ARINA
Arduino Remote Infrared Network Adapter

René Neff und Thomas Trimborn

16. Oktober 2013

Zusammenfassung

Diese Ausarbeitung beschreibt den Entwurf und die Konstruktion von ARINA, eines Adapters zur Weiterleitung von Infrarotsignalen. Ein Protokoll zum Transport von Infrarotsignalen wird definiert und seine Verwendung im Softwareentwurf des Projekts dargelegt. Die Funktionalität der Hard- und Sorftwarekonstruktion wird im Rahmen einer Evaluation durchgeführter Tests bestätigt. Abschließend werden im Rahmen eines Ausblicks zukünftige mögliche Erweiterungen und die Verwendung in Smart-Home-Anwendungsmöglichkeiten vorgestellt.

Inhaltsverzeichnis

1 Einleitung

Im Projekt *Arduino Remote Infrared Network Adapter* kurz ARINA entstand eine Lösung zum Bedienen von infrarotgesteuerten Geräten ohne direkten Sichtkontakt. Entworfen und gebaut wurden zwei kompakte ethernetbasierte Adapter, die die Steuerung von einem oder mehreren entfernten, stationären und infrarotansteuerbaren Multimediageräten ermöglichen. Beide Adapter sind baugleich und werden mit identischem Programmcode betrieben. Per SD-Karte wird eine komfortabel editierbar Konfigurationsdatei durch den Benutzer eingebracht. Beide Adapter können sowohl als Sender als auch als Empfänger genutzt werden und sind in der Lage, neben vielen weit verbreiteten IR-Protokollen [9], auch nicht decodierte IR-Daten (RAW), an die entsprechende Gegenstelle zu übermitteln und dort auszugeben. Voraussetzung für diese Funktionalität ist die technische Umsetzung einer Weiterleitung von Infrarotsignalen mittels Kapselung des jeweiligen Signals und kontrollierter Weiterleitung und Verarbeitung am Zielort. In der theoretischen Ausarbeitung [9] wurde festgehalten, dass eine sparsame, preiswerte und leicht zu bauende Lösung Ziel dieses Projektes sein soll. Ein ARINA nutzt als Grundlage ein Arduino Mega 2560 mit Ethernet-Shield, neben den benötigten IR-Sende- und Empfangs-Dioden geben weitere LEDs den aktuellen Status an. In dieser Ausarbeitung wird die technische Umsetzung der Hardwarekomponenten erläutert, ebenso werden das zugrundeliegende Programm und das entworfene Protokoll vorgestellt. Es erfolgt nach einer Vorstellung jeweils entwicklungsbezogener Probleme zur Hard- und Software eine Evaluation der erreichten Ziele. Die schon in der Entwicklung bedachte Möglichkeit zur Erweiterbarkeit des ARINA bildet den Abschluss mit der Vorstellung von zusätzlichen Features.

2 Hardware

In der theoretischen Ausarbeitung wurde sich für die Entwicklung eines, auf der Mikrokontroller-Plattform Arduino Mega 2560 basierenden Adapters festgelegt, diese bildet die Hardwaregrundlage für dieses Projekt. Ein Ethernet Shield stellt die benötigte Konnektivität per Ethernet bereit und bietet zusätzlich die Möglichkeit Konfigurationsdaten per SD-Karte einspielen zu können. Sowohl LEDs als auch IR-Empfangs- und Sende-Dioden werden mit entsprechenden Widerständen auf einer Steckplatine für Statusmeldungen bzw. IR-Empfang und -Versand genutzt. Der Aufbau dieser Komponenten und die dabei angetroffenen Probleme werden im folgenden Abschnitt näher erläutert.

2.1 Beschreibung der Komponenten

Grundlage des Adapters bildet ein Arduino Mega 2560, dessen Eigenschaften in der theoretischen Ausarbeitung bereits beschrieben wurden.

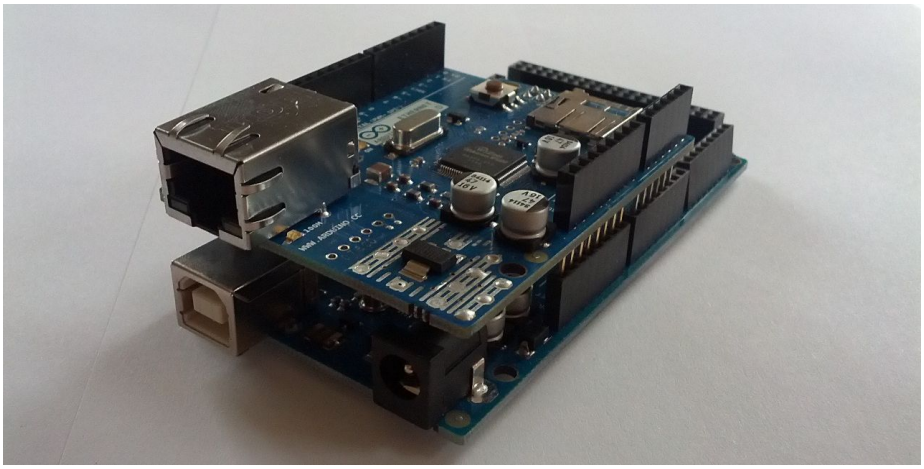

Abbildung 1: Arduino Mega 2560 mit aufgestecktem Arduino Ethernet Shield

Zur Ausgabe von Infrarotsignalen auf jeder Einheit wird ein IR-Emitter Harvatek HE3-290AC in 5mm Bauweise genutzt, der Infrarotstrahlung mit einer Wellenlänge von 940 nm ausgeben kann.

