Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Frank Roitzsch

Entwicklung eines Android-basierten Toolkits zur Steuerung aktiver Komponenten eines Arduino-XBee-Netzwerks

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Frank Roitzsch

Entwicklung eines Android-basierten Toolkits zur Steuerung aktiver Komponenten eines Arduino-XBee-Netzwerks

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Birgit Wendholt
Zweitgutachter : Prof. Dr.-Ing. Martin Hübner

Abgegeben am 21.06.2016

Frank Roitzsch

Thema der Bachelorarbeit

Entwicklung eines Android-basierten Toolkits zur Steuerung
aktiver Komponenten eines Arduino-XBee-Netzwerks

Stichworte

Android-Smartphone, App, Arduino, Mikrocontroller, XBee, Netzwerk, IoT

Kurzzusammenfassung

Diese Arbeit beschäftigt sich mit der Entwicklung eines Systems, welches die Steuerung und das Monitoring von IoT-Geräten an Arduino-Modulen in einem XBee-Netzwerk ermöglicht. Die erstellte Android-App bildet den Fernbedienungs-Client, der über WLAN Zugriff zum Arduino-Server-Modul erhält, welches per Ethernet angebunden ist. Die Arduino-Client-Module werden per Plug-and-play eingebunden.

Frank Roitzsch

Title of the paper

Development of an Android-based toolkit for controlling
active components of an Arduino XBee network

Keywords

Android smartphone, app, Arduino, microcontroller, XBee, network, IoT

Abstract

This paper has a focus on the development of a system to control and monitor IoT devices on Arduino modules within an XBee network. The Android app created functions as a remote control unit that has access to the Arduino server module which in turn is connected via ethernet. The Arduino client modules are integrated via plug-and-play.

Inhaltsverzeichnis

1. Einleitung und Motivation

In Abschnitt 1.1 ‚Motivation' wird verdeutlicht, warum es sinnvoll ist, ein Smartphone zur Steuerung von ‚Internet of Things'-Geräten einzusetzen. Außerdem sind hier mögliche technische Lösungen benannt.
In Abschnitt 1.2 ‚Zielsetzung' wird konkretisiert, was mit dieser Arbeit erreicht werden soll und welche Komponenten dafür verwendet werden.
In Abschnitt 1.3 ‚Gliederung' sind die Inhalte der weiteren Kapitel umrissen.

1.1 Motivation

In der heutigen Zeit ist das Smartphone aus unserem Alltag nahezu nicht mehr wegzudenken. Es findet sich für fast jeden Anwender ein zu seinem Nutzungs-verhalten passendes Gerät. Daher finden zunehmend mehr Menschen den Weg vom „Ur-Handy", dessen Funktion sich noch hauptsächlich auf das Telefonieren beschränkte, zu den modernen, individuell konfigurierbaren Smartphones und den daraus resultierenden vielen neuen Möglichkeiten, die so ein ‚schlaues' Gerät bietet. Die Hersteller suchen ständig nach weiteren Anwendungs-bereichen für Smart-phones.
Zurzeit wird die Haussteuerung diverser elektrischer/elektronischer Geräte und Komponenten über das Smartphone weiter ausgebaut. Die Haussteuerungen sollen nicht nur zu Hause im Heim-(W)LAN bedienbar sein, sondern auch erweitert über das Internet. Das ‚Internet of Things', welches diese Möglichkeit beschreibt, ist ein Begriff, der sich zunehmend verbreitet. Viele Firmen bieten mittlerweile Lösungen für verschiedene Bereiche, wie z.B. Licht oder Heizung, an. Dabei kommen Systeme zum Einsatz, die sich von Hersteller zu Hersteller deutlich unterscheiden und untereinander nicht kompatibel sind. Für den Anwender, der interessiert daran ist, unterschiedliche Bereiche fernzusteuern, kann dies aktuell bedeuten, sich mehrere Systeme zulegen zu müssen und für jedes dieser getrennt arbeitenden Geräte ein weiteres Steuerungsgerät oder verschiedene Apps auf seinem Smartphone installieren zu müssen. Diese Herangehensweise kann schnell kostenintensiv und in der Anwendung umständlich werden. Ein System, welches nicht beschränkt ist auf einen Anwendungsbereich, wäre attraktiver für den potenziellen Kunden und könnte die Kaufentscheidung erleichtern. Die heutigen Geräte bieten neben dem

obligatorischen Telefonnetzzugriff meist Bluetooth und als Standard WLAN. Weiterhin ist der große (und derzeit immer größer werdende), hochauflösende Bildschirm mit Touch-Funktion ideal dafür geeignet, variable Bedienungsfelder darzustellen. Somit bietet es sich an, das Smartphone als Fernbedienung zu verwenden, die durch WLAN in jedem Bereich des Hauses nutzbar wird. Der interessierte Anwender des Haussteuerungssystems muss somit keine separate Fernbedienung erwerben und kann mit dem ihm vertrauten Smartphone die Bedienungsseite abdecken. Im Vergleich zu einer zusätzlichen Fernbedienung ist das Smartphone durch die vielseitige Nutzung in unterschiedlichsten Bereichen stets greifbar und einsatzbereit. Eine Smartphone-App stellt somit eine naheliegende Wahl als moderne Bedienungs-Lösung für ein solches System dar.

Im Elektronik-Amateurbereich, aber auch zunehmend in wissenschaftlichen Abteilungen, haben sich in den letzten Jahren sogenannte Prototyping-Platforms verbreitet. Dabei handelt es sich in der Regel um ein Mikrocontroller-Board in unterschiedlicher Hardware-Ausführung und Größe. Im Vergleich zur Verwendung eines separaten Mikrocontrollers bieten solche Boards die benötigte Zusatz-Hardware, um sofort einsetzbar zu sein. Das Programm für den Mikrocontroller kann über einen PC per USB-Schnittstelle problemlos übertragen werden. Dies erleichtert die Benutzung und führt schneller zum Ziel. Die Grundfunktionen sind durch zusätzliche Hardware in Form von Erweiterungs-Boards ausbaubar. Dazu gehören z.B. weitere Schnittstellen, die eine Ethernet-Anbindung erlauben oder eine Funk-Verbindung (WLAN oder andere Funk-Standards) ermöglichen. Solche Boards bieten sich somit als Endgeräte an, auf die der Anwender mit dem Smartphone Zugriff erhält. Für die gewünschten Aufgaben besitzen diese Boards ausreichend Leistung und sind durch zahlreiche frei zu definierende und leicht zugängliche Ein- und Ausgänge vielseitig einsetzbar. Die Anbindung der Boards an einen Router macht den Zugriff aus dem Internet möglich, was auch über das Smartphone in einer eigenständigen App oder Internet-Seite per Browser-App denkbar ist. Es kann ein Funk-Netzwerk aus mehreren dieser Boards, denen der Anwender auf einfache Weise unterschiedliche Funktionen zuweisen kann, aufgebaut werden. Jedes dieser Endgeräte ist dann über das Smartphone steuerbar. Als Funkverbindung ist wie für das Smartphone WLAN denkbar, als kostengünstigere und stromsparendere Alternative können andere Lösungen gewählt werden, wie z.B. der etablierte Funk-Standard ZigBee. Aus diesen Komponenten lässt sich ein vielseitiges, erweiterbares und vom Anwender sehr individuell nutzbares System aufbauen, welches gerade im Amateur-Bereich eine attraktive Alternative zu bestehenden in sich geschlossenen Haussteuerungssystemen darstellt.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, die Grundlage der Entwicklung einer mobilen Smartphone-Fernbedienung zur Steuerung von ‚Internet of Things'-Geräten im Haushalt zu schaffen. Dazu werden einfache Prototyping-Platforms genutzt und ein Plug-and-play-Mechanismus für Client-Module, die die Endgerätesteuerung übernehmen, wird implementiert. Hierfür wird ein System entworfen und entwickelt, welches die Steuerung von Arduino-Modulen ermöglicht, die sich in einem XBee-Funk-Netzwerk befinden. Solch ein System soll sich als flexibles und anpassbares Haussteuerungssystem verwenden lassen. Es soll eine Smartphone-App für das Betriebssystem Android entwickelt werden, die beliebig im Haus verteilte Arduino-Boards als Client-Module ansteuern kann. Diese App soll die Fernbedienungsfunktion übernehmen und dabei auch einen Monitoring-Dienst für alle Client-Module erfüllen. Einem Arduino-Board soll die Rolle eines Server-Moduls (Hauptmodul) zugeteilt werden, welches über Ethernet an einen Router angeschlossen ist, zu dem das Android-Smartphone Kontakt über WLAN herstellen soll. Die App soll die Rolle des Android-Clients einnehmen, dessen Requests über das Server-Modul an das gewünschte Client-Modul verteilt werden. Diese soll das Server-Modul über einen XBee-Funk Broadcast an alle Client-Module versenden. Der Ziel-Client soll per XBee-Funk eine Response zurück an das Server-Modul senden, welches diese an den Android-Client über Ethernet weiterleitet. Die Android-App soll die ankommenden Daten der Response in der grafischen Oberfläche darstellen. Das Server-Modul soll neben den Serverfunktionalitäten (Routing) auch die Client-Funktionalitäten der Client-Module erfüllen. Für die Implementierung der App soll das Betriebssystem Android verwendet werden, da dieses auf einem Großteil der Smartphones mit unterschiedlichsten Hardwarespezifikationen eingesetzt ist. Als Prototyping-Platform sollen die weit verbreiteten Arduino-Boards verwendet werden. Für die Arduino-Boards werden Ethernet-Shields angeboten, die für die Datenübertragung von Smartphone zu Arduino-Board eingesetzt werden können. Für die Verbindung von mehreren Arduino-Boards untereinander bieten sich XBee-Funk-Module der Firma ‚Digi International' an, da sie eine hohe Reichweite bei geringem Energie- und Hardwareaufwand aufweisen und durch XBee-Shields von Arduino unterstützt werden. Die Verbindung von Arduino-Board und XBee-Modul wird durch die XBee-Shields ermöglicht. Durch Nutzung der seriellen Schnittstelle des Mikrocontrollers eines Arduino-Boards werden die Daten an das XBee-Modul weitergereicht, welche dieses per Funk an andere XBee-Module senden kann. Die serielle Schnittstelle wird als Standard von der Arduino-Software unterstützt. XBee-Module können nach

entsprechender Konfiguration ein Netzwerk aufbauen, welches Broadcast und point-to-point-Übertragung ermöglicht.

1.3 Gliederung der Arbeit

Diese Arbeit ist in drei Abschnitte unterteilt:

Kapitel 2 behandelt ‚Entwurf und Realisierung'.
In 2.1 ‚Entwurf' ist zunächst der Grundaufbau des Systems erläutert, daraufhin in 2.2 ‚Realisierung' die verwendete Hardware aufgeführt und die entwickelte Software erklärt. In 2.3 ‚Zusammenfassung' sind die einzelnen Entwicklungsschritte abschließend komprimiert.

Kapitel 3 beschäftigt sich mit ‚Ergebnis und Ausblick'.
In 3.1 ‚Ergebnis und kritische Bewertung' wird zu den Ergebnissen Stellung genommen unter Rücksichtnahme auf sowohl positive als auch zu verbessernde Lösungen. In 3.2 ‚Weiterführung' sind Vorschläge für Erweiterungen des Systems aufgeführt, die zukünftige Einsatzgebiete betreffen.

2. Entwurf und Realisierung

Die Anforderung an diese Arbeit ist, ein System zur Steuerung von verteilten Arduino-Client-Modulen vom Smartphone aus möglich zu machen. Auf dem Android-Smartphone läuft eine Client-App, die über das Arduino-Server-Modul Verbindung zu allen Client-Modulen erhält. Die Verbindung von Smartphone zu Server wird über WLAN/Ethernet aufgebaut, der Server erhält per XBee-Funk Zugriff auf alle Client-Module.
Im Abschnitt 2.1 ‚Entwurf' wird zunächst auf die Verbindungsverwaltung zwischen den einzelnen System-Komponenten eingegangen, sowie anschließend deren Applikationslogik beschrieben.
Im zweiten Teil 2.2 ‚Realisierung' werden zunächst die verwendeten Hardware-Komponenten aufgelistet. Im Software-Abschnitt werden die Arduino-Programme für den Server und die Clients beschrieben. Außerdem wird sich hier mit der nötigen Einrichtung der XBee-Modul-Firmware befasst, durch welche die Funk-Module ein Netzwerk aufbauen können. Anschließend wird auf den Aufbau und die Funktion der Android-App eingegangen. Die verwendeten Übertragungsprotokolle und die Struktur der verwendeten Datenformate werden im letzten Abschnitt erläutert.

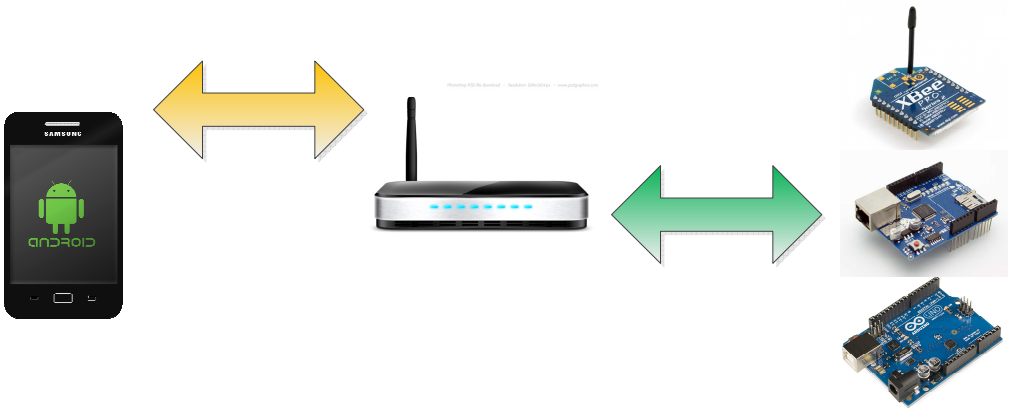
2.1 Entwurf

Verbindungsverwaltung

Das System besteht aus einem Arduino-Server-Modul, welches die Kommunikation zwischen dem Android-Client und allen weiteren Arduino-Clients verwaltet. Der Server befindet sich über eine Ethernet-Schnittstelle im gleichen Netzwerk wie das Smartphone, welches per WLAN Zugang hat. Eine zusätzliche Verbindung baut das Server-Modul zu allen Client-Modulen über ein XBee-Funk-Netzwerk auf. Wenn der Android-Client einen Request an das Server-Modul sendet, wird dieser per Broadcast an die Client-Module weitergeleitet. Die daraufhin ankommende Response des Ziel-Client-Moduls wird vom Server-Modul direkt wieder zurück an den Android-Client geschickt. Für den Request kann als Ziel-Modul auch der Server angegeben sein. Wie die Client-Module enthält der Server auch Client-Funktionalitäten. Der Android-

Client wartet nach dem Verschicken eines Requests auf die passende Response vom Server-Modul, die entweder vom Server-Modul weitergeleitet oder direkt bei diesem erstellt wurde.

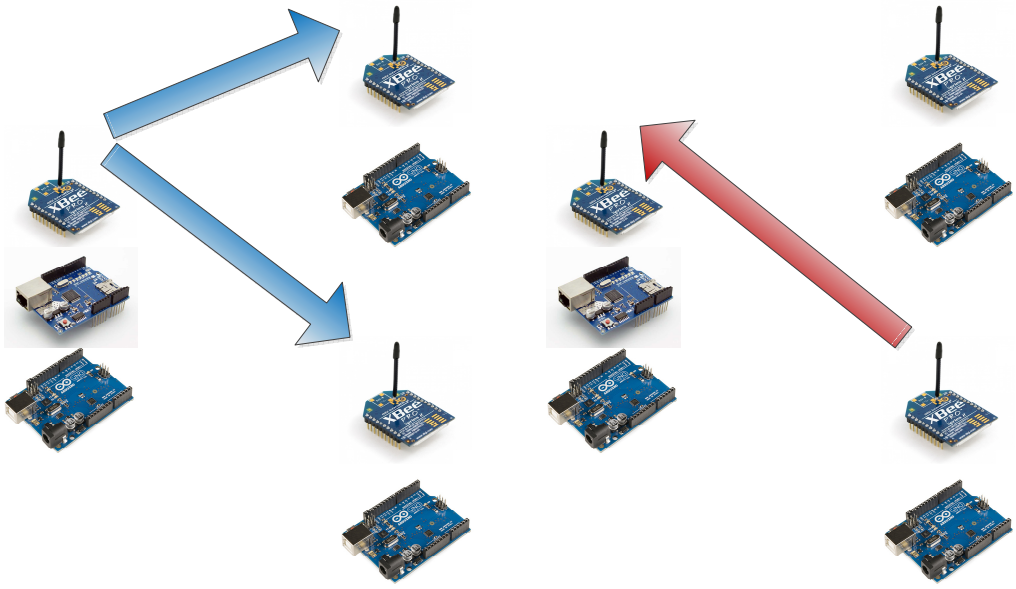
Abb. 1: Kommunikation zwischen Android-Client und Arduino-Server-Modul



Quelle: [1]

Die Client-Module empfangen die weitergeleiteten Requests vom Server-Modul, führen die angeforderte Funktion aus und schicken eine entsprechende Response an das Server-Modul zurück.

