function ... />
</functionList>
```

Figure 8. functionList example.

The roomList.cgi returns all Rooms and the corresponding channels via the roomList method. Rooms are stored in the Room table and the channels are stored in the Channel table.

```
<roomList>
        <room name="[room.name]" ise_id="[room._id]">
                <channel ise_id="[channel._id]"/>
                <channel ... />
        </room>
        <room ... />
</roomList>
```

Figure 9. roomList example.

The stateList.cgi calls the stateList method. The stateList provides information about the current state of all datapoints. The datapoints are grouped with their channels and the channels with their devices. The values of the datapoints are stored in the Value table.

```
<stateList>
        <device name="[device.name]" ise_id="[device._id]">
                <channel name="[channel.name]" ise_id="[channel._id]">
                        <datapoint name="[channel.name].[value.type]" type="[value.type]"
                        ise_id="[value._id]" value="" valuetype="" timestamp="[value.timestamp]"/>
                        <datapoint ... />
                </channel>
                <channel ... />
        </device>
        <device ... />
</stateList>
```

Figure 10. stateList example.

The state.cgi is almost the same as the stateList except it only returns information of the requested device with its components.

The statechange.cgi is responsible for changing values. The NanoHTTPD passes the _id of the channel to change and the new value to the statechange method. This method modifies the data in the database and returns the _id of the channel it just changed.

```
<result>
        <changed id="[channel._id]"/>
</result>
```

Figure 11. statechange example.

The version.cgi method returns the version of the emulated XML API.

```
<version>
        1.2
</version>
```

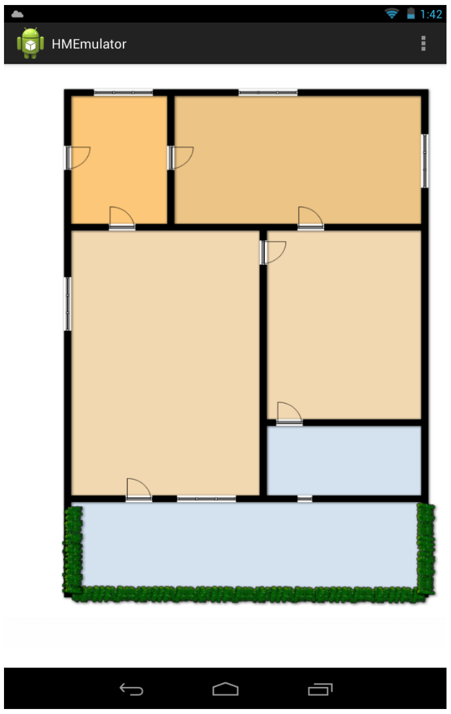
Figure 12. version example.

4.5 GUI

The graphical user interface consists of four activities. The Main Activity, the “Create” activity with dynamical content for each menu parameter, the “Assign Channel” activity and the “Edit Channel” activity.

4.5.1 HomeMatic Emulator Main Activity

The HomeMatic Emulator Main Activity is the standard activity and will be displayed on startup of the application. It represents the virtual house in a graphical form and there is access to all application functions from here. The menu is, as usual for Android 4.x, in the top right corner. It provides four options, namely “Edit Devices”, “Edit Rooms”, “Edit Functions” and “Connection”. The “Edit” menu points open the “Create” activity while the “Connection” displays a toast, i.e. message box on the screen.



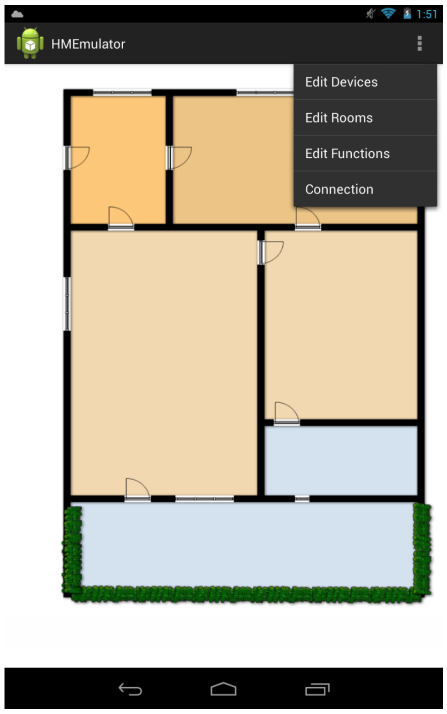


Figure 13. Main Activity. **Figure 14. Main Activity with expanded menu.**

By touching anywhere on the screen the “Assign Channel” activity appears. If a channel is already drawn on the screen and the screen is touched at roughly its coordinates, the “Edit Channel” activity will be called. The background of Main Activity is the ground plan of a virtual house and positions of the channels can be assigned and shown on it. If a channel is assigned, there’s an entry in the position database. Each time the activity comes in the foreground it will be checked whether there’s a new entry in the Position table. If there’s an entry, a red point is drawn at the coordinates contained.