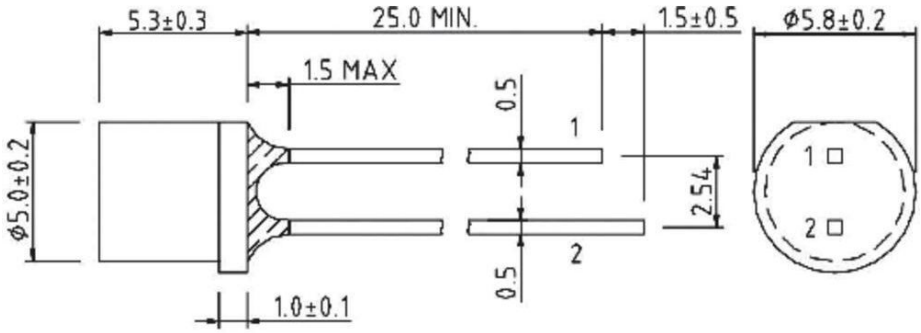

Abbildung 2: IR-Emitter Harvatek HE3-290AC; Gehäuseart 5 mm; Wellen-Länge: 940 nm. Quelle: www.conrad.de

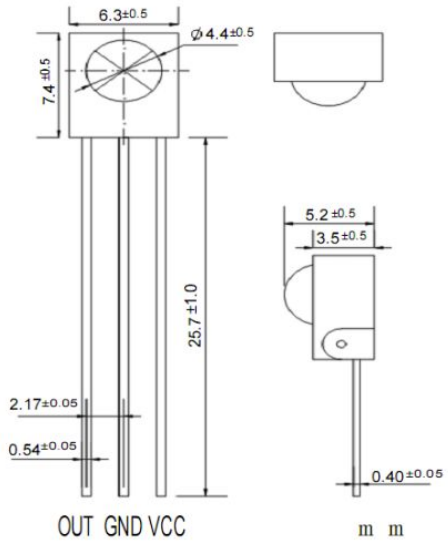

Abbildung 3: IR-Empfänger-Modul OS-OPTO OS-1638; Wellen-Länge: 940 nm. Quelle: www.conrad.de

Der Empfängerbaustein ist ein IR-Empfänger-Modul OS-OPTO OS-1638, das Wellenlängen von 940 nm mit einer Modulationsfrequenz von 38 kHz empfangen, verarbeiten und ausgeben kann. Es arbeitet bei einer Betriebsspannung von 2,7 bis 5,5 Volt, wodurch es direkt an einen Digitaleingang eines Arduino angeschlossen werden kann und vom 5V Port des gleichen Geräts mit Strom versorgt werden kann.

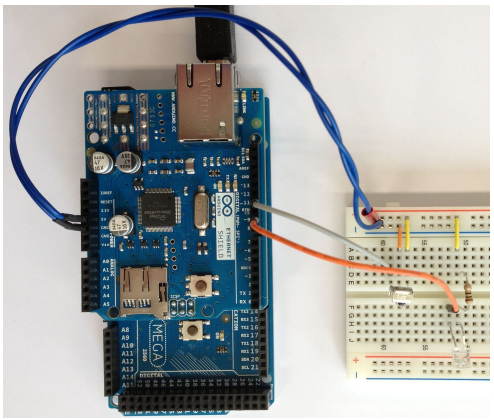

Abbildung 4: Draufsicht auf einen Aufbau in Minimalkonfiguration

Außerdem werden für den Aufbau pro ARINA eine Steckplatine, Steckbrücken, Kohlewiderstände, LED und flexible Steckbrücken genutzt. Die Steckplatine dient dabei der Verbindung der einzelnen Komponenten und ist in Abbildung 4 rechts zu erkennen. Die Steckbrücken verbinden dabei auf dem Board die einzelnen Komponenten miteinander. Während die flexiblen Steckbrücken die Bauteile mit elektrischer Energie von den Ausgängen des Arduino und mit den digitalen Aus- und Eingängen 3 und 6-9 verbinden, um alle Bauteile einzeln ansteuern zu können. Da die verwendeten LEDs nahezu keinen elektrischen Widerstand aufweisen und ihre Betriebsspannung unterhalb der vom Arduino gelieferten 5 Volt liegt, werden Kohlewiderstände genutzt, um Beschädigungen an den Komponenten zu verhindern. Die LEDs, die auf Abbildung 5 zu erkennen sind, zeigen an, ob eine Verbindung besteht oder nicht. Die grüne und rote LED geben den Zustand der Verbindung der beiden ARINA an, während Blinken der blauen LED signalisiert, dass gerade Daten verarbeitet werden.

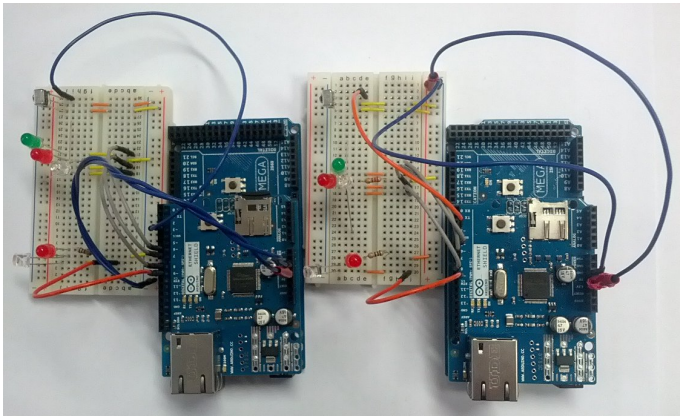

Abbildung 5: Aufbau von zwei fertigen ARINA

In dem elektrischen Schaltplan in Abbildung 6 lässt sich erkennen, dass der IR-Emitter, der ein PWM Signal ausgeben kann, an Pin 9 angeschlossen ist sowie der IR-Reciever an Pin 3 und die drei zusätzlichen LEDs an die Pins 7, 8 und 9.