Abb. 2: Kommunikation zwischen den Arduino-Modulen



Quelle: [2]

Applikationslogik beim Server-Modul

Die Kommunikation zwischen Arduino-Server-Modul und Android-Client wird über TCP realisiert. Der Server wartet permanent auf einen möglichen Verbindungsaufbau des Android-Clients. Nach Annahme der Verbindung wird auf ankommende TCP-Requests gewartet, die beim Eintreffen direkt verarbeitet werden können. Es gibt zwei Arten der Verarbeitung:
1. Das Ziel-Modul des Requests ist das Server-Modul: Der Server führt die angeforderte Server- oder Client-Funktionalität (siehe Applikationslogik beim Client-Modul) aus und schickt per TCP eine entsprechende Response zurück an den Android-Client.
2. Das Ziel-Modul des Requests ist ein Client-Modul: Der Server leitet den Request über die serielle Schnittstelle an das XBee-Modul, welches die Daten per Broadcast-Funk an alle Client-XBee-Module sendet.
Im Programm des Server-Moduls wird neben der Verwaltung der TCP-Verbindung die serielle Schnittstelle angesprochen, durch welche die XBee-Kommunikation zu den Client-Modulen möglich ist. Neben dem Weiterleiten als Broadcast wird auf einkommende Responses von Client-Modulen an der seriellen Schnittstelle gewartet. Beim Eintreffen von Responses werden die Daten nach dem Einlesen per TCP an den Android-Client weitergeschickt.
Zusätzlich prüft das Server-Modul an der seriellen Schnittstelle, ob ein Login-Request eines Client-Moduls eintrifft. Der Server kann diese Anmelde-Anfrage per Login-Response bestätigen.

Applikationslogik beim Client-Modul

Beim Starten eines Client-Moduls wird periodisch ein Login-Request an das Server-Modul über die serielle Schnittstelle mit XBee-Funk geschickt und auf eine Bestätigungs-Response gewartet, was den Client-Modulen ein Plug-and-play-Verhalten ermöglicht.
Die Bestätigungs-Response führt zur Aktivierung der folgenden Client-Funktionalität, die auch das Server-Modul besitzt:
Ein Client-Modul wartet über die serielle Schnittstelle (XBee) auf ankommende Requests vom Server-Modul. Wenn die mitgesendete Ziel-ID zum Client-Modul passt, wird die Anforderung bearbeitet, welche aus dem Setzen oder Auslesen von Modul-Pins bestehen kann und eine entsprechende Response zusammengesetzt, die

daraufhin über die serielle Schnittstelle wieder an das Server-Modul gesendet wird. Wenn die Ziel-ID nicht übereinstimmt, wird der Request verworfen.

Applikationslogik beim Android-Client

Über die Android-Client-App kann eine TCP-Verbindung zum Server-Modul aufgebaut und auch beendet werden. Nach Anmeldung wird ein Request an das Server-Modul verschickt, der alle Label-Daten des Servers und daran angemeldeten Client-Module enthält. Über verschiedene Requests können Client-Funktionalitäten von jedem Arduino-Modul, welches in der App anwählbar ist, abgerufen werden. Dazu gehört es, digitale Ausgänge zu steuern sowie digitale und analoge Eingänge auszulesen. Weiterhin können Server-Funktionalitäten vom Server-Modul angefordert werden. Der Android-Client wartet in einem Listener-Thread auf ankommende TCP-Responses vom Server-Modul, welche dann beim Eintreffen ausgewertet und entsprechend dargestellt werden.
Die Android-App übernimmt somit eine Art Monitoring-Funktion der Arduino-Module.

2.2 Realisierung

Die Realisierung ist in zwei Teilbereiche unterteilt, in 2.2.1 ‚Hardware-Komponenten' und 2.2.2 ‚Software-Komponenten'. Zunächst wird detailliert die Hardware behandelt, welche die Basis für die Software im darauffolgenden Teil bildet.

2.2.1 Hardware-Komponenten

Nachfolgend sind alle nötigen Hardware-Komponenten aufgelistet und beschrieben, die für die vorliegende Arbeit verwendet wurden. Auf Besonderheiten, die es zu beachten galt, wird dabei eingegangen.

2.2.1.1 Android-Smartphone

Für das Entwickeln der Android-App wurde ein Samsung Galaxy S4 (GT-I9505) verwendet, auf dem die letzte offizielle für das Gerät erschienene Android-Version (Lollipop 5.0.1) installiert ist. Es hat ein 5 Zoll OLED-Display mit einer Auflösung von 1920 x 1080 px, was zu einer Pixeldichte von 441 ppi führt. Damit bietet es eine gute Voraussetzung, viele Informationen kompakt auf einem Bild der App darzustellen. Die weitere Hardware-Leistung ist hier zu vernachlässigen, da die entwickelte App wenig Anforderung stellt. Eine WLAN-Funktion kann als Standard bei einem Android-Smartphone angenommen werden, zumal diese Fähigkeit ein integraler Bestandteil eines heutigen Smartphones ist.

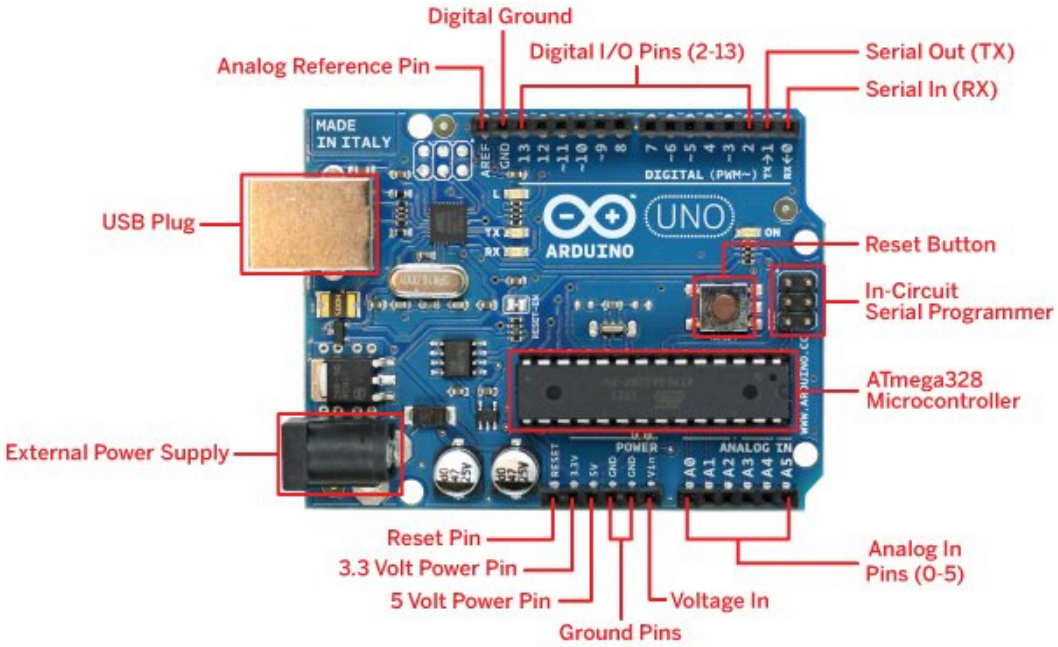
2.2.1.2 Arduino-Boards

Bei einem Arduino (oder auch seit März 2015 ‚Genuino' für den europäischen Markt) handelt es sich um ein Entwicklungsboard mit einem AVR Mikrocontroller des US-amerikanischen Herstellers Atmel. Arduino selbst ist eine italienische Firma, die seit 2005 nicht nur die Hardware-Plattformen entwickelt und herstellt, sondern auch die Software, wie auf den Mikrocontroller vorinstallierte Bootloader (Firmware, die die Mikrocontroller-Konfiguration enthält und das Installieren und Starten eines Programms bereitstellt und steuert), eine Basis-Softwarebibliothek und die Entwicklungsumgebung Arduino IDE.
Sowohl die Hardware als auch die Software unterliegen den Open Source Richtlinien, können also vollständig eingesehen, verwendet und abgeändert werden. Genau genommen beschreibt der Name Arduino also nicht nur die verschiedenen Boards, die mittlerweile entwickelt und verkauft werden, sondern das gesamte Entwicklungssystem aus der Software, welches gezielt auf die zugehörige Hardware abgestimmt ist und auf einfachem Wege zusammenarbeitet. In der Regel besteht ein Arduino-Board aus dem genannten 8bit-AVR Mikrocontroller, meist aus der ATmega-Familie, diversen leicht zugänglichen Ein- und Ausgängen, digital und analog, einem Serial-USB Wandler, der die Kommunikation zwischen Mikrocontroller und PC ermöglicht und einem Spannungsregler-Schaltkreis, welcher den Betrieb in einem erweiterten Spannungsbereich auch unabhängig des USB-Ports möglich macht. Alle Pins des Mikrocontrollers, bis auf einige Ausnahmen durch Platzmangel oder gezielter Einschränkung, werden über sogenannte Header (Pin-Leisten - male) am unteren und oberen Rand herausgeführt, welche als direkte Anschlussmöglichkeiten

für Steckkabel dienen, die für Breadboards genutzt werden oder als Adapter-Port für aufsetzbare Arduino-Shields. Die Betriebsspannung der Mikrocontroller beträgt 5V, welche entweder vom USB-Port abgegriffen oder aus dem separaten Netzteilanschluss über Spannungsregler aufbereitet wird. Diese Spannung wird auch über die Header zur weiteren Nutzung herausgeführt, außerdem 3,3V, womit unter anderem auch neuere Mikrocontroller arbeiten, eine Referenzspannung (IOREF), ein Vin-Pin, mehrere GND-Pins und der Reset-Pin. Auf der Platine befindet sich außerdem ein Quarz, wie auf allen Arduino 8MHz oder 16MHz, der standardmäßig als externer Taktgeber verwendet und auch im Bootloader von Arduino als Taktquelle vorkonfiguriert ist. Durch diverse Bauteile, die die Handhabung vereinfachen, wie ein Reset-Taster, eine frei nutzbare LED und weitere für eine optische Anzeige der seriellen Kommunikation und Komponenten, die eine Benutzung gegen Schäden absichert, wie Dioden, Kondensatoren und rückstellbare Sicherungen, ist ein Arduino-Modul ein Komplettpaket, welches leicht für Prototyping-Projekte einsetzbar ist. Der Anwender muss sich über die Hardware-Komponenten, die zum Grundbetrieb eines Mikrocontrollers nötig sind, keine Gedanken machen. Die Funktion ist ‚out-of-the-box' gegeben und die USB-Verbindung zum PC macht einfache Programm- und Datenübertragung möglich, neben der gleichzeitigen Spannungsversorgung über diese Schnittstelle. Somit ist es eine ideale Plattform, um schnell ohne besondere Hardware-Kenntnisse ein Projekt zu erstellen, welche für Steueraufgaben oder Sensorerfassung Verwendung findet, um nur zwei Beispiele zu nennen.
Für das in dieser Arbeit vorgesehene Einsatzgebiet, bei dem Server und Client als separate Module fungieren, eignet sich das Arduino-Board unter anderem deswegen, weil auf die Arduino-Shields (Platinen-Steckaufsätze für das Arduino-Hauptboard) zurückgegriffen werden kann, die die Grundfunktion wie gewünscht erweitern können. Auf die verwendeten Shields, die nötig für die Kommunikation zwischen Smartphone und Boards (im Folgenden auch Module genannt) untereinander sind, wird im nächsten Abschnitt eingegangen. Die Software wurde zunächst komplett auf Basis des Arduino UNO entwickelt, welches als ‚Einsteigerboard' gilt. Es handelt sich nicht um ein ‚Einsteigerboard', weil es besonders leicht für den Einstieg in die Arduino-Welt ist, sondern, weil es in seiner Leistungsfähigkeit und Eigenschaften im Mittelbereich einzuordnen ist. Es bietet Funktionen, die für einen Großteil der Anwender ausreichen und auch Reserven für etwas größere Projekte sind durch die Rechenleistung und umfangreichen integrierten Programm- und Datenspeicher (im Vergleich zu anderen 8bit-Mikrocontrollern) gegeben. Außerdem gibt es für dieses Board die meisten kompatiblen Shields, da dieses auch die verbreitetste Verwendung findet. Das UNO-Board gibt es mittlerweile in der Hardware-Revision 3, UNO R3, welches verschiedene kleinere Änderungen im Hardware-Design, wie Anzahl und

Platzierung der Bauteile und auch den Austausch des USB-Serial Chips erhalten hat. Für den Anwender sind das unerhebliche Änderungen und ein für ein UNO-Board vorgesehenes Programm läuft auf allen Board-Revisionen ohne Einschränkungen, etwaige Hardware-Änderungen werden in der Arduino-(Treiber)-Software gekapselt. Zu Anfang wird der Anwender kaum an die Hardwaregrenzen stoßen, es sei denn er möchte viele umfangreiche Bibliotheken verwenden, die schnell den benötigten Programmspeicher ansteigen lassen und auch viel von dem SRAM beanspruchen. Für Anwendungsfälle in denen der Arduino UNO zu knapp wird, kann ein erweitertes Modell in Erwägung gezogen werden, wie der ‚große Bruder' des Arduino UNO, der Arduino MEGA. Es handelt sich hier um ein größeres Board mit einem Mikrocontroller welcher deutlich mehr Ein- und Ausgängen bietet sowie mit zahlreichen weiteren integrierten Hardware-Funktionen und einem stark erweiterten Programm- und Datenspeicher, der auch für aufwendige Projekte ausreichend sein kann. Bei diesen Boards ist die Kompatibilität nicht zu allen Arduino-Shields ohne Weiteres gegeben. Zu Testzwecken und um auch die Möglichkeiten aufzuzeigen, wurde für diese Arbeit ein Arduino MEGA als Client zusätzlich zu den Arduino UNOs eingesetzt. Eine Nutzung als Server ist ebenso denkbar und durch die weiteren Ein- und Ausgänge auch als durchaus sinnvoll anzusehen. Für die derzeit implementierten Funktionen beim Server und bei den Clients reicht der Speicher und die Leistungsfähigkeit eines Arduino UNO aus, mit Toleranz für Erweiterungen. Durch die größtenteils plattformunabhängige Programmiersprache von Arduino, bei der es sich um ein erweitertes C handelt, welches Bibliotheken mit Funktionen beinhaltet, die gezielt auf die Hardware abgestimmt ist, kann der Quellcode in fast allen Bereichen unverändert bleiben und nur die Konfigurationsabschnitte sind anzupassen. Programme werden zuverlässig und schnell über die USB-Schnittstelle aus der Arduino-IDE übertragen.

Abb. 3: Arduino UNO mit Komponenten-Beschriftung



Quelle: [3]

Tab. 1: Vergleich Arduino UNO / MEGA

Arduino-Board	UNO	MEGA / MEGA 2560
Mikrocontroller	ATmega 328p	ATmega 1280 / 2560
digitale Ein- und Ausgänge	16	54
PWM	6	15
analoge Eingänge	6	16
Flash-Speicher (kB)	32	128 / 256
SRAM (kB)	2	8
EEPROM (kB)	1	4
UART	1	4

Quelle: [4]

2.2.1.3 Ethernet-Shield

Beim Ethernet-Shield handelt es sich um eine Hardware-Erweiterung für ein Arduino-Board. Das Shield erweitert ein Arduino-Board durch den enthaltenen Ethernet Controller (Wiznet W5100) und dem RJ45 Port um eine Ethernet-Schnittstelle. Es bietet Verbindungsgeschwindigkeiten von 10/100Mbit/s und kann für TCP und UDP verwendet werden. Die Spannungsversorgung erhält es direkt über das Arduino-Board. Das Shield wird auf das Arduino-UNO (oder MEGA, bei dem aber Hardware-Anpassungen durch Pin-Inkompatibilitäten nötig sind) gesetzt. Alle Pins werden durchgeschleift, so dass weitere Shields, wie das XBee-Shield (in folgenden Abschnitten behandelt) aufgesetzt werden können. Die auf dem Arduino-Board enthaltene ICSP-Schnittstelle, welche das XBee-Modul zur Spannungsversorgung nutzt, wird nicht durchgeschleift. In der Regel ist das Erweitern problemlos und erfordert laut Arduino selten, wie in diesem Fall (siehe Abschnitt ‚XBee-Shield') oder auch beim Arduino MEGA, Anpassungen. Das Ethernet-Shield wird über das Arduino-Board angesteuert, für das leistungsfähige Bibliotheksfunktionen bereitstehen und ermöglicht es dem Mikrocontroller, welcher als Server fungiert, Daten vom Smartphone über TCP zu empfangen und zu senden.