4.5.2 Edit devices

By clicking the menu item “Edit Devices” the “Create” activity with device parameters is started[27]. This activity enables the creation of new devices and the deletion of existing devices. A device is an instantiation of a hardware offered by HomeMatic.

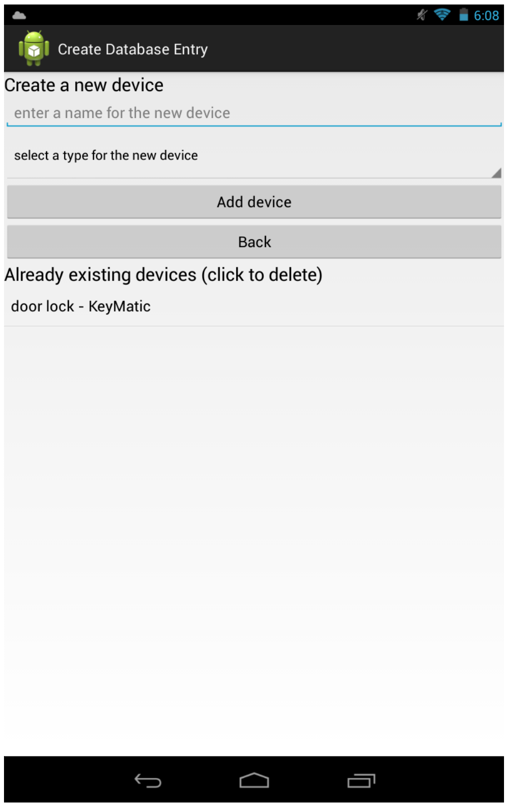

6:08
Create Database Entry
Create a new device
enter a name for the new device
select a type for the new device
Add device
Back
Already existing devices (click to delete)
door lock - KeyMatic

Figure 15. “Create” activity with devices setup.

To create a device, a name has to be entered in the EditText field and a hardware model has to be chosen in the spinner. The spinner shows the different types of devices contained in the HWDevice table. The add button verifies whether a name was entered and a device type was chosen, if one of the two is missing, an error toast is displayed. The back button closes the activity and recalls the Main Activity.

When a new device should be created and the add button has verified the two values correctly, the creation process starts. First the device is created by taking the strings from the EditText field and the current spinner position and using them as name and hardware type for the Device table entry. The device_interface is filled with the standard “BidCos-RF” and after the insertion into the database, the address is modified to match the standard string “JEQ” with 7 digits consisting of the device id filled with zeros in front.

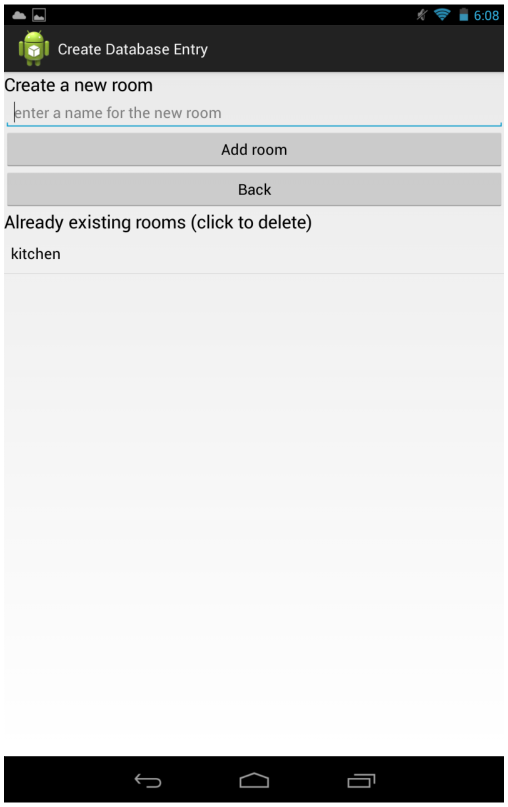
After the device is created and stored in the Device table, the *createChannels()* method is called with the new device as parameter. The method checks in the Devicetype table how many channels have to be created for this type of device and what type the channels are of. The information of the datapoints to be created for each channel, the amount of channels and the

parent device are then passed on to the *createChannel()* method.