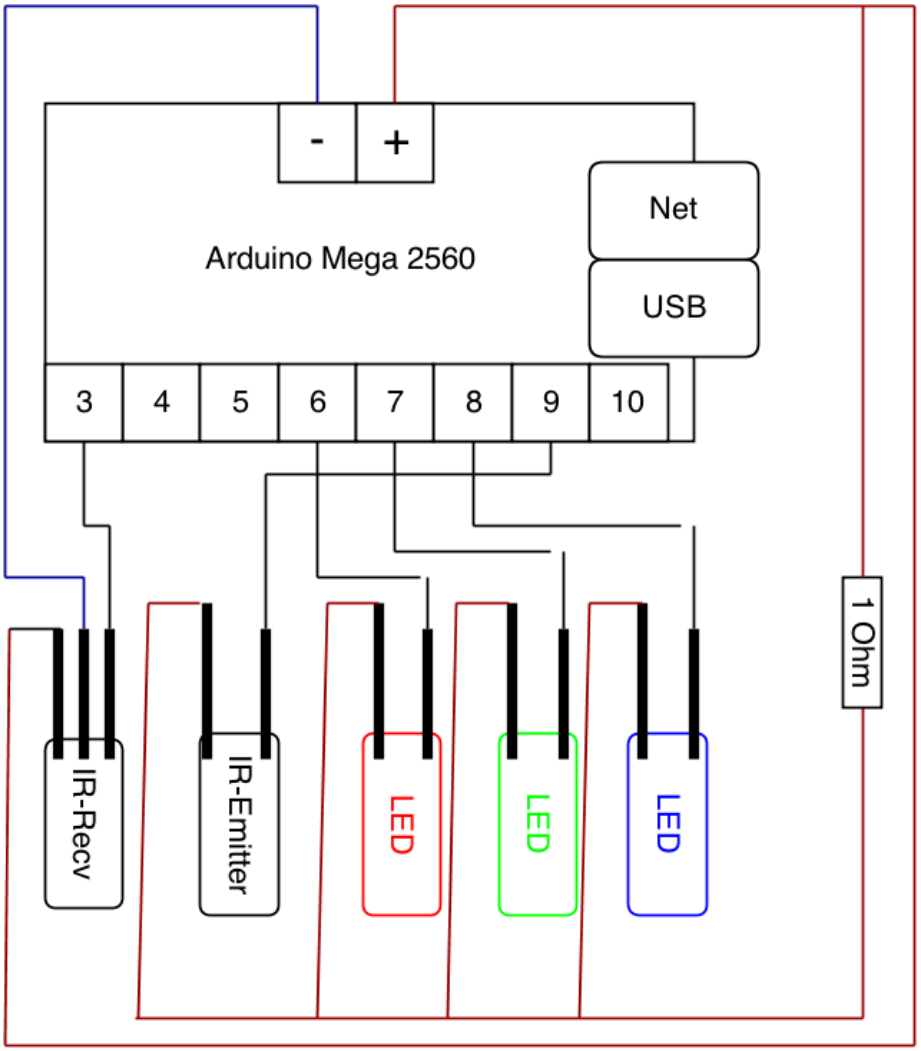

Abbildung 6: elektrischer Schaltplan eines ARINA

2.2 Besonderheiten und Probleme

Bei ersten Versuchen mit einer Infrarot-LED war auffallend, wie kurz die Reichweite der ausgesendeten Signale war. Nach Recherche [1] und einigen Versuchen stellten wir zudem fest, dass die digitalen I/O Pins des verwendeten Arduinos sowohl als Ein- als auch als Ausgänge dienen können, ohne etwas an der Konfiguration des Arduinos zu ändern. So ist es abhängig davon, ob man als Gegenpol zum Pin des Arduinos den positiven oder den negativen Pol der Energieversorgung nutzt, in welcher Richtung der Strom fließt. Nutzt man den digitalen I/O Pin des Arduinos als Pluspol, so ist die Reichweite der IR-Signale sehr begrenzt. Nutzt man ihn jedoch als Minuspol und den 5 oder den 12 Volt Pin als Pluspol, so ist die Reichweite um ein Vielfaches verbessert. Das liegt daran, dass die digitalen I/O Pins lediglich einen Strom von 40mA liefern können, [1] was gerade dem Verbrauch einer der von uns verwendeten LEDs entspricht, jedoch nicht ausreicht um den IR-Emitter zu versorgen, der laut Datenblatt 100mA benötigt.

Eine weitere Besonderheit ist die Pinbelegung des Arduino Mega 2560, der von uns verwendet wird, im Vergleich zum normalen Arduino Uno und seinen Varianten [7], was dazu führt, dass einige Bibliotheken nicht auf Anhieb mit dem Mega 2560 funktionierten. Aufgefallen ist diese Besonderheit mit der verwendeten IRRemote Bibliothek 3.2, da der Arduino den Ausgang des Timer 2, den die Infrarotbibliothek verwendet um ein PWM Signal zu erzeugen, nicht wie beim Uno auf Pin 3 liegen hat, sondern auf Pin 9. Das Timersignal wird genutzt, um das pulsweitenmodulierte Ausgabesignal zu erzeugen, mit dem die Infrarotsignale ausgegeben werden.