Abb. 4: Ethernet-Shield

Quelle: [5]

2.2.1.4 XBee-Module

Um eine Funkübertragung zwischen Arduino-Server und Arduino-Clients zu ermöglichen, werden XBee-Module der amerikanischen Firma ‚Digi International' eingesetzt. Bei den in der vorliegenden Arbeit verwendeten Modulen handelt es sich

um XBee ZB S2 (Series 2), auf denen eine ZigBee mesh Firmware läuft, die ein zu Series 1 nicht kompatibles Netzwerkprotokoll bereitstellt. Wie die Module der Series 1 können auch die Series 2 Module im einfachen point-to-point und star mode konfiguriert und eingesetzt werden. Weiterhin gibt es Module für beide Series in PRO Ausführungen für mehr Reichweite und Datendurchsatz sowie auch Module, die im 900 MHz Frequenzbereich arbeiten. Die ca. 24x27mm großen XBee S2 Module, welche in dieser Arbeit ausschließlich eingesetzt werden, sind die Standard-ausführung, die im 2,4 GHz Frequenzbereich arbeiten, eine RF Datenrate von 250 kbit/s bieten, außerdem eine Reichweite von 120m außerhalb bei Sichtverbindung und 40m in Gebäuden oder im städtischen Bereich. Die Übertragungsreichweite lässt sich verbessern, indem Module in besonderer Ausführung verwendet werden, die eine Drahtantenne oder einen Anschluss für eine externe Antenne bieten. Die hier verwendeten Module besitzen eine PCB-Antenne (integriert auf der XBee-Modul-Platine) die nicht die volle potenzielle Reichweite ausschöpfen kann, aber für den Testaufbau ohne feststellbare Störungen funktionierte. Sie sind dazu gedacht, ein zu bestehenden Funktechnologien wie Bluetooth oder WLAN günstigeres und ein-facheres Netzwerk (Wireless Personal Area Network - WPAN) aufbauen zu können. Für den vorgesehenen Einsatzzweck, wie Steuerung von Ausgängen oder Erfassung von Sensordaten über Eingänge ist die Datenrate ausreichend, im Vergleich zur gleichzeitig eingesetzten Ethernet-Schnittstelle aber um viele Faktoren geringer (10/100 Mbit/s <-> 250kbit/s), so dass die Übertragung zwischen dem Arduino-Server (Hauptmodul) und den Arduino-Clients in der Geschwindigkeit und Leistungsfähigkeit nicht die schnellere Ethernet-Übertragung zwischen Smartphone und Arduino-Server erreicht und somit Einschränkungen in Kauf genommen werden müssen. Zur Anbindung der XBee-Module wird die serielle Schnittstelle (UART) der Arduino-Board Mikrocontroller verwendet.

Abb. 5: XBee-Modul

Quelle: [6]

2.2.1.5 XBee-Shield

Als Adapter für die Verbindung von Arduino UNO/MEGA-Boards zu den XBee Modulen werden XBee-Shields verwendet. Wie auch beim Ethernet-Shield können diese auf das Arduino-Board gesteckt werden. Sie schleifen alle verdeckten Ein- und Ausgänge durch, verwehren aber den Zugang zu den Spannungsversorgungs-Pins, da diese unterhalb des Bereiches sitzen, wo das XBee-Modul auf das Shield gesetzt wird. Das Shield ist laut Arduino kompatibel zu verschiedenen Digi International-Funkmodulen mit gleichem Formfaktor, wie allen XBee/ZigBee-Ausführungen, Bluetooth-Modulen und weiteren angebotenen Funkstandards. Da für die vorliegende Arbeit nur XBee ZB S2 (Series 2) verwendet werden, kann die Funktion mit den anderen Modulen nicht bestätigt werden. Versorgt wird das Modul über den ICSP-Port des Arduino-Boards, von dem sich das Shield die Spannung abgreift. Dies hat zur Folge, dass das Shield nicht ohne Hardware-Anpassung auf das Ethernet-Shield beim Arduino-Server-Modul gesteckt werden kann, da das Ethernet-Shield in der benutzen Hardware-Revision den ICSP-Port nicht weiterleitet, somit also nicht wie beim Arduino-Board eine Ebene höher sitzt. Es gibt mittlerweile neuere XBee-Shields von anderen Herstellern, wie z.B. ‚sparkfun electronics' die die benötigte Versorgungsspannung aus den Spannungs-Headern verwenden und somit nicht auf den ICSP-Port angewiesen sind. Durch einen geringen Löteingriff konnten im vorliegenden Fall die benötigten Pins des ICSP-Ports vom Server-Arduino-Board auf das XBee-Shield geleitet werden. Diese Anpassung müsste auch für die Verwendung des Arduino MEGA als Server vorgenommen werden, was hier nicht gemacht wurde, so dass das Arduino MEGA nur testweise als Server für die Smartphone Client-App ohne Verbindung zu Arduino-Clients zum Einsatz kam. Das umgekehrte Aufsetzen, welches beim Arduino-Shield-Konzept theoretisch möglich sein soll, also zuerst das XBee-Shield und darauf das Ethernet-Shield, ist mit dem zur Verfügung stehenden XBee-Shield nicht möglich, da wie bereits bekannt, die Spannungs-Header, die in diesem Fall vom Ethernet-Shield verwendet werden, Design-bedingt nicht durchgeführt werden können. Bei angesprochenem sparkfun XBee-Modul wäre dies möglich. Die on-top-Anbringung des XBee-Shields erscheint als die ideale Variante, mit welcher das XBee-Modul freisteht und nicht durch die Ethernet-Platine abgedeckt wird, die durch die Elektronik Abschirmung verursachen kann.

Abb. 6: XBee-Shield

Quelle: [7]

2.2.1.6 WLAN-Router

Für die Verbindung vom Smartphone zum Arduino-Server-Modul ist ein WLAN-Router nötig. Alternativ könnte ein Switch oder Hub mit WLAN-Schnittstelle eingesetzt werden, der für die reine Kommunikation innerhalb des Smartphone <-> Arduino-Netzwerkes, welches in der Arbeit behandelt und verwendet wird, ausreichen würde. Durch einen Router mit integriertem Switch wird die Verbindung zum Internet möglich, welcher erst den Zugriff in das Heimnetzwerk von außerhalb bieten kann. Mit der DHCP-Funktion eines Routers kann dem Arduino-Ethernet-Shield optional automatisch eine IP-Adresse zugewiesen werden. Für die Arbeit wurde ein 300Mbit/s WLAN Router (TL-WR841N) der Firma TP-Link eingesetzt. Ein separater Hub/Switch stand nicht zur Verfügung, die Funktion kann daher hier nicht bestätigt werden.

2.2.2 Software-Komponenten

Zu den Software-Komponenten zählen vier Teilbereiche. Die Arduino-Programme (Server und Client), die XBee-Firmware-Konfiguration zur Einrichtung des XBee-Netzwerks, die Android-App als Fernbedienung und die Übertragungs- und Datenprotokolle als Bindeglied zwischen den einzelnen Komponenten. Bei der im folgenden verwendeten Bezeichnung ‚Android-App' ist der Software-Teil des Android-Clients gemeint. Zur Vereinfachung wird im Folgenden die Bezeichnung ‚Server(-Modul)' für das Haupt-Arduino-Board (UNO/MEGA) mit aufgestecktem Ethernet-

Shield und XBee-Shield verwendet, wobei hier das XBee-Shield zur reinen Kommunikation zwischen Android-Client und Server-Modul optional ist. Ein ‚Client-Modul' bezeichnet folgend ein Client-Arduino-Board (UNO/MEGA) mit aufgestecktem XBee-Shield. Die Bezeichnung ‚Arduino-Modul' steht für das Server-Modul oder ein beliebiges Client-Modul.

2.2.2.1 Arduino-Programme

In den folgenden Abschnitten wird die Implementierung der Arduino-Software beschrieben. Zunächst wird auf das Server-Programm eingegangen, anschließend auf das Client-Programm.
Vor der Installation auf den Arduino-Boards müssen die Programme jeweils in einer zugehörigen Header-Datei konfiguriert werden. Nach dem Vorkonfigurieren müssen die Programme vor der Übertragung zunächst neu kompiliert werden. Auf diese Weise sind Grundeinstellungen festgelegt und bei jedem Start eines Moduls vorgegeben.

2.2.2.1.1 Arduino-Entwicklungsumgebung – Arduino DIE

Als Entwicklungsumgebung diente die freie Arduino IDE in Version 1.6.x, mit der die Programme über die USB-Schnittstelle an ein Arduino-Modul übertragen und damit der Mikrocontroller mithilfe des internen Bootloaders geflashed wird. Die Programme werden bei Arduino ‚Sketches' (mit Endung ‚.ino') genannt, daher wird diese Bezeichnung im Folgenden verwendet. Die Arduino-Bibliotheksfunktionen erleichtern den Zugriff auf die Hardware des Mikrocontrollers, indem sie die unterschiedlichen Board-Fähigkeiten in Grundfunktionen kapseln und Ein- und Ausgänge entsprechend durch Board-Dateien mappen. Das hat den Vorteil, dass einmal geschriebene Sketches ohne aufwändige Änderungen, wie es bei direkter Programmierung von Mikrocontrollern in C oder Assembler der Fall wäre, für ein anderes Arduino-kompatibles Board verwendet werden kann, ohne sich in die Datenblätter der Mikrocontroller einzuarbeiten. Weiterhin sind in einem Arduino-Programm grundsätzlich Funktionen enthalten, die in einem Sketch direkt verwenden werden können, wie z.B. die Funktion millis(), die einen Hardware-Timer des Mikrocontrollers für eine Interrupt-gesteuerte Zeitbasis verwendet. Dadurch wird neben dem Platz (Programm- und Datenspeicher), der durch den Bootloader reserviert ist, auch ein Teil für die Arduino-Programm-Basis belegt und bestimmte Peripherie-Bausteine des

Mikrocontrollers automatisch verwendet. Darauf ist bei der Erstellung eines Sketches unbedingt zu achten, falls bereits verwendete Mikrocontroller-Peripherie angesprochen werden soll, was nicht geregelt wird, da Arduino-Programme ausschließlich aus einer Erweiterung der vollständig vorhandenen C-Funktionen bestehen. Der Quellcode wird nicht interpretiert, sondern vor der Übertragung in Maschinensprache kompiliert, welche direkt auf dem Mikrocontroller nach dem Flashen ausgeführt werden kann.

2.2.2.1.2 Server-Modul

Das Server-Modul übernimmt zum einen den Datenaustausch zwischen Android-Client über TCP und zum anderen schickt er Requests vom Android-Client weiter an alle Client-Module und leitet deren Responses zurück an den Android-Client.
In diesem Abschnitt wird auf den Aufbau und die einzelnen Funktionen des Servers eingegangen. Zusatzfunktionen, die nicht die Kommunikation betreffen, werden hier nicht betrachtet. Für das Ethernet-Shield wird die Arduino-Ethernet-Bibliothek verwendet, für die serielle Schnittstelle zur Funkübertragung per XBee werden von Arduino Funktionen bereitgestellt, die bei jedem Sketch direkt verwendet werden können.

Vorkonfiguration des Server-Moduls

Zunächst muss das Server-Programm ‚ArduinoServer(_MEGA).ino' über die Header-Datei ‚ArduinoServer_(MEGA)_config.h' vorkonfiguriert werden:

Mit den folgenden Zeilen wird das Modul-Label definiert:

```
const byte SERVER_MODULE_UNIQUE_ID = 0;
const char SERVER_MODULE_NAME[] = "Server";
```

SERVER_MODULE_UNIQUE_ID: Gibt die Modul-ID des Servers an, die sich eindeutig unterscheiden muss von allen Modul-IDs. Als default ist ‚0' vorgegeben, welche verwendet werden sollte.
SERVER_MODULE_NAME: Eine frei zu wählende Bezeichnung für das Server-Modul.
Im folgenden Wert wird die Größe der Client-Liste festgelegt:

```
const byte MAX_CLIENT_NUMBER = 3;
```

MAX_CLIENT_NUMBER: Die maximale Anzahl an Client-Modulen, die sich beim Server anmelden können. Der Wert kann auch deutlich höher gewählt werden.

Die folgenden Code-Zeilen betreffen die Ethernet-Einstellungen:

```
byte mac[] = {
  0x90, 0xA2, 0xDA, 0x00, 0x07, 0x56
};
// Set the static IP address to use if the DHCP fails to assign
const IPAddress ip(192, 168, 1, 150);
const IPAddress myDns(192, 168, 1, 1);
const IPAddress gateway(192, 168, 1, 1);
const IPAddress subnet(255, 255, 255, 0);
const word SERVER_PORT = 64000;
```

mac[]: Gibt die MAC-Hardware-Adresse für das Ethernet-Shield an. Verwendet wurde, wie von Arduino empfohlen, die auf dem Shield aufgeklebte Adresse.
IPAddress: Hier werden eine feste IP-Adresse und die restlichen Adressen Router-betreffend angegeben.
SERVER_PORT: Der gewählte Port, über den die TCP-Verbindung zur Android-App abläuft.

Die folgenden Code-Zeilen betreffen die Einstellungen der seriellen Schnittstelle:

```
const word BAUD_RATE = 38400;
const word SERIAL_TIMEOUT = 3000;
const unsigned long CLIENT_MODULE_RESPONSE_TIMEOUT = 3000;
```

BAUD_RATE: Gibt die Baud-Rate für die serielle Schnittstelle an und muss mit der des Server-Moduls übereinstimmen.
SERIAL_TIMEOUT: Gibt die maximale Timeout-Zeit beim blockierenden Auslesen des Eingangspuffers in msec an.
CLIENT_MODULE_RESPONSE_TIMEOUT: Gibt die maximale Timeout-Zeit beim optionalen blockierenden Warten auf eine Client-Response in msec an.

In den folgenden Zeilen der Header-Datei werden die Pin-Modi der Server-Modul-Pins konfiguriert:

```
// Pin-Modes
// NOT_DEFINED = 0;
// DIGITAL_OUT = 1;
// DIGITAL_IN = 2;
// DIGITAL_IN_PU = 3;
// ANALOG_OUT_PWM = 4;
// ANALOG_IN = 5;

#define PIN_0_MODE NOT_DEFINED;
…
#define PIN_19_MODE NOT_DEFINED;
```

Nicht jedem Pin kann jeder der sechs genannten Pin-Modi zugewiesen werden (Hinweise in der Header-Datei). Es gibt nur eine begrenzte Anzahl an ausgewiesenen PWM- und analogen Eingangs-Pins. Pins, die als NOT_DEFINED gesetzt werden, gelten als nicht benötigt und werden in der Android-App nicht aufgelistet.

Programmablauf des Server-Moduls

setup()-Funktion:

Die Arduino setup()-Funktion ist für die Initialisierung vorgesehen und wird einmal beim Start des Programms durchgeführt.
Zunächst wird jedem Arduino-Pin ein Pin-Modus zugewiesen. Die serielle Schnittstelle wird mit der gewählten Baud-Rate initialisiert und der Time-Out eingestellt. Außerdem wird die Ethernet-Schnittstelle mit einer festen IP-Adresse eingerichtet. Alternativ kann per DHCP eine Adresse zugewiesen werden (siehe ‚#define DHCP_IP_ADDRESS_ALLOCATION' in der Header-Datei).

loop()-Funktion:

Nach dem Initialisieren folgt die Hauptschleifen-Funktion, bei Arduino ‚loop()' genannt. Ein Arduino-Sketch muss zwingend neben der setup() eine loop()-Funktion

enthalten. Diese stellt die Superloop des Mikrocontrollers dar, die während das Betriebes endlos durchlaufen wird, vergleichbar mit einer while(1)-Schleife, die bei reinem C für Mikrocontroller verwendet werden muss, da das Verhalten sonst nicht definiert ist.
Dauerhaft wird in dieser Loop auf eine TCP-Verbindung des Android-Clients gewartet und wenn die Verbindung aufgebaut ist, ob Daten zum Lesen vorliegen.

Die folgende Funktion gibt dem Server Zugriff auf den Android-Client:

EthernetClient client = server.available();

Diese Nachrichten werden, nach dem Ablegen in den Eingangspuffer, zerlegt und ausgewertet. Entsprechend des Request-Codes wird die Anforderung durchgeführt, eine Response zusammengesetzt und zurückgeschickt. Wenn der Request für ein Client-Modul bestimmt ist, wird er über die serielle Schnittstelle, die die Daten an das XBee-Modul überträgt, weitergeleitet. Zu erkennen ist das für den Server anhand der mitgesendeten Modul-ID.
Außerdem wartet der Server auf einkommende Daten über die serielle Schnittstelle.