The *createChannel()* method creates a generic name and address for the channel by concatenating the device type, the address and the index for the name. The device _id is used as parent_device entry to link the channel to the device. The index is the running number of the channel of the device. It starts with 1 and is incremented for each channel created. The number of channels created is passed from the *createChannels()* method as well. Each record will be saved in the Channel table.
Finally the datapoints for the channels have to be created. They are stored in the Value table. The datapoints are passed on as a List from the c*reateChannel()* method, which calls the *createValue()* method with the datapoint list and the created channel. The method creates the entries for the Value table from the datapoints and adds the channel _id of the related channel and a timestamp to identify the last modification of the record.
The device and its components can now be listed in the *.cgi requests of the HomeDroid client.

The deletion of a device from the application is performed by clicking on a list item in the device list below the buttons. It reverses the sequence of device creation. On touching the device which should be deleted, the *deleteDevice()* method is called. The device clicked is passed to this method, which deletes its entry in the Device table. After that the *deleteChannel()* method is called with the same device _id to delete all channels with this device as parent_device. The same happens with the value and Position table records. The position of the channel is deleted by using *deletePosition()* and for each channel's value the *deleteValue()* method for the channel _id is called.

4.5.3 Edit rooms and edit functions

When the “Edit Rooms” option is selected in the Main Activity menu, the “Create” activity is opened with the room parameter, meaning rooms can be created and deleted.


Create Database Entry

Create a new room

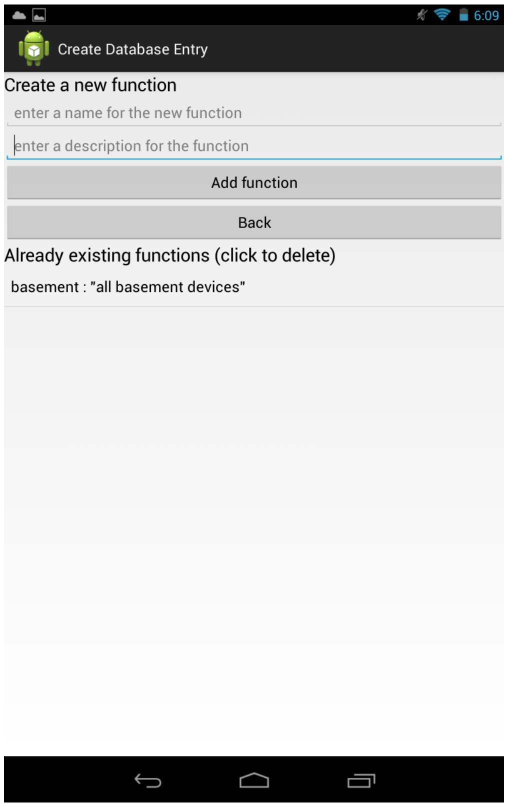
enter a name for the new room

Add room

Back

Already existing rooms (click to delete)

kitchen


Create Database Entry

Create a new function

enter a name for the new function

enter a description for the function

Add function

Back

Already existing functions (click to delete)

basement : "all basement devices"

Figure 16 “Create” activity with rooms setup.

Figure 17. “Create” activity with functions setup.

This activity interface structure consists of a EditText field for the room name, a add button to add the room to the table if a name is entered, a back button to return to the Main Activity and a list of existing rooms. Existing rooms can be deleted on touch, provided they are not assigned to a channel.

When a room name is entered and the add button is touched, a new room is created by calling the *createRoom()* function with the name as parameter. The name is set as name attribute and its _id is created.

When an existing room is touched for deletion, it is first checked whether a channel is assigned to the room. If so, an error toast is displayed, else the *deleteRoom()* method for this room is executed and the room is deleted.

The same rules apply for functions and their *createFunction()* and *deleteFunction()* methods, except that the description is optional. The function records are stored in the Function table.

4.5.4 Connection information

The fourth menu option is the “Connection”. It shows the current wireless network IP address of the device running HMEmulator and the port of the NanoHTTPD. This IP address has to be entered in the HomeDroid app to connect to the HomeMatic Emulator application.
The WiFi IP is created by using the Android WifiManager, an Android class to manage WiFi related aspects.