Ein Phänomen, das immer wieder auftrat, ist die Empfindlichkeit auf elektromagnetische Felder, wie sie der menschliche Körper ausstrahlt, bei dem Versuch einen Taster zu integrieren. Der Arduino hat auch dann gemeldet der Taster sei gedrückt, wenn man mit der Hand in der Nähe der Kabelbrücke war, die den Taster mit dem digitalen Eingang des Arduinos verband.

Entgegen der bekannten Praxis, dass Geräte mit einer Ethernetschnittstelle eine voreingestellte MAC-Adresse haben, ist es bei den Ethernet Shields für Arduinos immer notwendig, beim Initialisieren eine beliebige Adresse mit anzugeben. So lässt sich ein Arduino in einem Netzwerk nie eindeutig identifizieren, da sich die Adresse ja jederzeit ändern lässt.

3 Software

Um auch für zukünftige Weiterentwicklung und Erweiterungen eine Grundlage bieten zu können, wurden bereits in der theoretischen Ausarbeitung [10] strukturelle Grundsätze festgelegt. Zudem ermöglicht eine klare Kapselung benötigter Funktionalitäten in Klassen beste Wiederverwertbarkeit sowie einfaches Codeverständnis. Unterstützend dazu wurden eindeutige und funktionsbezeichnende Variablennamen gewählt, die ergänzend zur im Doxygen-Format erstellten Dokumentation, einem Leser mit lediglich grundlegendem Programmierverständnis, einen einfachen Einstieg in den Programmablauf ermöglichen. Wie in der theoretischen Ausarbeitung [9] erläutert, war gemäß der Aufgabenstellung ein mit begrenzten Ressourcen im Hinblick auf Platz und Rechenleistung realisierbares Gerät gewünscht, welches möglichst in Echtzeit arbeitet. Um gewonnene Informationen der verschiedenen, darauf spezialisierten Klassen weiterverarbeiten zu können, aber dennoch einen möglichst geringen Overhead an zusätzlichem Speicherbedarf zu haben, wird - so weit möglich - mittels "Call-by-Reference" Funktionsaufrufen gearbeitet. Damit des Weiteren eine möglichst hohe Abdeckung bei vielen unterschiedlichen Geräten, die diese Infrarot-Steuerung nutzen sollen, erreicht werden kann, wurde eine schon existierende Codedatenbank (Siehe 3.2) eingebunden. Eine schon existierende Code-Bibliothek, die beispielsweise das weitverbeitete NEC-Protokoll [11] decodiert, ermöglicht zudem die Menge der zu verarbeitenden sowie zu übertragenden Daten massiv zu reduzieren. Um jedoch nicht nur auf die erkannten Codierungen beschränkt zu sein, ist es ebenso möglich, nicht durch die Bibliothek decodierte Signale mittels eines Fallbacks auf eine RAW-Datenübertragung weiterzuleiten. Nach dem erarbeiteten Kenntnisstand existiert bislang kein standardisiertes Übertragungsprotokoll, welches decodierte und nicht-decodierte IR-Signale von Fernbedienungen per Ethernet überträgt.

Eine Definition des eigens für den ARINA entworfenen Übertragungsprotokolls erfolgt im anschließenden Abschnitt. Weiter werden die Funktionen der erstellten Klassen vorgestellt und in Grundzügen erklärt. Zur Programmierung wurde die auf der Hauptseite des Arduino Projekts frei verfügbare Arduino IDE in der Version 1.0.5 verwendet. Weitere verwendete Bibliotheken sind die direkt mitgelieferte Ethernet Bibliothek sowie die ebenfalls frei verfügbare Multi-Protokoll-IR-Fernbedienungs-Bibliothek von Ken Shirriff (verfügbar über seinen Blog [4]). Diese werden in 3.2 ebenfalls kurz in ihrer Funktionalität und Verwendung in ARINA vorgestellt.

3.1 Entwurf

Nachfolgend wird einleitend das eigens definierte Protokoll zur Übertragung von IR-Daten beschrieben. Anschließend wird in Abschnitt 3.1.2 gezeigt, wie der Programmablauf beim Empfangen und Weiterleiten von IR-Signalen abläuft. In 3.1.3 wird der Softwareentwurf vorgestellt, welcher im Kapitel 3.3 durch eine Implementierung umgesetzt wurde.

3.1.1 Protokoll

Die verschiedenen Anforderungen an ein Protokoll zur gekapselten Übertragung von IR-Signalen ergeben sich aus unterschiedlichen Teilbereichen des entworfenen Aufbaus. Um eine schnelle Übermittlung der Daten sowie ein geringes Datenvolumen zu erreichen und um auf unnötige Konvertierungen verzichten zu können, wurde von einer Entwicklung des Protokolls in einem Datenformat in einer plattform- und implementationsunabhängigen Auszeichnungssprache, wie zum Beispiel XML, abgesehen. Der gewählte Ansatz war der Entwurf eines eigenen kompakten Protokolls.