Über diese Funktionen wird der Puffer der seriellen Schnittstelle geprüft und beim Vorliegen von Eingangsdaten ausgelesen:

void checkSerialAvailable(EthernetClient client)
void checkSerialLoginAvailable()
void responseOnLoginRequest()

Im Falle einer Response von einem Client-Modul wird die Nachricht direkt an den Android-Client weitergeschickt. Auf einen Anmelde-Request eines Client-Moduls wird seriell geantwortet, welches in der Funktion responseOnLoginRequest() abgehandelt wird.

Der Ausgangspuffer (weitere Erläuterung unter ‚Datenprotokolle') wird mit folgenden Funktionen mit den angeforderten Daten beschrieben:

int printIPAddressToBuffer(char* bufferPointer, IPAddress address)
int printPinDataToBuffer(char* bufferPointer, byte pinNr)

Nur bei Pins, die als Ausgänge definiert sind, wird der Pin-Wert über die Funktion setPinValue() gesetzt:

void setPinValue(byte pinNr, byte value)

Die Funktion getClientListPosition() liefert die nächste freie Position in der Client-Liste des Servers. Die maximale Anzahl verbundener Clients wird durch MAX_CLIENT_NUMBER festgelegt:

int getClientListPosition(byte clientUniqueId)

Die Funktion sendBufferAsSingleChars() versendet über die serielle Schnittstelle den Inhalt des Ausgangspuffers zeichenweise:

void sendBufferAsSingleChars(const char* bufferPointer, int delayMillisBetweenChars, int delayMillisAfterTheLastChar)

2.2.2.1.3 Client-Modul

Im Folgenden wird nur auf Unterschiede zum Programm des Server-Moduls eingegangen, die sich im Sketch ‚ClientModule(_MEGA).ino' für ein Client-Modul zeigen. Server und Client teilen sich einen Großteil der Sketch-Funktionen und Variablen (siehe Abschnitt ‚Server-Modul'), da die ausgetauschten Nachrichten in gleicher Form bei Server-Modul und Client-Modul vorliegen und jedes Arduino-Modul die gleichen Anforderungen (Client-Funktionalität) durchführen kann. Der Hauptunterschied zum Server besteht darin, dass die Client-Module Daten nur über die serielle Schnittstelle per XBee austauschen.

Vorkonfiguration des Client-Moduls

Zunächst muss das Client-Programm ‚ArduinoClient(_MEGA).ino' über die Header-Datei ‚ArduinoClient_(MEGA)_config.h' vorkonfiguriert werden:

Wie beim Server muss eine ID und ein Name für jedes Client-Module gewählt werden:

```
const byte MODULE_UNIQUE_ID = 1;  // ID '0' is reserved for Server and every ID has to be unique
const char MODULE_NAME[] = "ClientModule 1";
```

MODULE_UNIQUE_ID: Gibt die Modul-ID des Client-Moduls an. Jede Modul-ID muss einzigartig sein, sonst können die Module nicht unterschieden werden. Es würde außerdem ein gleichzeitiges Senden von Client-Modulen mit identischer ID zur Folge haben, was zu Fehlfunktion führen würde.
MODULE_NAME: Eine frei zu wählende Bezeichnung für das Client-Modul.

Die folgenden Code-Zeilen betreffen die Einstellungen der seriellen Schnittstelle:

```
const word BAUD_RATE = 38400;
const word SERIAL_TIMEOUT = 3000;
```

BAUD_RATE: Gibt die Baud-Rate für die serielle Schnittstelle an und muss mit der des Server-Moduls übereinstimmen.
SERIAL_TIMEOUT: Gibt die maximale Timeout-Zeit beim blockierenden Auslesen des Eingangspuffers in msec an.

Weitere Einstellungen betreffen den Login-Request beim Starten des Client-Moduls:

```
const unsigned long LOGIN_TIME_INTERVAL_MILLIS = 2000;
const byte CONNECTED_STATE_LED_PIN = 13;  // used to indicate Connected State if defined
```

LOGIN_TIME_INTERVAL_MILLIS: Gibt das Intervall zwischen den versendeten Login-Requests beim Starten des Client-Moduls in msec an.
CONNECTED_STATE_LED_PIN: Wählt den Arduino-Pin aus, über den der Verbindungsstatus zum Server angezeigt wird. Durch Auskommentieren wird die Anzeige deaktiviert und der Pin kann frei anderweitig verwendet werden. An Pin 13 ist eine auf dem Arduino-Board integrierte LED angeschlossen.

In den folgenden Zeilen der Header-Datei werden wie beim Server-Modul die Pin-Modi der Client-Modul-Pins konfiguriert:

```
// Pin-Modes
// NOT_DEFINED = 0;
// DIGITAL_OUT = 1;
// DIGITAL_IN = 2;
// DIGITAL_IN_PU = 3;
// ANALOG_OUT_PWM = 4;
// ANALOG_IN = 5;
#define PIN_0_MODE NOT_DEFINED;
…
#define PIN_19_MODE NOT_DEFINED;
```

Nicht jedem Pin kann jeder der sechs genannten Pin-Modi zugewiesen werden (Hinweise in der Header-Datei). Es gibt nur eine begrenzte Anzahl an ausgewiesenen PWM -und analogen Eingangs-Pins. Pins, die als NOT_DEFINED gesetzt werden, gelten als nicht benötigt und werden in der Android-App nicht aufgelistet.

setup()-Funktion:

Zunächst wird jedem Arduino-Pin ein Pin-Modus zugewiesen. Die serielle Schnittstelle wird mit der gewählten Baud-Rate initialisiert und der Time-Out eingestellt. Noch in der setup() wird periodisch (eingestellt über LOGIN_TIME_INTERVAL_MILLIS) ein Anmelde-Request an den Server geschickt. Das Client-Modul führt dies solange aus, bis eine passende Bestätigung vom Server empfangen wird, erst dann wird in die loop()-Funktion gewechselt. Durch eine LED kann das Wechseln in die loop()-Funktion und damit die Bereitschaft angezeigt werden. (siehe CONNECTED_STATE_LED_PIN in der Header-Datei). Durch diese Funktion wird ein Plug-and-play der Client-Module ermöglicht.

loop()-Funktion:

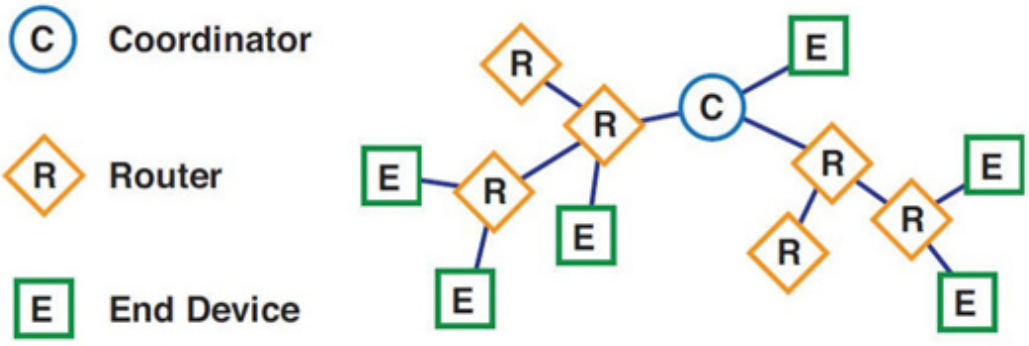
In der loop()-Funktion gilt das Client-Modul als beim Server angemeldet und kann auf alle vom Server weitergeleitet Requests (Client-Funktionalität aktiviert) eingehen. Die Requests werden vom Server per Broadcast an alle angemeldeten Client-Module verschickt. Jedes Client-Modul muss prüfen, ob der Request für ihn bestimmt ist, was

er anhand der ID erkennen kann. Wenn dies nicht der Fall ist, wird die Nachricht verworfen. Das Auswerten eines ID-passenden Requests läuft auf vergleichbare Weise ab, wie beim Server. Die Response wird über die serielle Schnittstelle zum Server gesendet, der das Weiterleiten an den Android-Client durchführt. Wenn ein Client-Modul über die Android-App entfernt wird, kann dieses nach einer Bestätigungs-Response keine Requests mehr entgegennehmen. Erst nach einem Reset des betreffenden Client-Moduls kann es sich wieder am Server anmelden.

2.2.2.2 XBee-Firmware-Konfiguration

Um die Kommunikation zwischen den XBee-Modulen zu ermöglichen, müssen diese zunächst konfiguriert werden. Die Hersteller-Firma Digi International bietet dafür ein kostenfreies Programm-Tool an. Das Programm ‚XTCU' wurde für den vorliegenden Fall in Version 6.3.1 verwendet. Mithilfe eines USB-Serial-Adapters (z. B. auch ein Arduino-Board mit entnommenen Mikroconroller) kann vom Programm auf das XBee-Modul per USB-Port zugegriffen werden.
Ein XBee-Mesh-Netzwerk besteht immer aus einem einzigen ‚Coordinator'-Modul, welches das Netzwerk anlegt. Weitere ‚Router'-Module können diesem Netzwerk als Knoten beitreten. Endpunkte bilden sogenannte ‚End device'-Module, die keine Routing-Funktion wie der Coordinator oder die Router übernehmen können. Diese End devices sind optional in einem XBee-Netzwerk und für die vorliegende Arbeit wird kein XBee-Modul als End device konfiguriert.

Abb. 7: Aufbau eines XBee/ZigBee-Netzwerks



Quelle: [8]

Auf jedes XBee-Modul muss zunächst eine aktuelle Firmware per XTCU geschrieben werden. Ein XBee-Modul erhält die Coordinator-Firmware ‚ZigBee Coordinator AT' in der Version 20A7, alle anderen die Router-Firmware ‚ZigBee Router AT' in der Version 22A7. Alle XBee-Module müssen zu einer miteinander kompatiblen Product family gehören, in diesem Fall ‚XB24-ZB'. Die Module arbeiten ausschließlich im AT-Modus (Transparent Mode), der API-Modus wird nicht verwendet.

Abb. 8: XBee-Modul-Firmware

Product family: XB24-ZB	**Function set:** ZigBee Coordinator AT	**Firmware version:** 20A7
Product family: XB24-ZB	**Function set:** ZigBee Router AT	**Firmware version:** 22A7

Quelle: [9]

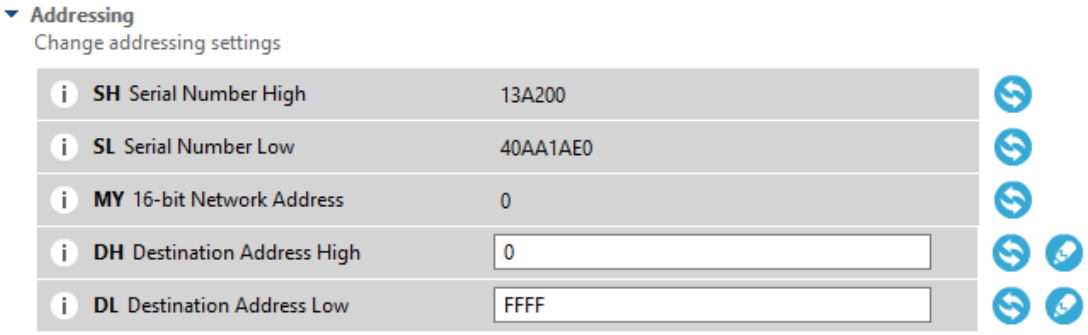
Das XBee-Modul, welches sich auf dem Server-Modul befindet, wird als Coordinator konfiguriert, alle XBee-Module der Client-Module als Router. Damit sich alle XBee-Module im gleichen Netzwerk befinden, muss die PAN ID, welche die Netzwerk-Nummer angibt, bei allen Modulen identisch sein.
Ein Wert von ‚0' hätte eine zufällige PAN ID zur Folge und kann somit nicht gewählt werden. Eine PAN ID zwischen 0x0001 und 0x7FFF steht zur Auswahl. Als PAN ID wurde hier ‚ABCD' (hex) gewählt, unter dem Abschnitt ‚Networking' lässt sich diese eintragen. Als Wert für Scan Channels sollte ‚FFFF' (hex) eingetragen sein.

Abb. 9: XBee-Netzwerk PAN ID

Quelle: [9]

Im Einstellungs-Abschnitt ‚Addressing' muss die ‚Destination Address' angegeben werden. Das Coodinator-XBee-Modul auf dem Server-Modul soll per Broadcast an alle Client-Module senden. Für Broadcast wird bei der Destination Address High ‚0' und bei Destination Address Low ‚FFFF' (hex) eingestellt.

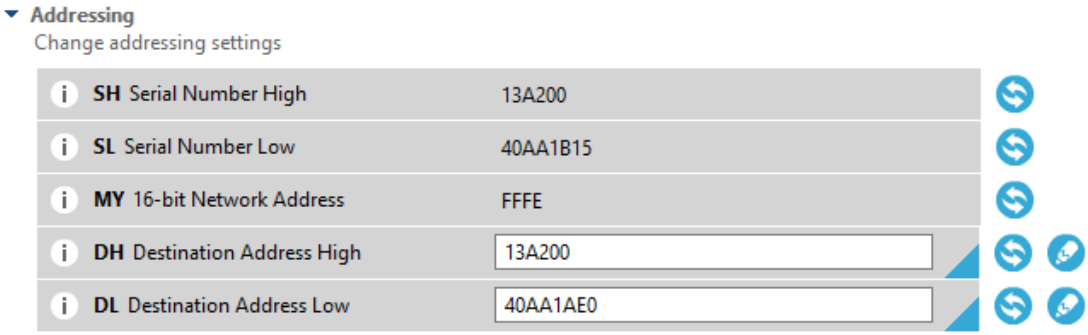
Abb. 10: XBee-Adress-Einstellungen für das Server-Modul



Quelle: [9]

Jedes Router-XBee-Modul auf den Client-Modulen darf nur gezielt an das Coordinator-XBee-Modul (Server-Modul) senden. Dies wird erreicht, indem als Destination Address High ‚13A200' (hex) eingetragen wird und als Destination Address Low ‚40AA1AE0' (hex). Um ein Modul als ‚Destination' zu wählen, müssen dessen Serial Number High und Serial Number Low entsprechend bei Destination Address High und Low eingetragen werden. Die Serial Number High ist bei allen Modulen gleich (13A200), die Serial Number Low bei jedem unterschiedlich und somit einzigartig. Alle Router-XBee-Module erhalten als Destination Address Low den Wert, der beim Coordinator-XBee-Modul als Serial Number Low angeben ist.

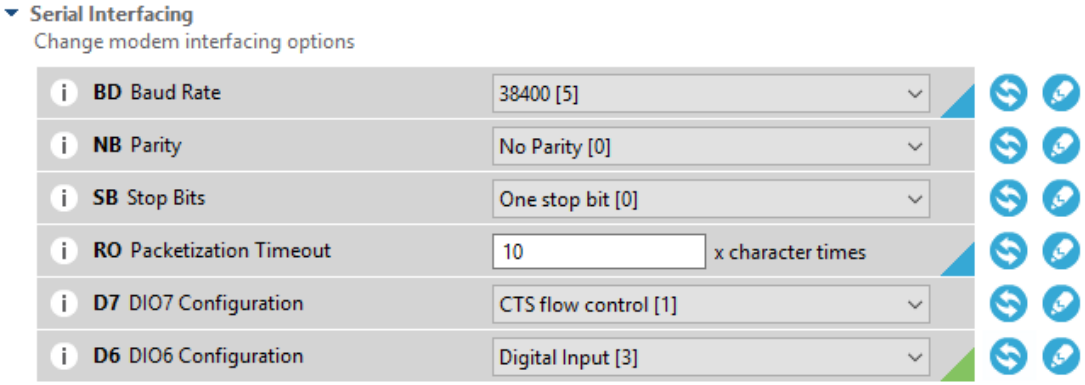
Abb. 11: XBee-Adress-Einstellungen für ein Client-Modul



Quelle: [9]

Als letzten Schritt müssen bei allen XBee-Modulen die gleichen Einstellungen unter dem Abschnitt ‚Serial Interfacing' getroffen werden. Wenn die Konfiguration der seriellen Schnittstelle nicht aufeinander abgestimmt ist, kann die Kommunikation nicht funktionieren. Folgende Einstellungen wurden getroffen:

Abb. 12: XBee-Einstellungen der seriellen Schnittstelle



Quelle: [9]

Die Baud-Rate wurde auf 38400 gesetzt und ist auch in den Arduino-Programmen eingestellt. Packetization Timeout wurde zu Testzwecken bei allen Modulen auf ‚10' gestellt. Dieser Wert kontrolliert, wie die zu verschickenden Daten zusammengefasst werden. Die anderen Werte wurden auf default belassen, müssen generell auf Gleichheit geprüft werden.
Damit wären die XBee-Module fertig eingerichtet. Das XBee-Modul welches vom Arduino-Server verwendet wird, funkt die seriellen Daten an alle anderen XBee-Module (Broadcast), die zu den Arduino-Clients gehören. Bei den XBee-Modulen der Arduino-Clients ist als Ziel-Adresse das XBee-Modul des Arduino-Servers fest eingestellt (point-to-point).