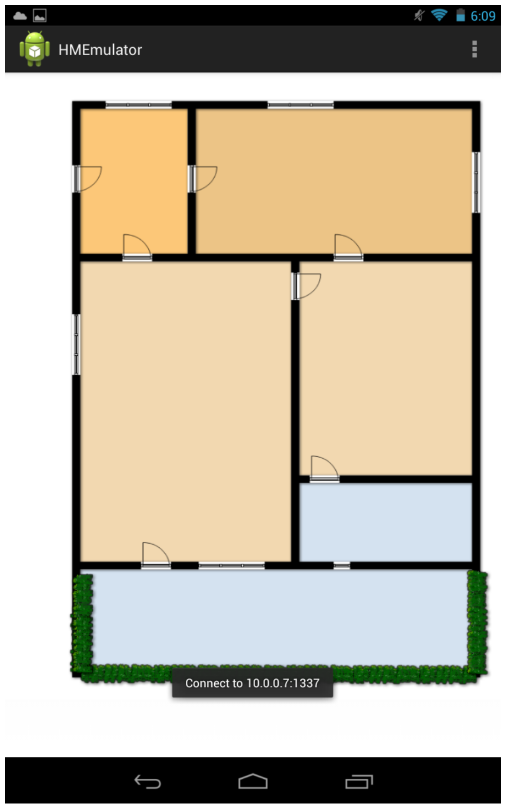
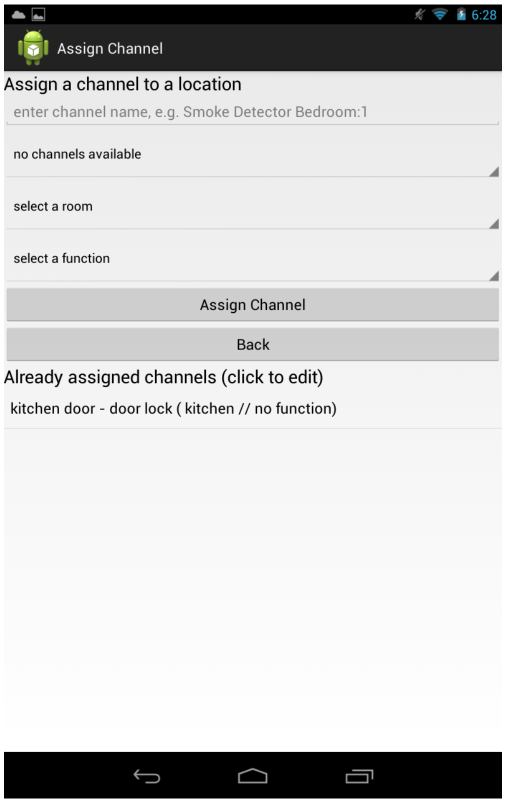


Figure 18. Main Activity with “Connection” information.

4.5.5 Assign channel

The emulator can visualize the channels of the device, i.e. placing a channel at a certain position in the virtual house, allowing to interact with it. This is done by clicking on the position of the background of the Main Activity where the channel should be placed. When a spot is touched, the coordinates are captured and the “Assign Channel” activity will start.



Already assigned channels (click to edit)

kitchen door - door lock (kitchen // no function)

Figure 19. “Assign Channel” activity.

In this activity, an existing channel can be chosen and given a name. The name has to be entered in the EditText field. The channel can also be installed in a virtual room, which was created in the “Edit Rooms” menu. Likewise a channel can be assigned to a function which can be created in the “Edit Functions” menu. Both is done by selecting an existing room respective function from a spinner.

When the “Assign Channel” button is activated, it is checked whether a name was entered and channel was picked. Assigning a function to a channel is optional.

If everything is correct, the *createPosition()* is called with the parameters of the new name, the chosen channel, the coordinates, a size for the dot representing the channel, the new room and the new function. The method creates a new position entry into the Position table containing the channel _id, x and y coordinates and the size for the visualization. The size parameter is used to set the diameter of the dot drawn on the Main Activity background and the area around the point counting as the dot, too.

The method also updates the channel with the new name, room and function. After creating the positioning dot of the channel, the activity closes and the next channel can be assigned.
In the “Assign Channel” activity below the buttons, there is a list with already assigned channels. They are listed with their name, the name of their parent device, name of their room and their function. They can be edited in the “Edit Channel” activity when touched.

4.5.6 Edit channel

The “Edit Channel” activity is used for editing channels and their values, already to a position assigned. They can also be deleted in this activity.