Wie in der theoretischen Ausarbeitung [11] vorgestellt, gibt es verschiedene Möglichkeiten, um ein IR-Signal weiterzuleiten. Das Interpretieren des empfangenen IR-Signals in den entsprechenden Befehl (z.B. “Play”) würde einen immensen Aufwand zur Erstellung einer umfassenden Befehlsdatenbank bedeuten, insbesondere würden neue Geräte und Typen nur durch beständige Pflege der Befehlsdatenbank unterstützt. Der größte Vorteil wäre eine sehr kompakte Übertragung zur Weiterleitung eines einzelnen Befehls.

Einen universalen Ansatz stellt das Decodieren des empfangenen Signals in den jeweiligen einzelnen Protokolltyp und Befehlscode dar. Hierbei ist die übertragene Datenmenge ebenfalls gering und nur ein überschaubares Maß an Decodierfunktionalität nötig, mit schon sieben erkannten Protokollen wird ein Gros der auf dem Markt befindlichen Geräte abgedeckt [11]. Für nicht decodierte IR-Signale wird eine Übertragung des reinen IR-Signals (RAW) unterstützt. Dies stellt mit einem höheren Datenaufkommen die universelle Funktionalität des ARINA zum Übertragen von IR-Signalen im entsprechenden Frequenzbereich sicher. Die Übertragung eines einzelnen Befehlscode mit dem codierenden IR-Protokoll stellt somit einen guten Kompromiss zwischen aufzubringendem Aufwand zum Erkennen und der Übertragungsmenge dar und bietet Zukunftssicherheit. Die Verwendung des Arduino Ethernet Shields und die damit bereitgestellte Ethernet Bibliothek 3.2 erlaubt eine Übertragung von einzelnen Bytes, andere übertragene Datentypen, wie zum Beispiel ein 32 Bit long-Wert, werden implizit in vier Byte-Werte konvertiert. Zur direkten Übertragung bietet sich daher ein in einzelne Bytes unterteiltes Übertragungsformat an.

Die verwendete Codebibliothek, die in 3.2 beschrieben ist, kann die am häufigsten verwendeten [11] IR-Protokolle decodieren. In der später vorgestellten Evaluation war dies der häufigst zu beobachtende Fall. (Abbildung 11) Nicht decodierte IR-Signale sollen jedoch ebenfalls übertragen werden, so müssen die erfassten RAW-Daten gegebenenfalls auch verarbeitet werden. Das decodierte Protokoll und der jeweilige Befehlscode, beziehungsweise die entsprechenden RAW-Daten, müssen jedoch noch in ein für den Versand per Netzwerk (siehe 3.3.5) passendes, nur aus Bytes zusammengesetztes, Format gebracht werden. Ein solches Format, zur Kapselung von IR-Signalen zur Übertragung mittels

TCP/IP, konnte bei der vorhergehenden Recherche zur theoretischen Ausarbeitung [9] jedoch nicht gefunden werden. Ein, den genannten Anforderungen, konformes Format wird durch unser entworfenes Protokoll definiert und im Folgenden erklärt. Um die durch die in 3.2 beschriebene IR-Codebibliothek er-

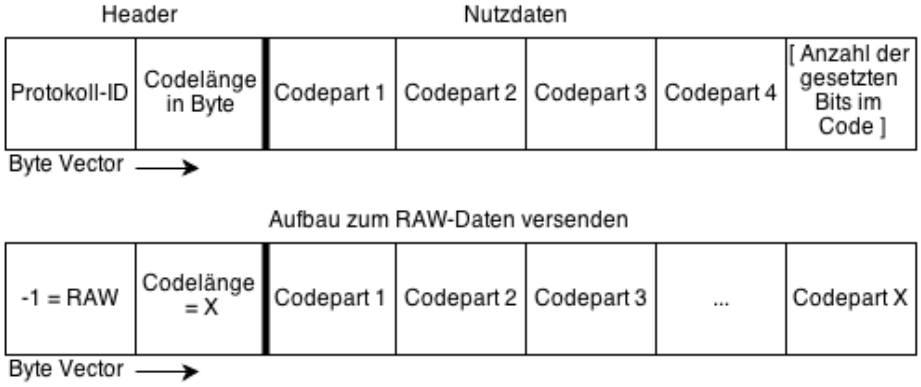

Abbildung 7: Allgemeiner Protokollaufbau und ein Beispiel für die Übertragung von RAW-Daten

kannten Signale übertragen zu können, wird das von der Bibliothek erkannte Protokoll einleitend zum Aufbau des zu übertragenden Bytevektors im Header entsprechend gesetzt. Die Codelänge gibt die Anzahl der folgenden Bytefelder an, und ermöglicht es einen Abschluss des übertragenen Bytevektors festzustellen. Im dargestellten Fall eines Befehlscodes, repräsentiert durch einen 32 Bit long-Wert, wurde dieser long-Wert in vier Byte unterteilt und dem Nutzdatenbereich des Bytevektors angefügt. Das zuletzt gesetzte Vektorfeld gibt die Anzahl der tatsächlich gesetzten Bits im zu rekonstruierenden long-Wert an; diese Angabe ist für eine korrekte Rekonstruktion des decodierten Befehlscodes notwendig. Wie in Abbildung 8 schematisch dargestellt, unterscheidet sich der Aufbau des Nutzdatenbereiches bei Übertragung der IR-Signale durch RAW-Daten. (Abbildung 7) Einleitend wird durch das Protokoll-Flag "-1" die folgende Übertragung als RAW deklariert, eine Längenangabe im zweiten Vektorfeld hilft beim Rekonstruieren der auszugebenden IR-Signale. Die einzelnen IR-Impulslängen werden mittels eines Bytewertes codiert und dem Vektor angehängt, dieser kann dann abschließend zum empfangenden ARINA übertragen werden.