2.2.2.3 Android-App als Fernbedienung

Im Folgenden wird die Implementierung der Android-App ‚ArduinoControllerClient' beschrieben, die eine Client- und Monitoring-Rolle zum Server-Modul und den Client-Modulen einnimmt.

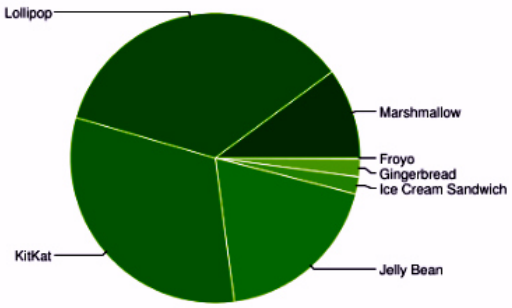
2.2.2.3.1 Android-Entwicklungsumgebung – Android Studio

Zur Entwicklung der Android App wurde ausschließlich das von Google frei angebotene ‚Android Studio' verwendet, welches sich aktuell in Version 2.1.2 befindet. Zum Entwicklungsbeginn stand es auf Version 1.5.x. Android Studio ist, da es von Google entwickelt wird, stets auf dem aktuellen Android SDK-Stand und bietet somit die aktuellen System-Neuerungen ohne Verzögerung. Neben dem Entwicklungsprogramm muss ein aktuelles Java SE Development Kit (aktuell Java 8 x86 / x64) installiert werden. Die Installation der restlichen benötigten Software, wie ein Android SDK in der zu verwendenden Version, lässt sich Programm-intern über den Android SDK Manager durchführen. Für das Ausführen der App wurde im vorliegenden Fall nicht der enthaltene Simulator verwendet, sondern die App wurde zum Test direkt auf einem Android-Smartphone ausgeführt, auf dem Android 5.0.1 Lollipop (API 21) installiert war.
Aus diesem Grund wurde für das Android-App-Projekt als Target SDK Version Android 5.0.1 Lollipop (API 21) gesetzt. Um eine Abwärtskompatibilität zu bieten, wurde als Min SDK Version Android 4.1.2 Jelly Bean (API 16) gewählt. Die Wahl wurde gezielt so getroffen, dass auf aktuelle Funktionen nicht verzichtet werden muss und trotzdem ein großer Bereich von aktuell genutzten Versionen abgedeckt wird (siehe Abb. 13) Kompiliert wurde mit der aktuellen Release Version Android 6.0.1 Marshmallow (API 23). Google garantiert die Aufwärtskompatibilität, so dass die App auch auf folgenden SDKs funktionieren wird. Für die Abwärtskompatibilität muss der Entwickler Sorge tragen.

Abb. 13: Anteile derzeit genutzter Android-Versionen

Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	2.0%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	1.9%
4.1.x	Jelly Bean	16	6.8%
4.2.x		17	9.4%
4.3		18	2.7%
4.4	KitKat	19	31.6%
5.0	Lollipop	21	15.4%
5.1		22	20.0%
6.0	Marshmallow	23	10.1%

Data collected during a 7-day period ending on June 6, 2016.
Any versions with less than 0.1% distribution are not shown.

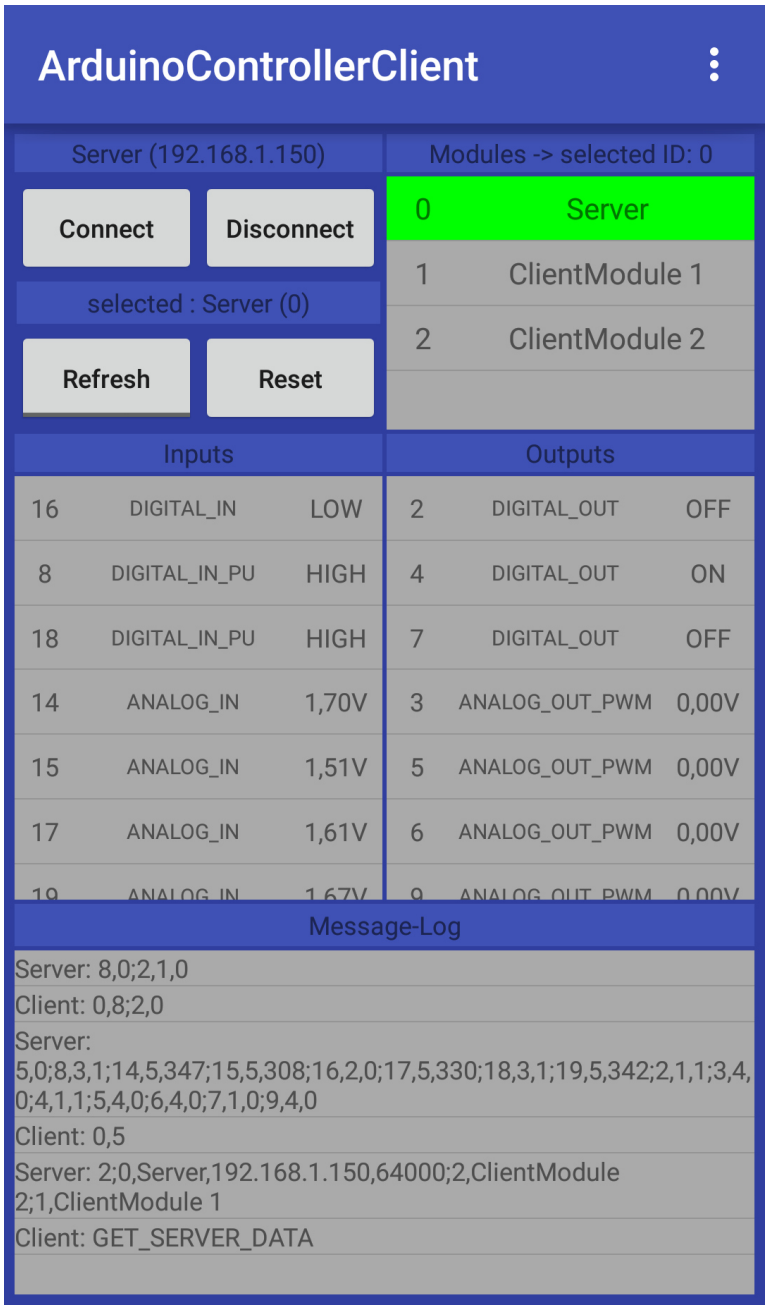


Quelle: [10]

2.2.2.3.2 Grafische Benutzeroberfläche – GUI

Zur Vereinfachung wurde eine Ein-Activity-App entworfen. Ein Vorteil ist, dass alle Daten auf einem Screen angezeigt werden. Durch mehrfach ineinander geschachtelte Linear-Layouts passt sich die Darstellung im Verhältnis der Displaygröße und Pixeldichte an, was durch das Layout-Attribut ‚layout_weight', welches die prozentuale Gewichtung angibt, erreicht wird. Das Layout ist auf Hochkant-Darstellung ausgelegt, somit ist dies auch fest im Quellcode gesetzt. Für Querformat würde es sich in dieser Form nur auf einem Tablet eignen.

Abb. 14: Android-App - MainActivity



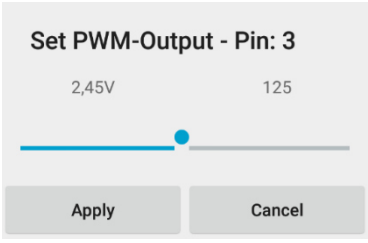
Quelle: [11]

Im Folgenden wird der Aufbau der GUI beschrieben:

Die MainActivity besteht zum einen aus 4 ListViews (siehe Abb. 14):

Modules: Listet alle Module als Module-Labels auf, die aus der ID und einem Namen bestehen. Hierüber kann mittels Touch auf einen Eintrag ein Modul selektiert werden, was dadurch für die Bedienung (Steuerung und Abfrage der Pins) aktiviert ist. Durch einen langen Touch auf ein Client-Modul-Label wird es vom Server und aus dessen Client-Liste entfernt und daraus folgend auch der Label-Eintrag.
Inputs: Listet alle Arduino-Pins auf, die auf dem selektierten Modul als Input gesetzt wurden. Ein Pin-Eintrag besteht aus der Arduino Pin-Nummer (pinNr), dem eingestellten Pin-Modus (pinMode) und dem aktuell ausgelesenen Wert, bei DIGITAL_IN oder DIGITAL_IN_PU (pull-ups gesetzt) 0/1 bzw. LOW/HIGH, bei ANALOG_IN eine 10-bit-stufiger 0-5V-Wert (pinValue).
Beim Touch auf einen Eintrag (Input-Pin) wird ein Request verschickt, der den Eingang abfragt und dessen Wert in einer Response zurücksendet, welche das Smartphone anzeigt
Durch einen Touch auf das Label ‚Inputs' über der zugehörigen ListView kann zwischen der Sortierung nach Pin-Nummer oder Pin-Modus gewechselt werden.
Outputs: Listet alle Arduino-Pins auf, die auf dem selektierten Modul als Output gesetzt wurden. Ein Eintrag besteht aus der Arduino Pin-Nummer (pinNr), dem eingestellten Pin-Modus (pinMode) und dem aktuell nach einer Ausgangs-Änderung ausgelesenen Wert, bei DIGITAL_OUT 0/1 bzw. OFF/ON und bei ANALOG_OUT_PWM (pseudo-analog) ein 8-bit-stufiger 0-5V Wert (pinValue).
Beim Touch auf einen Eintrag (Output-Pin) wird ein Request verschickt, der den Ausgang bei Pin-Modus DIGITAL_OUT toggelt und bei ANALOG_OUT_PWM einen ‚Set PWM-Output'-Dialog (siehe Abb. 15) aufruft, über den der Ausgangswert per SeekBar einstellbar ist. Durch einen Touch auf das Label ‚Outputs' über der zugehörigen ListView kann zwischen der Sortierung nach Pin-Nummer oder Pin-Modus gewechselt werden.

Abb. 15: Android-App - PWM-Dialog



Quelle: [11]

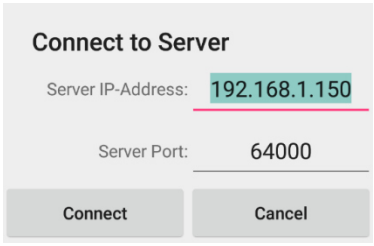
Message-Log: Hier werden, wenn aktiviert, alle Nachrichten, die zwischen Android-Client und Server-Modul ausgetauscht werden, auch diese, die an Client-Module weitergeleitet werden, aufgelistet. Die Einträge werden vergleichbar zu einem Stack eingefügt, d.h. der aktuellste Eintrag, wie eine Response auf einen Request, erscheint oben. Ein Eintrag, der mit ‚Client:' beginnt, zeigt die Nachricht (Request) an, die von dem Smartphone verschickt wurde, ‚Server:' zeigt an, dass es sich um eine Nachricht (Response) vom Server handelt und ‚ClientModule:' steht für eine von einem Client-Modul zurückgeleitete Nachricht (Response).

Neben den ListViews sind 4 Buttons auf der Oberfläche zu finden (siehe Abb. 14):

Server-spezifisch:

Connect: Ruft einen ‚Connect to Server'-Dialog (siehe Abb. 16) auf, wenn das Smartphone nicht bereits verbunden ist. Hierüber ist eine TCP-Verbindung zum Server-Modul aufbaubar. Die Server IP-Adresse und der Port sind wählbar. Bei erfolgreicher Verbindung zum Server wird dieser als Modul-Label in der Modules-ListView angezeigt und per default ausgewählt. Alle beim Server als Ausgang und Eingang gesetzten Arduino-Pins werden vom Server abgerufen und angezeigt sowie entsprechend den Inhalten der ListViews ‚Inputs' und ‚Outputs' die Buttons ‚Refresh' und ‚Reset' aktiviert. Bereits beim Server angemeldete Client-Daten werden direkt nach dem Anmelde-Request in der Server-Response zurückgesendet und in der Module-Liste eingetragen.

Abb. 16: Android-App - Connect-Dialog



Quelle: [11]

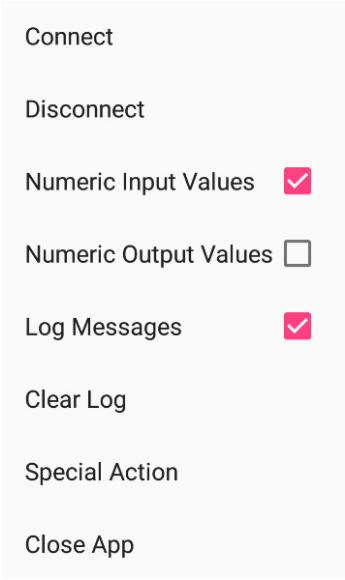
Disconnect: Beendet die TCP-Verbindung. Die ListViews ‚Modules', Inputs' und ‚Outputs' werden geleert, die Buttons ‚Refresh' und ‚Reset' werden deaktiviert. Der ‚Message-Log'-ListView bleibt unverändert.

Modul-spezifisch (über Modules selektiert):

Refresh (ToggleButton): Startet einen Thread (Klasse: RequestSender), der periodisch einen Request versendet, der die Daten aller Input-Pins des selektierten Arduino-Moduls abfragt. Somit können die Eingangswerte in bestimmten Zeitabschnitten aktualisiert werden. Der Button ist nur aktiv, wenn vorkonfigurierte Input-Pins vorhanden sind. Durch erneuten Touch auf gesetzten ToggleButton oder beim Wechsel zu einem anderen Arduino-Modul wird der Thread gestoppt.
Reset: Verschickt einen Reset-Request, der alle Ausgänge des selektierten Arduino-Moduls auf ‚0' bzw. ‚OFF' stellt. Die Ausgänge werden nach dem Reset/Clear ausgelesen und diese Werte werden in der Response zurückgeschickt, wo sie dann entsprechend angezeigt werden.

Die App besitzt ein Activity-Menü (siehe Abb. 17), in dem neben einigen Hauptfunktionen zur alternativen Auswahl, Nebenfunktionen und verschiedene Einstellungsmöglichkeiten zu finden sind.

Abb. 17: Android-App - Activity-Menü

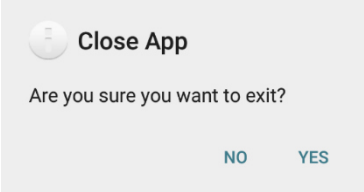


Quelle: [11]

Connect: Die gleiche Funktion wie der Button ‚Connect'
Disconnect: Die gleiche Funktion wie der Button ‚Disconnect'

Numeric Input Values: Wenn die Checkbox nicht gesetzt ist, werden statt Zahlenwerten entsprechende Spannungswerte bei DIGITAL_IN(_PU)-Pins (0/1 -> LOW/HIGH) und ANALOG-IN-Pins (10bit, 0-1023 -> 0.00-5.00V) angezeigt.
Numeric Output Values: Wenn die Checkbox nicht gesetzt ist, werden statt Zahlenwerten entsprechende Spannungswerte bei DIGITAL-OUT-Pins (0/1 -> OFF/ON) und ANALOG_OUT_PWM (8bit, 0-255 -> 0.00-5.00V) angezeigt.
Log Messages: Bei deaktivierter Checkbox wird keine Nachricht im ‚Message-Log'-ListView angezeigt.
Clear Log: Löscht den gesamten Inhalt der ‚Message-Log'-ListView
Special Action: Startet einen Thread (Klasse: RequestSender), der in schneller Abfolge Requests zum Ändern von allen definierten PWM-Ausgangswerten des selektierten Moduls verschickt, um die Übertragungsgeschwindigkeit zu testen.
Close App: Schließt die App nach einem Bestätigungsdialog. Erscheint auch, wenn durch den Back-Button das Schließen der App eingeleitet wird (siehe Abb. 18)

Abb. 18: Android-App – Close-Dialog

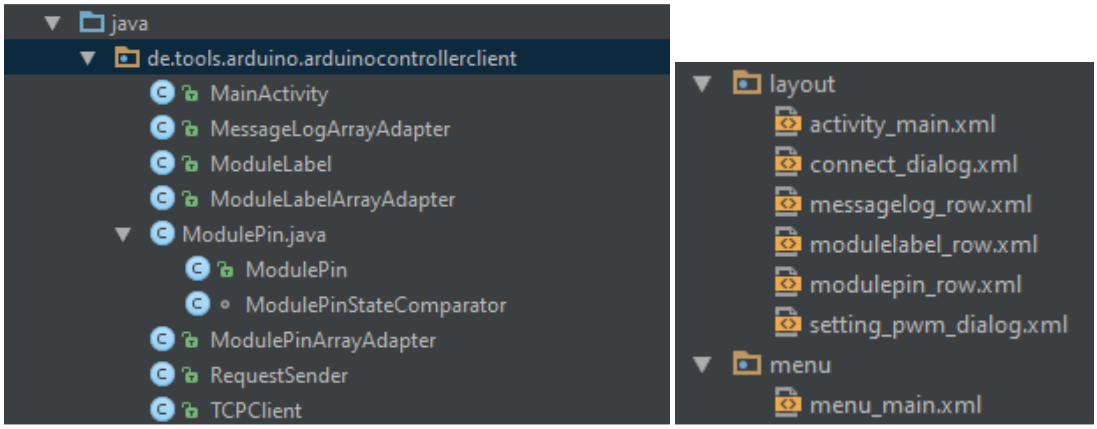


Quelle: [11]

2.2.2.3.3 Klassen der Android-App

Im Folgenden wird auf die einzelnen Klassen der Android-App eingegangen. Die Erläuterungen werden auf das Wesentliche beschränkt, was zum Verständnis der Grundstruktur wichtig ist. Auf Erklärungen zu Klassen- und Objektmethoden, die Standard-Funktionen abbilden, wird verzichtet. Variablen-Bezeichner und Methoden-Namen wurden so gewählt, dass deren Funktion leicht ersichtlich ist. Als Quellcode-Sprache wurde generell Englisch verwendet.