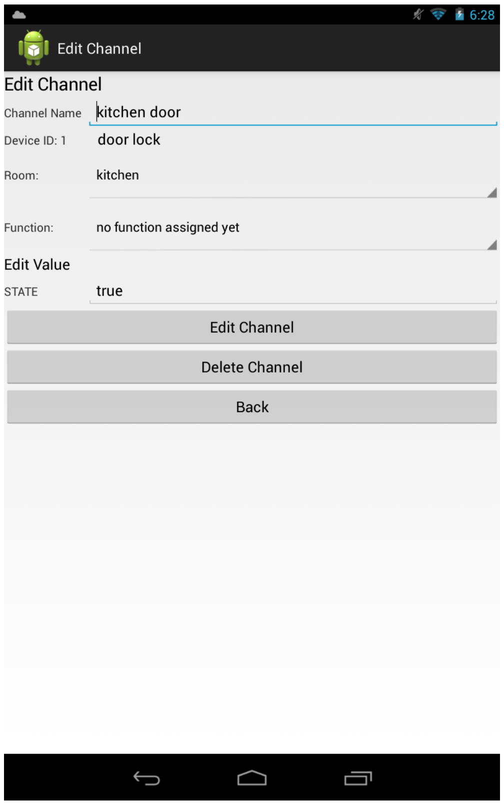


Figure 20. “Edit Channel” activity.

On pressing the “Delete Channel” button, their position entry is removed from the Position table. The other values remain untouched and the channel is still visible for HomeDroid. It just has no longer a virtual place in the house.
The name of the channel can be changed at the top EditText. The name field mustn’t be empty. The Text below shows the parent devices _id and its name. With the two spinners below the room and function can be changed. These fields are filled on start of the activity with the values already assigned to the selected channel.
Below “Edit Values”, all values editable for the chosen channel are listed. On the left side, the name of the value is shown while on the right side the value can be edited. There are three types of values, boolean, integer and float. Each value for a given channel requires a certain type of one of these three. By hitting the “Edit Channel” button the entered value will be checked whether it has the correct type.
If everything is correct on button activation, the activity is closed and the record in the Value table and the channel record in the Channel table are modified with *modifyValues()* and *modifyChannel()*.

III Quality management

5 Testing

Testing is done on various levels to ensure the functionality of the application.
I want to thank my application tester Franziska Stützl and Klaus Neutze.

5.1 Test criteria

It is vital to employ several person testings for bugs and misspellings the developer will never catch on his own. Different hardware and Android versions need to be tested to ensure compatibility. There are many different devices with different hardware specs and software iterations. For this purpose, three different devices with two different versions of Android were chosen.

- Samsung Galaxy S Advance - Android 2.3.6
- LG Nexus 4 - Android 4.2.2
- Asus Nexus 7 - Android 4.2.2

In addition, the application was installed on a variety of devices, including the Samsung Galaxy Nexus, Sony Xperia S and the Samsung Galaxy S3, running the current version of their Android.

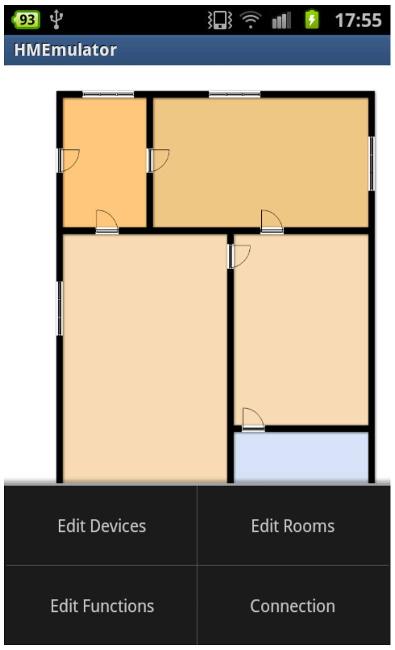


Figure 21. Galaxy S Advance.

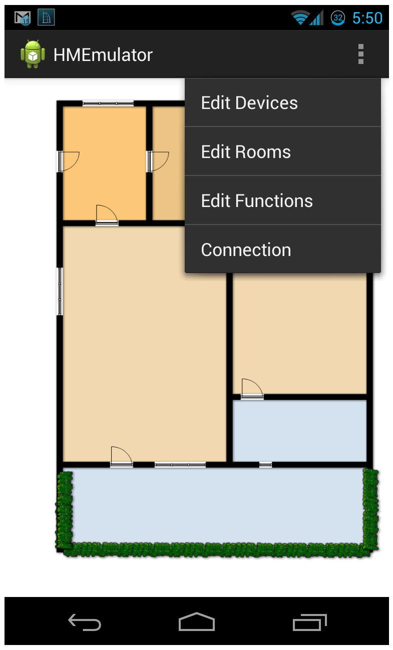


Figure 22. Nexus 4.