3.1.2 Ablaufdiagramm

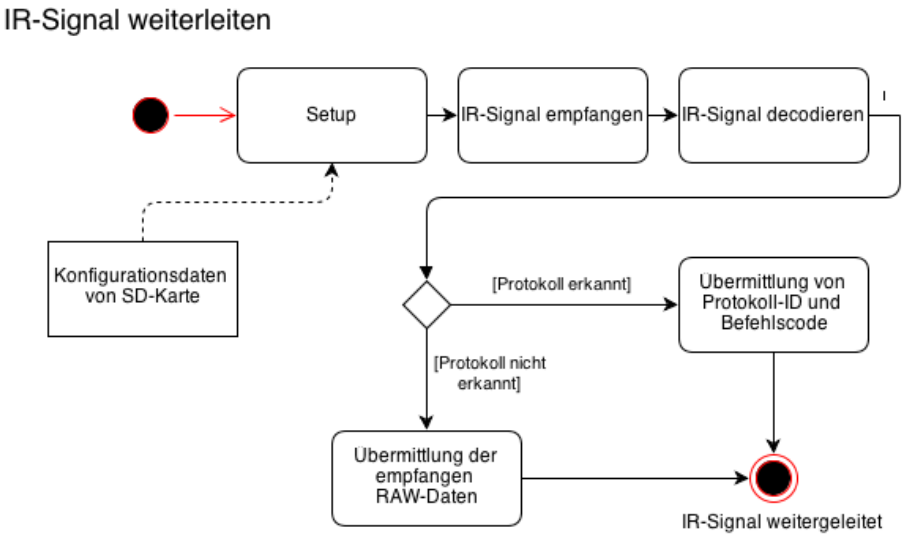

Abbildung 8: Ablaufdiagramm: Weiterleitung von IR-Signalen mittels ARINA

3.1.3 Klassendiagramm

Eine Übersicht über die erstellten Klassen und die für den Programmablauf zuständige arina.ino schafft das Klassendiagramm in Abbildung 9. Weitere, die einzelnen Klassen behandelnden, Erläuterungen finden sich im folgenden Abschnitt.

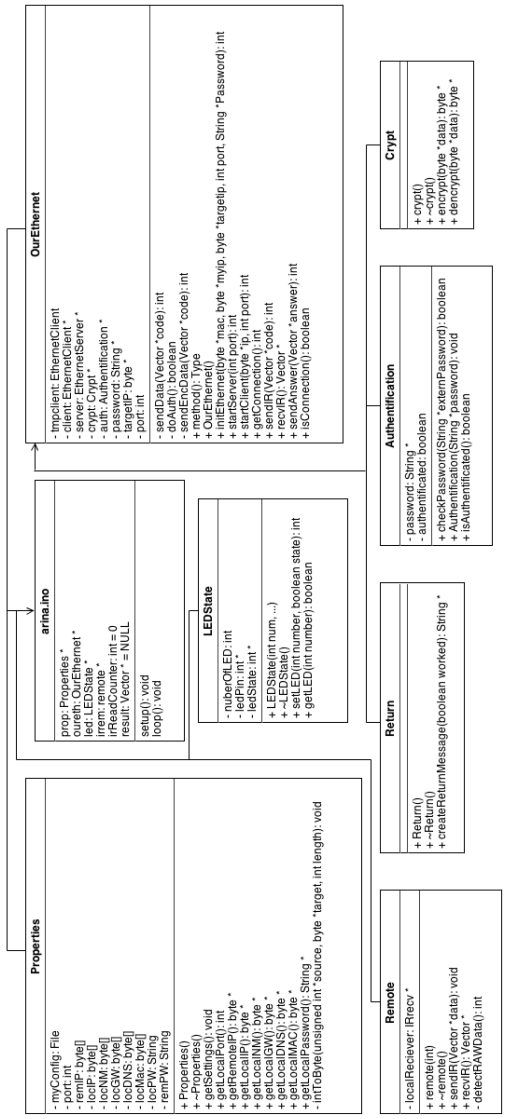

Abbildung 9: Das Klassendiagramm beinhaltet alle erstellten Klassen sowie die für den Ablauf zuständige und Arduino-charakteristische *.ino Datei: arina.ino

3.2 Verwendete Bibliotheken

Zum Decodieren der empfangenen IR-Signale wird eine von Ken Shirriff geschriebene Multi-Protokoll-IR-Fernbedienungs-Bibliothek [8] genutzt. In der aktuellen Version 1.0 der Bibliothek können IR-Signale der Protokolle NEC, Sony, Philips RC5, Philips RC 6, DISH, Sharp, JVC und Panasonic en- und decodiert werden. Die Bibliothek liefert entsprechend den Protokoll-Typ und Befelscode des jeweils empfangenen Signals als Long-Wert zurück, indem Bit für Bit der als IR-Signal empfangenen Daten in den Speicherbereich des 32bit langen Long-Wertes geschrieben wird. Sharp und Panasonic bilden hier eine Ausnahme und liefern protokollspezifische, zusätzliche Informationen.

Bei durchgeführten Tests zeigte sich, dass sich eine höhere Protokollerkennungsrate einstellte, wenn die von Shirriff alle 50 Microsekunden aufgerufene Decodierroutine schon nach jeweils 15 Microsekunden aufgerufen wird. Eine mögliche Erklärung könnten baulich unterschiedliche IR-Empfangs-Dioden, IR-Sende-Dioden oder die Verwendung eines Arduino Uno an Stelle des verwendeten Arduino Mega 2560 sein.