Abb. 19: Android-App - Klassen- und Layout-Dateien



Quelle: [12]

ModuleLabel: Enthält die Label-Daten eines Arduino-Moduls welche in der ‚Modules'-ListView angezeigt werden. Findet zunächst nur als Datenstruktur Verwendung.

```
public final int uniqueId;
public final String name;
```

ModulePin: Enthält alle Daten eines Arduino-Pins, die in der ListView ‚Inputs' oder ‚Outputs' dargestellt werden.

```
public static final double HIGH_VOLTAGE = 5.0;
public static final int PWM_MAX_VALUE = 255;
public static final int ANALOG_MAX_VALUE = 1023;

public final int pinNr;
public final PinMode pinMode;
private int pinValue;
```

Jeder Pin besitzt einen der folgenden Modi, welche durch das Enum ‚PinMode' repräsentiert werden (ein Pin im Modus: NOT_DEFINED wird nicht in den Listen aufgeführt):

```
public enum PinMode {
      NOT_DEFINED,
      DIGITAL_OUT,
      DIGITAL_IN,
      DIGITAL_IN_PU,
      ANALOG_OUT_PWM,
      ANALOG_IN}
```

ModuleLabelArrayAdapter: Beschreibt die zur ListView ‚Modules' zugehörige Adapter-Klasse, abgeleitet von ArrayAdapter. Speichert zusätzlich die ausgewählte Position zur Hervorhebung bei Auswahl und die ID des Servers zur Unterscheidung von anderen Modul-IDs. Verwendet für das Layout eines Listen-Eintrags die Datei ‚modulelabel_row.xml'.

```
public static final int NO_SELECTED_POSITION = -1;

private List<ModuleLabel> moduleLabelList;
private Context context;
private int selectedPosition;
private int serverUniqueId;
```

ModulePinArrayAdapter: Beschreibt die zu den ListViews 'Inputs' und 'Outputs' zughörige Adapter-Klasse, abgeleitet von ArrayAdpater. Speichert zusätzlich die Information, ob die Pin-Werte als Zahlenwert oder als Spannungswert dargestellt werden und in welcher Form die Pin-Einträge sortiert werden, nach Pin-Nummer oder Pin-Modus. Verwendet wird für das Layout eines Listen-Eintrags die Datei ‚modulepin_row.xml'.

```
public static final boolean NUMERIC_VALUES_DEFAULT =
false;

private List<ModulePin> modulePinList;
private Context context;
private boolean sortToggle;
private boolean numericValues;
```

MessageLogArrayAdapter: Die zu der ListView ‘Message-Log’ zugehörige Adapter-Klasse, abgeleitet von ArrayAdapter. Speichert zusätzlich die Information, ob das Aufnehmen der Einträge aktiviert ist.
Verwendet für das Layout eines Listen-Eintrags die Datei ‚messagelog_row.xml‘.

```
public static final boolean LOG_MESSAGES_DEFAULT = true;

private List<String> messageLogList;
private Context context;
private boolean allowedAdding;
```

TCPClient: Aus dieser Klasse wird in der MainActivity über das Android-Thread-Konstrukt ‚AsyncTask‘ ein Objekt erzeugt, welches parallel zum MainThread (UI-Thread) auf Responses vom Server-Modul wartet und diese bei Eintreffen zur Verarbeitung und Ausgabe an den UI-Thread weiterreicht. Eingeleitet wird der AsyncTask durch den erfolgreichen TCP-Verbindungsaufbau über den ‚Connect‘-Button. Bei Abmeldung über den ‚Disconnect‘-Button wird der AsyncTask beendet. In diesem Fall oder bei Verbindungsfehlern wird der TCP-Socket geschlossen und der Objekt-Speicher wieder freigegeben. Beim Eintreffen einer Response wird die beim Erzeugen eines TCPClients implementierte messageReceived(String message)-Methode aufgerufen, die die publishProgess(message) ausführt, die wiederum die AsyncTask-Methode onProgressUpdate() startet, welche im UI-Thread abläuft. Dies sorgt dann dafür, dass alle Daten aus der Nachricht an die entsprechenden Views weitergereicht werden. Dies muss im UI-Thread ablaufen, da bei Android eine klare Trennung zwischen Hintergrund-Threads und MainThread, der allein für das Aufbauen der GUI reserviert ist, bleiben muss und der MainThread keine zeitintensiven Abläufe beinhalten darf. Dies würde den UI-Thread behindern und zu einer trägen Darstellung und langsamer bis gar keiner Reaktion auf Eingaben führen. Über die Methode sendMessage(String message), welche im MainThread aufgerufen wird, wird ein Request an das per TCP verbundene Server-Modul verschickt.

Folgende Requests sind über ein Enum definiert:

```
public enum Request {    UNDEFINED_REQUEST,
                         CLIENT_STOPPED,
                         GET_SERVER_DATA,
                         REMOVE_CLIENT_MODULE,
                         GET_PIN_DATA,
                         GET_ALL_PINS_DATA,
                         GET_INPUT_PINS_DATA,
                         GET_OUTPUT_PINS_DATA,
                         SET_PIN_VALUE,
                         SET_MULTI_PIN_VALUES,
                         RESET_OUTPUT_PINS}
```

CLIENT_STOPPED: Gibt dem Server-Modul zu erkennen, dass sich der Android-Client abmeldet.
GET_SERVER_DATA: Fordert vom Server-Modul alle aktuellen Daten an wie ID, Name, IP-Adresse und Port. Zusätzlich werden alle Daten von aktuell am Server angemeldeten Client-Modul in der Response mitgesendet, welche wie beim Server je aus ID und Name bestehen.
REMOVE_CLIENT_MODULE: Sorgt dafür, dass das Server-Modul ein bestimmtes Client-Modul aus der Client-Liste entfernt. Dies wird vom Server und beim betreffenden Client-Modul bestätigt.
GET_PIN_DATA: Fordert die Daten eines einzelnen Arduino-Pins von einem Arduino-Modul (Server oder Client) an, die je aus Pin-Nummer, Pin-Modus und Pin-Wert bestehen.
GET_ALL_PINS_DATA: Fordert Daten (siehe PIN_DATA) aller Pins von einem Arduino-Modul (Server oder Client) an, die bei diesem als zu nutzende Pins (Eingang oder Ausgang) vorkonfiguriert sind. Dieser Request wird automatisch beim Anmelden am Server oder beim Wechsel auf ein anderes Modul versendet.
GET_INPUT_PINS_DATA: Fordert Daten (siehe PIN_DATA) aller Pins von einem Arduino-Modul (Server oder Client) an, die bei diesem als Eingangs-Pins vorkonfiguriert sind.
GET_OUTPUT_PINS_DATA: Fordert Daten (siehe PIN_DATA) aller Pins von einem Arduino-Modul (Server oder Client) an, die bei diesem als Ausgangs-Pins vorkonfiguriert sind.
SET_PIN_VALUE: Veranlasst das Ändern des Pin-Wertes eines Ausgangs-Pins (als Ausgang vorkonfiguriert) von einem Arduino-Modul (Server oder Client). Liefert als Response darauf die entsprechenden Ausgangs-Pin-Daten (siehe PIN_DATA).

SET_MULTI_PIN_VALUES: Veranlasst das Ändern von mehreren Pin-Werten von Ausgangs-Pins (als Ausgänge vorkonfiguriert) eines Arduino-Moduls (Server oder Client) enthalten in einem Request. Liefert als Response darauf die Pin-Daten aller zu ändernden Ausgangs-Pins (siehe PIN_DATA).
RESET_OUTPUT_PINS: Veranlasst das Löschen (Pin-Wert = 0) aller Ausgangs-Pin-Werte (wenn als Ausgang vorkonfiguriert) eines Arduino-Moduls (Server oder Client). Liefert als Response darauf die Pin-Daten aller Ausgangs-Pins (siehe PIN_DATA).
RequestSender: Die Klasse ist von ‚Thread' abgeleitet, somit kann ein Objekt daraus als paralleler Thread gestartet werden. Durch den Toggle-Button ‚Refresh' oder durch den Activity-Menü Eintrag ‚Special' wird ein Thread gestartet, welcher periodisch Requests zum Abfragen von Eingangs-Pin-Daten oder Setzen von Pin-Werten versendet.
MainActivity: In dieser Klasse werden alle Views deklariert, die zur Anzeige der Arduino-Modul-Daten benötigt werden. In der Activity-Initialisierungs-Methode onCreate(Bundle savedInstanceState) werden die Views und Buttons mit den entsprechenden Layout-Definitionen in der Layout-Datei ‚activity_main.xml' verknüpft und mit Listenern versehen, um ihnen Touch-Funktionen zuzuweisen. In dieser Klasse ist die innere Klasse ‚ClientTask' enthalten, die von AsyncTask abgeleitet ist und für das Warten auf Responses und deren Weiterleiten an den Main-/UI-Thread zuständig ist. Ein aus ihr erzeugtes Task-Objekt erstellt parallel zum MainThread ein Client-Objekt von der Klasse TCPClient.
Die onProgressUpdate(String… values)-Methode des ClientTask, die im UI-Thread abläuft, sorgt für das Auswerten der Responses anhand der Request/Response-Nummer, die in den Nachrichten enthalten ist und ruft bei Responses auf Requests, die sich auf Arduino-Pins beziehen, die Methode addOrRefreshModulePin(String responsePinPart) auf, welche die aktuellen Daten in die Views einträgt. Bei einer GET_SERVER_DATA-Response werden alle enthaltenen Daten in der ‚Modules'-ListView eingetragen, bei einer REMOVE_CLIENT_MODULE-Response entsprechend entfernt.
In der MainActivity wird das Activity-Menü in der Methode onCreateOptionMenu(menu) angelegt, deren Layout in der Datei ‚menu_main.xml' definiert ist. In der Methode onOptionsItemSelected(MenuItem item) werden die Menü-Aktionen verwaltet. Jede CheckBox-Einstellung, die darin vorgenommen wird, wird, wenn sich die App im Pause-Zustand befindet, durch die Methode saveSettings(), die einen AsyncTask startet in einer SharedReference-Datei, festgehalten. Beim Start der App werden durch die Methode loadSettings(), die einen AsyncTask startet, die Daten der SharedReference-Datei eingelesen und gesetzt. Zu den Menü-Einstellungen werden die Server-IP-Adresse und der Server-Port aus dem ‚Connect-Dialog' in der SharedPreference-Datei abgespeichert.

In der Methode showConnectDialog() wird der ‚Connect-Dialog' erzeugt, der beim Betätigen des ‚Connect'-Buttons erscheint und die Eingabe von IP-Adresse und Port des Server-Moduls erwartet. Die Eingaben werden auf Korrektheit geprüft, sodass nur gültige Werte gewählt und gespeichert werden können. Das zugehörige Layout ist in der Datei ‚connect_dialog.xml' definiert.
In der Methode showSettingsPwmDialog() wird der ‚Set PWM-Output'-Dialog erzeugt, der beim Anwählen eines ANALOG_PWM-OUTPUT-Pins das Einstellen des PWM-Wertes erlaubt.
Das zugehörige Layout ist in der Datei ‚setting_pwm_dialog.xml' definiert.
In der Methode showCloseDialog() wird der ‚Close App'-Dialog erzeugt, der bei der Auswahl des Eintrages ‚Close App' im Activity-Menü oder beim App-Verlassen durch den Back-Button erscheint und eine Bestätigung zum Beenden der App erwartet. Das Layout verwendet das Standard-Layout vom Android-AlertDialog.

2.2.2.4 Übertragungs- und Datenprotokolle

Für die Übertragung der Daten werden zwei Übertragungsformen verwendet. TCP wird für die Übertragung zwischen Android-Client und Server-Modul gewählt. Die serielle Schnittstelle (UART) wird gebraucht, um Daten zwischen Server-Modul und Client-Modulen über die XBee-Funk-Module auszutauschen. In welcher Form die Daten mit diesen Übertragungsprotokollen weitergeleitet werden, wird im anschließenden Abschnitt 2.2.2.4.3 ‚Datenprotokoll - Aufbau der Nachrichten' erläutert. Im abschließenden Abschnitt wird eine alternative Datenübertragung in Form eines ByteStreams zwischen den Arduino-Modulen als Erweiterung aufgeführt.

2.2.2.4.1 Android-Client <-> Server-Modul

Für die Übertragung der Daten zwischen Android-Client und Server-Modul wird ein TCP-Socket verwendet. TCP ist als Grundfunktion in der Arduino-Ethernet-Bibliothek integriert und lässt sich problemlos verwenden. Der Einsatz ist von Arduino gut dokumentiert und wird durch Beispiel-Sketches unterstützt. UDP kann auch eingesetzt werden, empfiehlt sich aber nicht, da es verbindungslos ist und bei einer Steuerung von Ausgängen eine sichere Übertragung mit verlässlicher Paket-Reihenfolge gewährleistet sein muss.

2.2.2.4.2 Server-Modul <-> Client-Modul

Für die Übertragung der Daten zwischen Server-Modul und Client-Modulen wird die serielle Schnittstelle (UART) des Mikrocontrollers auf dem Arduino-Board verwendet. Der Zugriff auf die Schnittstelle wird durch Arduino-Funktionen unterstützt und vereinfacht. Der Mikrocontroller auf dem Arduino UNO bietet im Vergleich zum auf dem MEGA integrierten Mikrocontroller nur eine serielle Schnittstelle in Hardwareform. Das XBee-Modul ist über diese Schnittstelle mithilfe des XBee-Shields verbunden und sendet die seriellen Daten per Funk zu andern XBee-Modulen oder empfängt Daten, die daraufhin an die serielle Schnittstelle des Mikrocontrollers gereicht werden. Die XBee-intern ablaufende Übertragung und Verwaltung der Daten bzw. deren Protokolle müssen nicht betrachtet werden. Für die serielle Datenverbindung von Server zu Client-Modul über die XBee-Funkmodule wurde als default Baud-Rate 38400 eingestellt, welche die geringste Abweichung bei dem 16 MHz-getakteten Mikrocontroller aufweist und einen guten Kompromiss zwischen Übertragungsgeschwindigkeit und Stabilität darstellt. Die Baud-Rate muss auf allen Arduino-Modulen vorkonfiguriert werden, sowie bei den XBee-Modulen.