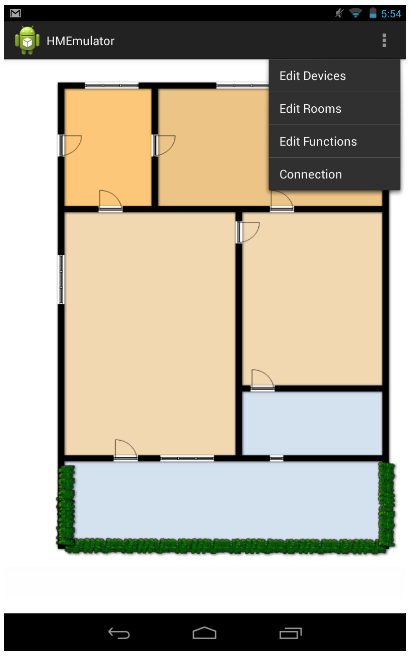


Figure 23. Nexus 7.

5.2 User test results

The specified user tests were done manually on all three devices. These tests were performed on the final version of the software by different users to avoid developer routine.
Test cases were given to the testers. They derived from the System Requirements as follows.

From REQ-1 Stable Android application:

- test: app runs for 10 minutes untouched
- expectation: application does not crash
- passed: working as intended

From REQ-2.1 Data exchange between application and HomeDroid:

- test: connect to the application with a device running HomeDroid
- expectation: HomeDroid finds the server and starts requesting the *.cgi scripts
- passed: start HomeDroid connect to HMEmulator

From the requirements:

REQ-2.1.1 deviceList.cgi retrievable and working
REQ-2.1.2 roomList.cgi retrievable and working
REQ-2.1.3 functionList.cgi retrievable and working
REQ-2.1.4 state.cgi and stateList.cgi retrievable and working
REQ-2.1.6 version.cgi retrievable and working

- test: synchronize HomeDroid client
- expectation: HomeDroid requests the *.cgi scripts above and the HMEmulator replies in a correct way
- passed: HomeDroid can process the requested *.cgi script replied and shows them on its GUI

From REQ-2.1.5 statechange.cgi executable and working

- test: edit a value on HomeDroid
- expectation: value is edited in HomeMatic Emulator and HomeDroid
- passed: Click on the edited channel in the Main Activity and check the values in the “Edit Channel” activity. Re-sync HomeDroid and check the channels values.

From REQ-2.2.1 Creation of relevant entities:

- test: create a device and check if it is properly visible
- expectation: device is created, shown in the list at the bottom of the screen and its channels are visible in the “Assign Channel” activity
- passed: channel appears in the list and the respective spinner in the “Assign Channel” activity

- test: create 20 devices
- expectation: all 20 devices are shown in the list below and its channels are shown the “Assign Channel” activity
- passed: devices appear in the list and the respective spinner in the “Assign Channel” activity

- test: create a room
- expectation: room is created, shown in the list at the bottom of the screen and appears in the room spinner in the “Assign Channel” activity
- passed: room appears in the list and the respective spinner in the “Assign Channel” activity

- test: create a function
- expectation: function is created, shown in the list at the bottom of the screen and appears in the function spinner in the “Assign Channel” activity
- passed: function appears in the list and the respective spinner in the “Assign Channel” activity

From REQ-2.2.2 Persistency of data:

- test: create a device, room and function, close the app, restart it and check objects created before
- expectation: device, room and function are still available as created
- passed: in “Create Channel” all created entities are available

From REQ-2.2.3 Initial load of config database:

- test: check availability of hardware models in the “Edit Device” activity
- expectation: spinner shows available hardware models
- passed: relevant hardware models are available in the “Edit Device” activity

From REQ-2.3.1 Interactive creation of real configuration:

- test: Assign a channel to a position, give a new name, room and function. Then use HomeDroid to resynchronize.
- expectation: changes are shown in the HomeDroid application after resynchronization
- passed: newly assigned channel appears correctly on HomeDroid

From REQ-2.3.1.1 Editable devices:

- test: delete a device by clicking on it in the “Edit Device” activity
- expectation: Device is deleted, including its channels, positions and datapoints. Check this by resynchronizing HomeDroid
- passed: Device, channel and position are no longer shown in their respective activities and on HomeDroid

From REQ-2.3.1.2 Editable rooms:

- test: delete a room not yet assigned to a channel
- expectation: room disappears from room list and is not shown at the “Assign Channel” activity
- passed: room is no longer shown in the list and in the respective spinner in the “Assign Channel” activity

- test: delete a room with assigned channels
- expectation: error toast and no deletion
- passed: error toast appears when trying to delete and the room is not removed from the list and the respective spinner in the “Assign Channel” activity

From REQ-2.3.1.3 Editable functions:

- test: delete a function not yet assigned to a channel
- expectation: function disappears from function list and is not shown at the “Assign Channel” activity
- passed: function is no longer shown in the list and in the respective spinner in the “Assign Channel” activity

- test: delete a function with assigned channels
- expectation: error toast and no deletion
- passed: error toast appears when trying to delete and the function is not removed from the list and the respective spinner in the “Assign Channel” activity

From REQ-2.3.1.4 Editable channels:

- test: delete a channel, including its position in the “Edit Channels” activity
- expectation: points will disappear from the Main Activity and channels can be reassigned in the “Assign Channel” activity
- passed: the channel has no more position dot in the Main Activity and can be used in “Assign Channel”

From REQ-2.3.1.5 Editable values:

- test: click on an assigned channel dot in the Main Activity, edit its values and return to that channel
- expectation: “Edit Channel” opens and the channels values can be edited. On return, the values have the edited value.