Die Bibliothek zur Ansteuerung der Netzwerkschnittstelle auf Ethernet Shields „Ethernet.h“ ist einfach zu einzubinden, und funktionierte bei allen Versuchen einwandfrei. Ein Problem ist jedoch, dass unabhängig vom versendeten Datentyp immer Daten vom Typ „Byte“ empfangen werden. Als Folge ist der Empfänger gezwungen, vorher zu wissen welche Daten empfangen werden sollen, um beispielsweise aus 4 Bytes wieder ein long zusammenzubauen, wie es bei der Verwendung des in 3.1.1 beschriebenen Protokolls notwendig ist und deshalb von der Klasse „Remote.h“ (3.3.4) bei jedem empfangenen Datenstrom durchgeführt wird, wenn dieser keine RAW-Daten enthält.

3.3 Implementierung

In diesem Abschnitt wird die konkrete Implementierung der im Projekt ARINA verwendeten Klassen und ihre Funktion erklärt. Einleitend wird knapp auf die Besonderheiten eingegangen, welche beim Programmieren zu beachten sind und sich aus der Arduino-Umgebung ergeben.

3.3.1 Programmablauf (arina.ino)

Für Arduino wird eine Datei benötigt, die den Programmablauf beschreibt. Dazu muss sie die zwei folgenden Funktionen enthalten:

```
void setup(){}
void loop(){}
```

Der Precompiler setzt diese dann in eine main-Funktion ein, die der C++-Compiler verarbeiten kann, in der zuerst einmalig die „setup“-Funktion aufgerufen wird um dann wie folgt die „loop“-Funktion auszuführen:

```
while(true){
        loop();
}
```

In dieser Datei definieren wir global je einen Pointer auf ein Objekt, das eine der Klassen „Properties“ (3.3.2), „OurEthernet“ (3.3.5), „Remote“ (3.3.4) und „LED“ (3.3.6) instanziert.

In der „setup“ wird dann zuerst die „Properties“ initialisiert und veranlasst den Inhalt der Datei „settings.txt“ auf der SD-Karte zu lesen und zu verarbeiten. Anschließend wird die OurEthernet (3.3.5) initialisiert und veranlasst einen listen-Port zu öffnen. Diese Reihenfolge muss auch zwingend eingehalten werden, wegen der im Abschnitt 3.4 beschriebenen Eigenarten der von uns verwendeten kombinierten Ethernet/SD-Card Shields. Zuletzt initialisieren wir noch die „Remote“ (3.3.4), um Infrarotsignale empfangen und senden zu können sowie die „LED“ (3.3.6) zur Ansteuerung der Signal-LEDs.

Dann wird in der „loop“ Funktion mehrfach überprüft, ob Infrarotdaten vom IR-Reciever empfangen wurden. Wenn welche empfangen wurden, wird versucht, sie per Ethernet an die Gegenstelle zu senden. Anschließend wird einmal der Ethernet Buffer abgefragt, ob dort Daten eingegangen sind. Wenn ja, werden sie über die IR-Diode ausgegeben. Es muss mehrfach der IR-Reciever Baustein abgefragt werden, bevor einmal die Ethernetschnittstelle nach neuen Daten abgefragt wird, da der IR-Reciever bauartbedingt über keinen Puffer verfügt, der Signale vorhalten würde. So wird jedes Signal nicht empfangen, das in der Zeit am Reciever ankommt, in der die Anfrage nach Daten an der Ethernetschnittstelle läuft. Da diese über einen 16 kB großen Puffer [12] verfügt ist es unproblematisch, diesen seltener abzufragen als den IR-Reciever. Bei unseren Versuchen hat sich ein Verhältnis von 250 zu 1 als sehr sicher erwiesen, kein Signal zu verpassen.

3.3.2 Properties

Die Klasse Properties erweitert die Funktionalität des ARINA um einen nicht zwangsweise essentiellen, aber in Bezug auf Komfort und Benutzerfreundlichkeit angenehm zuträglichen Aspekt. Wie im Abschnitt zur Ethernet-Klasse 3.3.5 erläutert, benötigt der ARINA eine festgelegte lokale IP- sowie MAC-Adresse, zusätzlich dazu ist die Angabe der IP-Adresse des Partner-ARINA sowie der entsprechende Port sicherzustellen. Für die Authentifizierung, die in Abschnitt 3.3.5 beschrieben ist, wird zudem ein vor Verbindungsaufbau beiden ARINA bekanntes Passwort benötigt. Mit der Möglichkeit diese Angaben über eine Konfigurationsdatei, welche sich einfach und unproblematisch über eine SD-Karte einspielen lässt, zu liefern, bleibt es dem Nutzer erspart sich zwangsweise mit einzelnen Elementen des Programmcodes auseinanderzusetzen. Es sind keine festcodierten Angaben zur IP, MAC und Empfänger-IP-Adresse im Programmcode des ARINA enthalten, somit ist die Angabe dieser mittels der zuvor genannten Konfigurationsdatei obligatorisch für den Betrieb des jeweiligen Adapters. In diesem Fall wird automatisch der Port 4711 gewählt. Folgende notwendige Angaben können mittels der Konfigurationsdatei gemacht werden: lokale IP-Adresse, lokale MAC-Adresse, Empfänger-IP-Adresse und Port. Zusätzlich können ebenfalls Passwort zur Authentifizierung, Gateway, DNS-Server, und Netzmaske angegeben werden. Sind die angegebenen Passwörter zweier sich verbindender ARINA identisch, kann sich der IR-Signale-empfangende-ARINA dem IR-Signale-sendenden-ARINA gegenüber authentifizieren, eine genauere Erläuterung findet sich in einem der folgenden Abschnitte. (Siehe 3.3.5)