2.2.2.4.3 Datenprotokoll - Aufbau der Nachrichten

Für das Speichern der ausgelesenen Request-Daten und für das Zusammenfügen der Response-Daten wurden bei den Arduino-Modulen jeweils zwei Puffer verwendet:

```
char readBuffer[8 + (PIN_COUNT - PWM_PIN_COUNT) * 5 +
PWM_PIN_COUNT * 7];
char writeBuffer[8 + (PIN_COUNT - PWM_PIN_COUNT - ANALOG_IN_PIN_COUNT)
* 7 + PWM_PIN_COUNT * 9 + ANALOG_IN_PIN_COUNT * 10];
```

readBuffer[]: Der Eingangspuffer für die Requests vom Android-Client oder einer weiterzuleitenden Response eines Client-Moduls.
writeBuffer[]: Der Ausgangspuffer für die Responses auf einen Request vom Android-Client an den Server oder ein Client-Modul.
Um den eng bemessen Daten-Speicher des Mikrocontrollers zu schonen, wurde nur der maximal benötigte Speicherbereich reserviert, damit jede Nachricht ohne Überlauf aufgenommen werden kann. Als Alternative bietet Arduino auch den Datentyp String an, der die C-string Verwaltung kapselt und Zusatzfunktionen mitbringt. Durch

Arduino Strings erhöht sich die Gefahr während der Laufzeit zu viel Datenspeicher zu belegen. Der Heap wächst durch Anlegen von Strings dynamisch an und wird nicht sicher freigegeben. Somit bleibt wenig Platz für den Stack oder es kommt im schlimmsten Fall zu Heap-Stack Überschneidungen, was zu unbestimmten Programmabläufen führen kann. Ein Mikrocontroller bricht in diesem Fall nicht sein Programm ab und kann keine Fehlermeldung geben.
Durch fest definierte statische Puffer kann vor dem Programmstart besser eingeschätzt werden, wie viele SRAM-Ressourcen während der Laufzeit zur Verfügung stehen, was bei Mikrocontrollern einen hohen Stellenwert einnimmt.

Folgende Arduino-Programm-Zeilen (Server und Client) legen Zeichen fest, die für den Aufbau des Datenprotokolls verwendet werden:

```
const char MESSAGE_SEPARATORS[] = ";,";
const char MESSAGE_SEGMENT_SEPARATOR = MESSAGE_SEPARATORS[0];
const char MESSAGE_SEPARATOR = MESSAGE_SEPARATORS[1];
#ifdef ADDED_SERIAL_START_CHAR
const char SERIAL_START_CHAR = '<';
#endif
const char SERIAL_STOP_CHAR = '>';
```

MESSAGE_SEGMENT_SEPARATOR: -> Semikolon: wird zur Trennung von Nachrichten-Header und zwischen einzelnen Datensegmenten verwendet.
MESSAGE_SEPARATOR: -> Komma: wird zur Trennung der einzelnen Daten-Werte eines Segments verwendet.
SERIAL_START_CHAR: Zur besseren Erkennung kann optional ein Zeichen für den Anfang der seriell übertragenen Nachricht verwendet werden (siehe ‚#define ADDED_SERIAL_START_CHAR' in der Header-Datei).
SERIAL_STOP_CHAR: An dem Zeichen wird das Ende einer seriell übertragenen Nachricht erkannt.

Auflistung der Nachrichten

Nachfolgend wird der Aufbau aller implementierten Requests mit dazugehörigen Responses aufgeführt. Die Header-Daten einer Nachricht bestehen immer aus der Modul-ID und dem Request-Code, die Reihenfolge der Werte ist bei Request und Response vertauscht. Die Nachrichten werden zunächst als Strings zusammengesetzt (Android-App: Java String-Objekte, Arduino: C-strings -> char-Arrays) und

daraufhin zeichenweise per TCP bzw. serieller Schnittstelle übertragen. Responses, die von jedem Arduino-Modul kommen können, die somit auch der Server nur von einem Client-Modul weiterreicht, werden nur einmal aufgelistet. Beispiele für die Nachrichten-Strings werden zusätzlich in Anführungszeichen angegeben.

Die ersten drei Requests definieren die Server-Funktionalität. Diese Request werden nur zwischen Android-Client und Server-Modul ausgetauscht, ausgenommen Request 3, der außerdem an Client-Module weitergeleitet wird:

CLIENT_STOPPED_REQUEST (1)
Request (Android-Client): “CLIENT_STOPPED”
Response (Server-Modul):
<CLIENT_STOPPED_REQUEST_CODE>,<SERVER_GOODBYE>
-> “1,Bye…”

GET_SERVER_DATA_REQUEST (2)
Request (Android-Client): “GET_SERVER_DATA”
Response (Server-Modul): <GET_SERVER_DATA_REQUEST_CODE>;
<SERVER_MODULE_UNIQUE_ID>,<SERVER_MODULE_NAME>,<IPAddress[0]>,
<SERVER_PORT>;
<MODULE_UNIQUE_ID(1)>,<MODULE_NAME(1)> >;<MODULE_UNIQUE_ID(2)>,
<MODULE_NAME(2)>;…;<MODULE_UNIQUE_ID(n)>,<MODULE_NAME(n)>
-> “2,0,Server,192.168.1.150,64000;1,ClientModule 1;2,ClientModule 2”

REMOVE_CLIENT_MODULE_REQUEST (3)
Request (Android-Client): REMOVE_CLIENT_MODULE,<MODULE_UNIQUE_ID>
-> “REMOVE_CLIENT_MODULE,1”
Response (Server-Modul):
<REMOVE_CLIENT_MODULE_REQUEST_CODE>,<MODULE_UNIQUE_ID>
-> “3,1”
Request (Server-Modul):
<MODULE_UNIQUE_ID>,<REMOVE_CLIENT_MODULE_REQUEST_CODE>
-> “1,3”
Request (Client-Modul):
<REMOVE_CLIENT_MODULE_REQUEST_CODE>,<MODULE_UNIQUE_ID>
-> “3,1”

Die weiteren Requests (4-10) bilden die Client-Funktionalität ab, die alle Client-Module, aber auch das Server-Modul verarbeiten kann:

GET_PIN_DATA_REQUEST (4)
Request (Android-Client):
<MODULE_UNIQUE_ID>,<GET_PIN_DATA_REQUEST_CODE>;<pinNr>
-> “1,4;5”
Response (Arduino-Modul):
<GET_PIN_DATA_REQUEST_CODE>,<MODULE_UNIQUE_ID>;
<pinNr>,<pinMode>,<pinValue>
-> “4,1;5,4,127”

GET_ALL_PINS_DATA_REQUEST (5)
Request (Android-Client):
<MODULE_UNIQUE_ID>,<GET_ALL_PINS_DATA_REQUEST_CODE>
-> “0,5”
Response (Arduino-Modul):
<GET_ALL_PINS_DATA_REQUEST_CODE>,<MODULE_UNIQUE_ID>;
<pinNr(1)>,<pinMode(1)>,<pinValue(1)> ;<pinNr(2)>,<pinMode(2)>,
<pinValue(2)> ;…;<pinNr(n)>,<pinMode(n)>,<pinValue(n)>
-> “5,0;8,3,1;14,5,438;2,1,0;5,4,127”

GET_INPUT_PINS_DATA_REQUEST (6)
Request (Android-Client): <MODULE_UNIQUE_ID>,<
GET_INPUT_PINS_DATA_REQUEST_CODE>
-> “2,6”
Response (Arduino-Modul):
<GET_INPUT_PINS_DATA_REQUEST_CODE>,<MODULE_UNIQUE_ID>;
<pinNr(1)>,<(input)pinMode(1)>,<pinValue(1)>;<pinNr(2)>,
<(input)pinMode(2)>,<pinValue(2)>;…;<pinNr(n)>,
<(input)pinMode(n)>,<pinValue(n)>
-> “6,2;7,2,0;17,5,268”

GET_OUTPUT_PINS_DATA_REQUEST (7)
Request (Android-Client):
<MODULE_UNIQUE_ID>,<GET_OUTPUT_PINS_DATA_REQUEST_CODE>
-> “1,7”
Response (Arduino-Modul):
<GET_OUTPUT_PINS_DATA_REQUEST_CODE>,<MODULE_UNIQUE_ID>;
<pinNr(1)>,<(output)pinMode(1)>,<pinValue(1)>;<pinNr(2)>,
<(output)pinMode(2)>,<pinValue(2)>;…;<pinNr(n)>,
<(output)pinMode(n)>,<pinValue(n)>
-> “7,1;3,4,33;7,1,1”

SET_PIN_DATA_REQUEST (8)
Request (Android-Client):
<MODULE_UNIQUE_ID>,<SET_PIN_DATA_REQUEST_CODE>;<pinNr>,<pinValue>
-> “0,8;6,255”
Response (Arduino-Modul):
<SET_PIN_DATA_REQUEST_CODE>,<MODULE_UNIQUE_ID>;<pinNr>,
<(output)pinMode>,<pinValue>
-> “8,0;6,4,255”

SET_MULTI_PIN_VALUES_REQUEST (9)
Request (Android-Client):
<MODULE_UNIQUE_ID>,<SET_MULTI_PIN_VALUES_REQUEST_CODE>;
<pinNr(1)>,<pinValue(1)> ;<pinNr(2)>,<pinValue(2)>;… ;<pinNr(n)>,<pinValue(n)>
-> “0,9;6,10;2,1;7,0”
Response (Arduino-Modul):
<SET_MULTI_PIN_VALUES_REQUEST_CODE>,<MODULE_UNIQUE_ID>;
<pinNr(1)>,<(output)pinMode(1)>,<pinValue(1)>;<pinNr(2)>,
<(output)pinMode(2)>,<pinValue(2)>;…;<pinNr(n)>,
<(output)pinMode(n)>,<pinValue(n)>
-> “9,0;6,4,10;2,1,1;7,1,0”

RESET_OUTPUT_PINS_REQUEST (10)
Request (Android-Client):
<MODULE_UNIQUE_ID>,<RESET_OUTPUT_PINS_REQUEST_CODE>
-> “1,10”
Response (Arduino-Modul):
<RESET_OUTPUT_PINS_REQUEST_CODE>,<MODULE_UNIQUE_ID>;
<pinNr(1)>,<(output)pinMode(1)>,<pinValue(1)>;
<pinNr(2)>,<(output)pinMode(2)>,<pinValue(2)>;…;
<pinNr(n)>,<(output)pinMode(n)>,<pinValue(n)>
-> “10,1;3,4,0;7,1,0;9,4,0”

Der letzte Request ist nur für die Login-Funktion zwischen Client-Modulen und Server-Modul vorgesehen und stellt somit eine weitere Server-Funktionalität dar:

LOGIN_REQUEST (11)
Request (Client-Modul):
LOGIN_REQUEST;<MODULE_UNIQUE_ID>,<MODULE_NAME>
-> “LOGIN,1;ClientModule 1”
Response (Server-Modul): <MODULE_UNIQUE_ID>,LOGIN_REQUEST
-> “1,LOGIN”

2.2.2.4.4 ByteStream-Datenprotokoll zwischen Arduino-Modulen

Die Übertragungsrate per XBee-Module, die über die serielle Schnittstelle arbeiten, ist stark eingeschränkt. Zudem besitzen die XBee-Module nur einen kleinen Empfangspuffer, was zu Verzögerung bei der Datenübertragung führen kann. Im Vergleich zur schnellen TCP-Datenübertragung zwischen Client-Modul und Server-Modul müssen Abstriche gemacht werden. Um das Datenaufkommen zwischen den Arduino-Modulen zu reduzieren, wurde eine erweiterte Version des Server- und Clientprogramms entwickelt. Die Programme sind in den Sketches ‚ServerModule_ByteStream.ino‘ und ‚ClientModule_ByteStream.ino‘ abgebildet. Die String-Versionen (‚ServerModule.ino‘ und ‚ClientModule.ino‘) sind nicht mit den ByteStream-Versionen kompatibel, so dass entweder bei allen Arduino-Modulen die String-Version oder die ByteStream-Version geflasht werden muss, wenn diese miteinander kommunizieren sollen. Bei der ByteStream-Version des Server-Moduls bleibt die TCP-Kommunikation zum Android-Client unverändert. Die Datenübertragung zwischen den Arduino-Modulen läuft rein Byte-orientiert ab. Ein Request, der für ein Client-Modul bestimmt ist, wird vom Server-Modul über eine Kon-

vertierungsfunktion zu einzelnen Bytes umgewandelt. Somit können die Trennzeichen (, und ;) der String-Daten entfallen und Zahlenwerte nehmen nur ein Byte Platz ein, statt ein Byte für jede Ziffer als ASCII-Zeichen. Die Ausnahme bildet hier der Pin-Wert eines analogen Eingangs-Pins, der 10bit breit ist, somit als zwei Bytes übertragen wird. Der Pin-Wert eines PWM-Pins passt in ein Byte. Beim Empfangen und Übertragen verarbeiten die Client-Module die Daten ausschließlich in ByteStream-Form. Das Zerlegen eines C-strings in den ByteStream und das Zusammensetzen eines ByteStreams in einen C-string wird nur vom Server-Modul durchgeführt. Die per XBee-Funk zu übertragenden Bytes werden insbesondere bei vielen Pin-Daten deutlich reduziert, ein Geschwindigkeitsvorteil konnte aber nicht festgestellt werden. Als Nachteil zeigt sich, dass der ByteStream im Vergleich zum String/C-string kein Stopp-Zeichen, wie das bei der String-Version verwendete ‚\n', enthalten kann. Jeder Byte-Wert kann vorkommen, was eine aufwändigere Start- oder Stopp-Erkennung nötig macht. Implementiert wurde eine Starterkennung aus zwei Bytes mit Werten, die in dieser Form nicht aufeinanderfolgend im ByteStream vorkommen können. Hinter dem zweiten Start-Byte wird die Anzahl der folgenden Bytes zugehörig zum Request bzw. Response-ByteStream in einem zusätzlichen Byte angegeben. Eine Stopp-Erkennung ist somit überflüssig.
Beide Arduino-Programm-Versionen funktionieren gleichwertig und bei der Benutzung der Android-App ist der interne Unterschied bei der Daten-Übertragung nicht festzustellen.

2.3 Zusammenfassung

Zunächst wurde eine Verbindung von der Android-App zum Server-Modul im (W)-LAN Netzwerk via TCP aufgebaut. Danach wurde eine Verbindung des Server-Moduls zu den Client-Modulen über XBee-Funk eingerichtet, wofür die XBee-Module zur Erstellung des XBee-Netzwerks vorkonfiguriert werden mussten. Folgend wurde eine gleichzeitige Verwaltung von Android-Client und Client-Modulen durch das Server-Modul realisiert. Ein flexibles Datenprotokoll zum Austausch von Nachrichten zwischen Android-Client und allen Arduino-Modulen wurde entwickelt. Um alle Requests und Responses des Datenprotokolls verwenden zu können, wurde die Android-App ausgebaut. Durch einen später hinzugefügten Login-Request der Client-Module haben diese eine Plug-and-play-Fähigkeit erhalten.

3. Ergebnis und Ausblick

Im ersten Abschnitt 3.1 ‚Ergebnis und kritische Bewertung' dieses Kapitels sind die Ergebnisse der Arbeit zusammengefasst und es wird kritisch dazu Stellung genommen. Im zweiten Abschnitt 3.2 ‚Weiterführung' werden Vorschläge für Erweiterungen des Systems genannt, die zukünftige Einsatzgebiete betreffen.

3.1 Ergebnis und kritische Bewertung

Eine Lösung für eine Smartphone-basierte flexible und anpassbare Haussteuerung mit einem XBee-Funk-basierten Netzwerk von Client-Modulen für die Steuerung und Überprüfung von Internet of Things-Geräten ist gelungen. Dazu wurde 1) Android-App 2) Netzwerk von Arduino-Modulen und 3) ein Server-Modul, das als Router zwischen Android-Client und Client-Modulen fungiert, entwickelt. Ebenso wurde ein Kommunikations-und Datenprotokoll für das System aufgebaut. Ein Prototyp aus Android-App, dem Server-Modul (Arduino UNO-Board + Ethernet-Shield + XBee-Shield mit XBee-Modul) und mehreren Client-Modulen (Arduino UNO-Board + XBee-Shield mit XBee-Modul) weist die grundlegende Funktionalität nach.
Die TCP-Verbindung vom Android-Client zum Server-Modul läuft zuverlässig und schnell. Ein Request vom Android-Client wird unverzüglich und ohne Verzögerung bearbeitet. Somit zeigen sich Änderungen von Ausgangs-Pins sofort am Server-Modul, ausgelesene Werte erscheinen ebenso schnell in der Android-App. Die Response wird direkt an den Android-Client gesendet, der diese durch den AsyncTask parallel zur Bedienung der GUI (Main/UI-Thread) bearbeitet und enthaltene Daten entsprechend darstellt. Hauptsächlich sind AsyncTasks für kurze Berechnung oder Abläufe im Hintergrund vorgesehen, aber in diesem Fall arbeitet der AsyncTask zufriedenstellend als Client-Listener-Thread und kann somit genutzt werden. Dadurch ist es auch möglich, PMW-Pins (pseudo-analoge Ausgänge) in ‚Echtzeit' anzupassen, d. h. die Spannungsveränderung ist unverzüglich beim Verstellen des PWM-Wertes in der Android-App am Server-Modul feststellbar. Dabei werden viele Requests verschickt, die aber ausreichend schnell vom Server verarbeitet werden können. Alternativ könnte ein Android-Service (Hintergrund-Activity ohne GUI) anstelle des AsyncTasks eingesetzt werden, welcher speziell für solche Anwendungsfälle vorgesehen ist, was eine Steuerung im Hintergrund auch bei ausgeschalteter MainActivity, d. h. beendeter App, ermöglichen würde.