- passed: changed values appear when clicking on the channel circle of the just edited channel

- test: changed value is propagated to HomeDroid
- expectation: value is changed on HomeDroid after re-synchronizing
- passed: changed value are shown after re-synchronizing in HomeDroid

From REQ-2.3.2 Visualization of positions and values of channels in the virtual house:

- test: click on the Main Activity background and assign a channel
- expectation: on return a dot is shown at that position
- passed: dot is shown at the selected position

- test: click on an existing point
- expectation: “Edit Channel” activity shows values
- passed: “Edit Channel” activity opens and shows values of the channel associated to the dot selected.

IV Conclusion and future work

6 Conclusion

The goal of this thesis was to develop an emulator for the HomeMatic system. It allows the creation of devices, rooms and functions and assignments of channels to locations in the virtual house are possible. The assigned channels can be manipulated in the Emulator and the HomeDroid client is able to download and change values in the emulated HomeMatic system. Android was chosen as platform for the implementation. The application can easily be installed on any Android device and the touch screen allows intuitive handling of the application.

The thesis includes basic facts, definitions and an overview over the planning and architecture of the application.
The development of the application was divided into four smaller projects. The first project was to implement the NanoHTTPD which provides the web server functions that are needed.
The second part was to re-engineer the XML-API and define the data model.
The third project was to successfully transfer data between the HomeDroid and the HMEmulator.
The final work was to create a GUI and implement functions to create, edit and delete different entities of the system.
Due to the big amount of different devices in the documentation available from HomeMatic and their sometimes incomplete definitions in the documentation, only a sample was selected to be present in the application. [16]

Finally the testing of the application was accomplished.

7 Future work

The application contains all basic functionalities required to emulate the HomeMatic system. HMEmulator could be extended by an alarm system, e.g. due to too high temperature at a thermometer the fire alarm will be activated and the HMEmulator starts blinking. Other events could be defined, like motion sensors causing an alarm as well.

On the design side, for example the current value of a channel's datapoint could be shown right next to the channel point on the Main Activity. This could cause problems with clarity and small screens, so a smart technique has to be found not to overload the screen with information.

Expanding the functionalities for professional use could be another future project. Instead of integrating the predefined database into the application, an interface could be implemented to add new devices from HomeMatic. Via this interface, the direct upload of spreadsheets and conversion into the database could be realized. Thinking far ahead the application could communicate with real sensors and actors via WiFi or could act as visualization of the HomeMatic system.

Appendix

Storage medium

As part of my thesis, I provide a CD with the program source code.

Bibliographic references

[1] Android Developers. android.database.sqlite.
http://developer.android.com/reference/android/database/sqlite/package-summary.html,
accessed 08.04.2013.

[2] Android Developers. Activities.
http://developer.android.com/guide/components/activities.html
accessed 08.04.2013.

[3] Android Developers. Fundamentals.
http://developer.android.com/guide/components/fundamentals.html
accessed 08.04.2013.

[4] Android Developers. Storage Options.
http://developer.android.com/guide/topics/data/data-storage.html
accessed 08.04.2013.

[5] Android Developers. Spinner.
http://developer.android.com/guide/topics/ui/controls/spinner.html
accessed 08.04.2013.

[6] Android Developers. Toasts.
http://developer.android.com/guide/topics/ui/notifiers/toasts.html
accessed 08.04.2013.

[7] Android Developers. Android Developers.
http://developer.android.com/index.html
accessed 08.04.2013.

[8] Android Developers. EditText.
http://developer.android.com/reference/android/widget/EditText.html
accessed 08.04.2013.

[9] Android Developers. Android SDK.
http://developer.android.com/sdk/index.html
accessed 08.04.2013.

[10] Android Developers. Android Debug Bridge.
http://developer.android.com/tools/help/adb.html
accessed 08.04.2013.