Die Steuerung und Überprüfung von Client-Modulen, die sich mit dem Server-Modul in einem XBee-Funk-Netzwerk befinden, ist vergleichbar zuverlässig. Jedoch ist bei einem Client Modul keine gleichwertige Übertragungs- und Reaktionsgeschwindigkeit möglich. Einen Flaschenhals für die Daten bildet die serielle Schnittstelle des Mikrocontrollers, über die das Server-Modul den Request an die Client-Module weiterleiten muss, ein zweiter ist die noch stärker einschränkende XBee-Funk-Übertragung. Das ist zum einen dem geschuldet, dass die Datenübertragungsrate und damit der Datendurchsatz einer seriellen Schnittstelle deutlich unter dem einer Ethernet-Schnittstelle bleibt, zum anderen dem, dass die Übertragungsrate der XBee-Module, welche die Daten der seriellen Schnittstelle per Funk übertragen, im Verhältnis sehr niedrig ist, selbst deutlich unter der Übertragungsrate von Bluetooth liegt. Hinzu kommt, dass die XBee-Funk-Module einen kleinen Eingangspuffer haben, der durch die schnelle Füllung über die serielle Schnittstelle zunächst geleert werden muss, was eine Verzögerung bewirkt. So kann es passieren, dass eine spürbare Verzögerung nach dem Verschicken einer Request an ein Client-Modul zu sehen ist, also die Response nicht unmittelbar eintrifft. In der Regel wird die Request durchgeführt, in seltenen Fällen kann es zu einem verlorenen Request (verworfen durch Time-Out) kommen, der daraufhin erneut versendet werden muss. Die Konsistenz der angezeigten Arduino-Pin-Daten bleibt in jedem Fall erhalten, da die Anzeige ausschließlich mit den in der Response erhaltenen Daten aktualisiert wird. Diese Daten werden durch das Auslesen der Arduino-Pins nach der Zustands- bzw. Werte-Änderung, initiiert durch den Request, gebildet. Das schnelle Hintereinander-Versenden von Requests an ein Client-Modul muss durch die Android-App reduziert werden, somit ist z.B. die erwähnte ‚direkte' Verstellung eines PWM-Pins wie beim Server-Modul deaktiviert und eine Änderung ist erst nach der Übernahme des eingestellten neuen Wertes sichtbar.
Das ‚gleichzeitige' Verwalten der Ethernet- und seriellen Verbindung durch das Server-Modul wird zuverlässig erfüllt. Eine wirkliche parallele Bearbeitung wie es in der Android-App durch Threads geschieht, ist bei Mikrocontrollern nicht möglich. Durch eine passende Struktur der Superloop ist die serielle Abarbeitung (TCP und XBee im Wechsel) zufriedenstellend schnell und verlässlich gelungen.
Über den Android-Client kann ein Client-Modul entfernt werden, was dazu führt, dass das Server-Modul das entsprechende Client-Modul aus der Client-Liste entfernt und wieder Platz für ein neues Client-Modul bietet. Somit kann ein Client-Modul im laufenden Betrieb entfernt, eventuell neu konfiguriert und/ oder umplatziert werden, woraufhin es sich durch einen Neustart wieder ohne Umstand ins System integrieren lässt. Ebenso ist auch ein neues vorkonfiguriertes Client-Modul jederzeit hinzufügbar. Gerade diese Funktion erweitert das System mit einem nützlichen Plug-and-play-Verhalten.

Die Android-App stellt die Arduino-Pin-Daten übersichtlich und kompakt dar und bietet durch Verwendung von ListViews unbegrenzten Platz für potenzielle Arduino-Module und Arduino-Pins.
Sie lässt sich benutzerfreundlich und intuitiv bedienen. Im Moment ist das Layout nur für Hochkant-Benutzung ausgelegt und auch fest eingestellt. Für Querformat müssten ein anderer Layout-Aufbau gewählt und eventuell weitere Activities angelegt werden. Insbesondere für Tablets würde sich ein Querformat eignen. Die Einstellungen der App könnten erweitert und in eine separate Activity ausgelagert werden, statt dafür das Activity-Menü zu verwenden. Bei Verbindungsproblemen zu einem der Arduino-Module wird derzeit keine Fehlermeldung in der Android-App angezeigt. Im Falle eines Fehlers wird dieser aber sicher abgefangen und es kommt nicht zu einem App-Absturz. Ebenso kommt es bei schnell hintereinander übertragenen Requests, deren Responses die ListViews aktualisieren, zu keinen Fehlern, da der Zugriff thread-save geregelt ist.
Derzeit werden einige Klassen nur als Datenstruktur für die grafische Oberfläche verwendet, was keine optimale Lösung darstellt, insbesondere müssen die Klassen ModuleLabel und ModulePin über eine Hauptklasse wie z.B. ArduinoModule in Verbindung stehen. Diese Lösung wurde vorerst gewählt, um bestehende List-Objekte wiederzuverwenden und den Umgang mit den ArrayAdaptern zu vereinfachen, die die ListViews verwalten. Um dies zu ändern, müsste die Struktur in einigen Bereichen angepasst werden.
Die Arduino-Module müssen vor dem Programmieren vorkonfiguriert werden, d. h. bestimmte Einstellungen wie z.B. Modul-Label (ID und Name) oder Pin-Modi der Arduino-Pins sind nicht im laufenden Betrieb aus der Android-App änderbar. Diese Lösung wurde aus dem Grund so gewählt, da sonst eine geänderte Konfiguration im Mikrocontroller des Arduino-Boards entweder im EEPROM oder Flash-Speicher abgelegt werden müsste. Diese Speicher-Funktion müsste jedesmal nach Änderung im Arduino-Programm durchgeführt werden, andernfalls wären nach Neustart die Daten verloren. Wenn die Daten überhaupt nicht gespeichert würden, müsste die benötigte Konfiguration bei jedem Neustart der Module vom Android-Client erneut übersendet werden. Eine Änderung der Pin-Modi von Arduino-Pins während der aktiven Nutzung eines Arduino-Moduls ist derzeit aus folgendem Grund nicht möglich: Die Änderung während des Betriebes (z. B. Wechsel von Ein- zu Ausgang) könnte zu Problemen führen, sogar zur Beschädigung von Arduino-Modulen oder angeschlossener Peripherie.
Der Aufbau des Datenprotokolls hat sich als gute Lösung erwiesen, da dies eine übersichtliche und Daten-reduzierte Möglichkeit ist, verschiedenste Requests kompakt auch über XBee-Funk zu übertragen. Außerdem eröffnet es flexible und vielseitige Nutzungsmöglichkeiten.

Die Wahl eines Eingangs- und Ausgangspuffers bei den Arduino-Modulen hat sich als bessere Variante gegenüber Arduino-Strings aufgezeigt, die in einer früheren Programm-Version genutzt wurden, da dadurch ein sicherer Programmablauf durch statische Speicherallokation gewährleistet ist. Darauf muss bei Mikrocontrollern der Arduino-Boards, die nur einen eingeschränkten Datenspeicher (SRAM) besitzen, stets geachtet werden.
Insgesamt lässt sich also resümieren, dass die gewählte Lösung durchaus nutzbare Ergebnisse liefert, auch wenn Optimierungs- und Verbesserungsmöglichkeiten bestehen.

3.2 Weiterführung

Das entwickelte System bietet sich als sinnvolle Möglichkeit zur Steuerung verschiedenster Endgeräte an. Zudem ist das Erfassen von Daten gezielt verteilter Sensoren eine wichtige erweiternde Funktion. Die leichte Zugänglichkeit des Systems und die überschaubare Benutzeroberfläche der App können es auch für Amateur-Anwender interessant machen. Die Verwendung der etablierten und verbreiteten Arduino-Boards, die bei diesen Anwendern beliebt sind, kann es zusätzlich attraktiver machen im Vergleich zu bestehenden Lösungen.
Für den Einsatz wären als Aktoren zum einen einfache Funkschalter und -dimmer für Licht und andere Elektro-Endgeräte denkbar, darüber hinaus aber auch komplexere Komponenten wie Lichteffekt-Steuermodule, Servo-Motoren oder Heizthermostate. Sensor-Module sind z. B. vorstellbar im Bereich Helligkeit, Temperatur und Feuchtigkeit. Die möglichen Anwendungsfälle sind aber durchaus vielfältiger.
Das bereits entwickelte Toolkit (System- und Softwarekomponenten) bietet weitere flexible zukünftige Einsatzmöglichkeiten. Entweder wird es in einer fest vorgegeben App genutzt, die im Vergleich zur aktuell bestehenden App erweitert ist, auch sind aber neue Apps denkbar, welche die verwendete Software-Struktur samt Datenprotokoll verwendet. Durch die Steuermöglichkeiten mehrerer unterschiedlicher Module wäre es auch sinnvoll und nützlich, sogenannte Funktionsprofile in der App anlegen zu können. Dabei werden bestimmte vom Anwender festgelegte Module und deren Aufgaben zu Profilen zusammengefasst, womit es vereinfacht wird, mehrere Module gleichzeitig und automatisiert zu steuern. Dies kann entweder manuell, in Abhängigkeit von einem einprogrammierten Zeitpunkt, oder auch durch Module mit angeschlossenen Sensoren über die Smartphone-App gesteuert, durchführbar sein. Es wären in der App auch Zeitpläne vorstellbar, in der die Aktoren und

Sensoren aller verwendeten Module beliebig in Verbindung gebracht werden können, eine Art leistungsfähige kontextbezogene Zeitschaltuhr.
Für das Ansprechen der Endgeräte wären ausschließlich noch Steuereinheiten nötig, wie z.B. (elektronische) Relais, die an die Arduino-Module direkt angeschlossen werden.
Die Fernbedienungs-App übernimmt dabei die gesamte Steuerungsverwaltung der verteilten Module, durch eine alternative Verwendung eines Android-Service als App-Komponente könnten alle Steuer- und Erfassungsfunktionen unabhängig der Bedieneroberfläche im Hintergrund ausgeführt werden. Dies wäre als Erweiterung der App denkbar.
Auch die verwendete Hardware betreffend, sind Modifikationen/Erweiterungen denkbar. Das eingesetzte Ethernet-Shield des Server-Moduls könnte durch ein WLAN-Shield ersetzt werden. Somit könnten beim Server-Modul durch den kabellosen Einsatz Client-Funktionalitäten flexibler nutzbar gemacht werden.
Der Einsatz von XBee-Modulen bei sämtlichen Arduino-Modulen kann außerdem überdacht werden. Die Übertragungsrate ist im Vergleich zu Ethernet deutlich geringer, was zu einer schlechteren Benutzererfahrung mit der App führen kann. Alternativ könnten z. B. Bluetooth-Module auf den XBee-Shields oder komplette Bluetooth-Shields verwendet werden, welcher eine höhere Übertragungsrate bieten.
Wie auch beim Server-Modul wären WLAN-Shields bei allen Client-Modulen einsetzbar, die es in verschiedenen Ausführungen gibt. Durch WLAN wäre der Zugriff auf alle Module einheitlich in Geschwindigkeit und Reaktionsschnelle. Nachteilig wären der höhere Hardware- und Energieaufwand, zudem die höheren Kosten. XBee-Module sind im Vergleich kostengünstiger und sehr energiesparend.
Für die Arbeit wurden hauptsächlich Arduino UNO-Boards eingesetzt, jedoch ist die Verwendung von leistungsfähigeren Arduino-Boards zweckmäßig, wenn diese, wie das testweise genutzte Arduino MEGA-Board, mehr nutzbare Ein- und Ausgänge bieten. Zu beachten wäre hier, dass wenn bei einem Client-Modul anstelle des Arduino UNO ein Arduino MEGA eingesetzt werden soll, der Arduino-UNO des Server-Moduls ebenfalls durch einen Arduino MEGA ersetzt werden muss, um die dann mögliche erhöhte Datenmenge verarbeiten zu können. Das bedeutet, die verwendeten Arduino-Boards müssen aufeinander abgestimmt sein. Die Arduino-Programme müssten dafür nur geringfügig angepasst werden, um auf verschiedenen Arduino-Boards lauffähig zu sein. Dies wird durch Arduinos plattformunabhängigen Code erreicht.
Im Vergleich zum Arduino UNO bietet der Arduino MEGA weitere serielle Hardware-Schnittstellen, die zu einer Verbesserung der Kommunikation zwischen den Arduino-Modulen führen können. Dadurch könnte das Server-Modul beispielsweise parallel

mehrere Nachrichten von Client-Modulen annehmen, ohne dass ein gleichzeitiges Senden zu Interferenzen führen würde.
Zusammenfassend lässt sich also feststellen, dass die Grenzen des Systems keinesfalls ausgeschöpft sind. Verschiedene Modifikationen und Erweiterungen sind denkbar, welche je nach gewünschtem Einsatzgebiet des Systems eingebracht werden können.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

- Günter Schmitt - Mikrocomputertechnik mit Controllern der Atmel AVR-RISC-Familie - 3. Auflage 2007 - Kapitel 1, 3, 4
- Michael Magolis – Arduino Kochbuch - 1. Auflage 2012 - Kapitel 2, 4, 15
- Video-Lehrgang: Apps entwickeln für Android 5 - Das umfassende Training
- https://cdn-learn.adafruit.com/downloads/pdf/memories-of-an-arduino.pdf (letzter Abruf 04.04.2016)
- https://developer.android.com/guide/index.html (letzter Abruf 11.05.2016)
- http://www.diffen.com/difference/TCP_vs_UDP (letzter Abruf 10.06.2016)
- https://www.arduino.cc/ (letzter Abruf 13.06.2016)
- http://www.digi.com/de/ (letzter Abruf 16.06.2016)
- http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_21_004.htm (letzter Abruf 15.04.2016)
- http://www.cplusplus.com/reference/ (letzter Abruf 08.05.2016)
- http://www.programmierenlernenhq.de/tutorial-android-activities-und-intents/ (letzter Abruf 06.05.2016)
- http://examples.digi.com/get-started/basic-xbee-zb-zigbee-chat/ (letzter Abruf 11.04.06.2016)
- http://tutorial.cytron.com.my/2012/03/08/xbee-series-2-point-to-point-communication/ (letzter Abruf 10.04.2016)
- https://www.sparkfun.com/pages/xbee_guide (letzter Abruf 15.06.2016)
- http://mcukits.com/2009/04/06/arduino-ethernet-shield-mega-hack/ (letzter Abruf 29.05.2016)

Hinweis: Die angegebenen Quellen im Literaturverzeichnis wurden zur Vorbereitung bzw. während der Arbeit genutzt, ein direkter Bezug auf diese findet innerhalb der Arbeit keine Verwendung.

Anhang A – Inhalt der CD

- **Bachelorarbeit – Roitzsch.pdf**
- **Arduino-Programme**
  - **Arduino UNO**
    - **ServerModule_ByteStream**
      - **ServerModule_ByteStream_config.h**
      - **ServerModule_ByteStream.ino**
    - **ServerModule**
      - **ServerModule_config.h**
      - **ServerModule.ino**
    - **ClientModule_ByteStream**
      - **ClientModule_ByteStream_config.h**
      - **ClientModule_ByteStream.ino**
    - **ClientModule**
      - **ClientModule_config.h**
      - **ClientModule.ino**
  - **Arduino MEGA**
    - **ServerModule_MEGA**
      - **ServerModule_MEGA_config.h**
      - **ServerModule_MEGA.ino**
    - **ClientModule_MEGA**
      - **ClientModule_MEGA_config.h**
      - **ClientModule_MEGA.ino**
- **Android-App**
  - **Java-Klassen**
    - **arduinocontrollerclient**
      - **TCPClient.java**
      - **RequestSender.java**
      - **ModulePinArrayAdapter.java**
      - **ModulePin.java**
      - **ModuleLabelArrayAdapter.java**
      - **ModuleLabel.java**
      - **MessageLogArrayAdapter.java**
      - **MainActivity.java**
  - **APKs**
    - **app-release.apk**
    - **app-debug-unaligned.apk**
    - **app-debug.apk**
  - **Android Studio-Project**
    - **AndroidControllerClient**
      - ***.***

Danksagung

Auf diesem Wege möchte ich mich bei einigen Personen bedanken:

- Prof. Dr. Birgit Wendholt für ihre Geduld und das interessante Thema der Arbeit, welches meine Begeisterung für die Arduino-Welt auch privat geweckt hat
- Frau Iris Khalilian für ihre moralische Unterstützung und ihre Beharrlichkeit
- Meiner Partnerin Anke Roga für den Glauben an mich, Verbesserungsvorschläge und Korrekturlesen
- Charly & Muttern, ohne die das Studium nicht möglich gewesen wäre

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den _____ _____