[11] Android Developers. ADT Plugin.
http://developer.android.com/tools/sdk/eclipse-adt.html
accessed 08.04.2013.

[12] Android Developers. SQLiteOpenHelper.
http://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html
accessed 08.04.2013.

[13] dirch, Falk Werner, Maik Breternitz, Philipp Ebert. XML-API v1.2.
http://www.HomeMatic-inside.de/software/addons/item/xmlapi
accessed 08.04.2013.

[14] Eclipse. The Eclipse Foundation.
http://www.eclipse.org/
accessed 08.04.2013.

[15] eq-3. Objektmodell.
http://www.eq-3.de/Downloads/PDFs/Dokumentation_und_Tutorials/HM_Script_Teil_2_Objektm
odell_V1.2.pdf
accessed 08.04.2013.

[16] eq-3. Datenpunkte.
http://www.eq-3.de/Downloads/PDFs/Dokumentation_und_Tutorials/HM_Script_Teil_4_Datenpu
nkte_1_503.pdf
accessed 08.04.2013.

[17] eq-3. Partner finden.
http://www.eq-3.de/partner-finden.html
accessed 08.04.2013.

[18] hobbyquaker. XML-API.
https://github.com/hobbyquaker/XML-API
accessed 08.04.2013.

[19] HomeMatic. HomeMatic.
http://www.HomeMatic.com/
accessed 08.04.2013.

[20] HomeMatic Wiki. HomeMatic Script.
http://www.HomeMatic-wiki.info/mw/index.php/HomeMatic_Script
accessed 08.04.2013.

[21] HomeMatic Wiki. Was ist HomeMatic?.
http://www.HomeMatic-wiki.info/mw/index.php/Was_ist_HomeMatic%3F
accessed 08.04.2013.

[22] HomeMatic-forum.de. Erweiterung der XML-API - Version 1.2 ?.
http://HomeMatic-forum.de/forum/viewtopic.php?f=26&t=10098
accessed 08.04.2013.

[23] HomeMatic-forum.de. HMCompanion - Schnittstelle zur CCU.
http://HomeMatic-forum.de/forum/viewtopic.php?f=26&t=4639#p28188
accessed 08.04.2013.

[24] HomeMatic-forum.de. XML-RPC Interface-Beschreibung.
http://HomeMatic-forum.de/forum/viewtopic.php?t=4827
accessed 08.04.2013.

[25] Jarno Elonen. NanoHTTPD.
http://elonen.iki.fi/code/nanohttpd/
accessed 08.04.2013.

[26] Juan-Manuel Fluxà. Using your own SQLite database in Android applications, 03.03.2009.
http://www.reigndesign.com/blog/using-your-own-sqlite-database-in-android-applications/
accessed 08.04.2013.

[27] Lard Vogel, Android Intents - Tutorial, 28.01.2013.
http://www.vogella.com/articles/AndroidIntent/article.html
accessed 09.04.2013.

[28] Lars Vogel. Android SQLite Database and ContentProvider - Tutorial, 16.01.2013.
http://www.vogella.com/articles/AndroidSQLite/article.html
accessed 08.04.2013.

[29] Open Handset Alliance. Open Handset Alliance.
http://www.openhandsetalliance.com/
accessed 08.04.2013.

[30] Osmar R. Zaïane. Database Systems and Structures.
http://www.cs.sfu.ca/CourseCentral/354/zaiane/material/notes/contents.html
accessed 08.04.2013.

[31] Philipp Ebert. HomeDroid.
http://www.HomeDroid.de/
accessed 08.04.2013.

[32] SQLite. SQLite.
http://www.sqlite.org/
accessed 08.04.2013.

[33] w3.org. Hypertext Transfer Protocol.
http://www.w3.org/Protocols/rfc2616/rfc2616.html
accessed 08.04.2013.

[34] Womack, Brian. "Google Says 700,000 Applications Available for Android."
BloombergBusinessweek. Bloomberg, 29.10.2012.
http://www.businessweek.com/news/2012-10-29/google-says-700-000-applications-available-for-android-devices
accessed 08.04.2013.

List of figures

List of tables

Software versions used

Android SDK Bundle Version 20130219
http://developer.android.com/sdk/index.html#download
Eclipse v3.8.0
http://www.eclipse.org/
NanoHTTPD v1.25
http://elonen.iki.fi/code/NanoHTTPD/
SQLite v3.7.16.1
http://www.sqlite.org/
XML-API v1.2 installed on HomeMatic hardware
http://www.HomeMatic-inside.de/software/addons/item/XMLapi