Für die Versuchsaufbauten in 4.2 wurde diese Konfiguration gewählt:

Konfiguration ARINA 1

```
locMac=10:00:00:00:01:00
locIP=10.0.0.1
remIP=10.0.0.2
port=1234
locPW=5icheres)(P4sswort
ENDE
```

Konfiguration ARINA 2

```
locMac=10:00:00:00:01:01
locIP=10.0.0.2
remIP=10.0.0.1
port=1234
locPW=5icheres)(P4sswort
ENDE
```

3.3.3 Vector

Die rudimentäre Vektorklasse wurde von uns implementiert, da nichts vergleichbares vorhanden war und wir zur Speicherung der Rohdaten einen erweiterbaren Byte-Vektor benötigen. Es wurden nur die für unsere Zwecke notwendigen Funktionen „push-back()“ und „at()“ implementiert, mit der gleichen Funktionalität wie in der Vektorklasse der C++ Standardbibliothek. Wir mussten uns auf den Datentyp Byte festlegen, da der Compiler in der Version 1.0.5 keine Templates unterstützt (siehe 3), die es ermöglichen würden, diese Klasse für beliebige Datentypen zu verwenden.

3.3.4 Remote

Das entwickelte Protokoll findet durch die Klasse Remote seinen Einsatz im ARINA. Beim Erstellen eines remote-Objekts wird ein durch die Bibliothek gesteuertes IRrecv-Objekt (3.2) erstellt, welches für die direkte Interaktion mit der jeweiligen IR-Sende und Empfangs-Diode zuständig ist. Die Funktion sendIR der Klasse Remote bereitet durch die Gegenstelle übertragene IR-Signale aus ihrer protokollkonformen Form wieder in ein für die IR-Bibliothek interpretierbares Format auf. Anschließend werden die empfangenen Daten an die aus der Bibliothek dafür vorgesehenen Methoden übergeben. Abschließend gibt die Funktion genutzten Buffer wieder frei und schaltet die IR-Sende-Diode aus (Siehe 3.4). In der Funktion recvIR werden die Rückgabewerte der Bibliothek als Grundlage für den Aufbau eines Byte-Vektors genutzt. Der Aufbau der später übertragenen Daten wird durch einzelne bitweise Verschiebungen des aus der Bibliothek erhaltenen 32 bit long-Wertes in einzelnen 4 bit Byte-Werte und Einfügen in einen Byte-Vektor (Siehe 3.3.3) realisiert. Es wurden diese hardwarenahen Operationen verwendet und nicht auf verfügbare Umwandlungsfunktionen zurückgegriffen, um höchstmögliche Performanz zu erreichen.

3.3.5 OurNetwork

Die Netzwerkklasse „OurNetwork“ abstrahiert die von uns benötigten Netzwerkfunktionen. Sie instanziert Objekte der weiteren Klassen „authentification.h“, „crypt.h“ und „return.h“, die später in diesem Abschnitt erklärt werden. Zunächst gibt es mit der „initEthernet“ eine Funktion, mit der die MAC-Adresse, die eigene IP, der Port, auf dem der eigene und fremde Listen-Sockets geöffnet werden, die IP der Gegenstelle sowie das zur Authentifikation notwendige Passwort übergeben werden. Sie initialisiert dann auch die Netzwerkschnittstelle. Die beiden Funktionen „startServer“ und „startClient“ machen was ihr Name beschreibt, sie starten die Serverfunktionalität, die ein listen Port öffnet, und die Client Funkionalität, die versucht sich zu einem Server zu verbinden. „getConnection“ überprüft, ob ein Client am eigenen Server versucht eine Verbindung aufzubauen oder als Client selbst versucht eine Verbindung zu einem anderen Server aufzubauen. „sendIR“ sendet empfangene Infrarotdaten an die verbundene Gegenstelle, wenn bereits eine existiert, und versucht eine Verbindung aufzubauen, wenn keine vorhanden ist. Wohingegen „recvIR“ Daten empfängt, wenn eine Verbindung vorhanden ist und wenn nicht überprüft, ob ein anderer Arduino versucht eine Verbindung aufzubauen. Die Antwortfunktion „sendAnswer“ sendet eine eventuelle Antwort an die Gegenstelle, dass ein empfangenes Infrarotsignal erfolgreich oder nicht erfolgreich ausgegeben wurde.

Als letzes gibt es noch die Möglichkeit mit „isConnected()“ zu überprüfen, ob es eine aktive Verbindung zu einem anderen ARINA gibt. Wenn zwei ARINA verbunden sind ist es möglich, dass beide sowohl Daten senden als auch empfangen können, wodurch auch eventuelle Antworten der Geräte zurück übermittelt werden.

Die Klasse „Authentification“ ist für die Authentifizierung der beiden miteinander kommunizierenden ARINA zuständig und wird zu Beginn jeder Verbindung automatisch ausgeführt.

„Crypt“ ist eine abstrakte Klasse, die jedoch immer aufgerufen wird, wenn Daten gesendet oder empfangen werden, um die leichte Implementation einer Verschlüsselung des Datenverkehrs zu ermöglichen.

Die außerdem implementierte Klasse „Return“ ist für die Erstellung einer Antwort zuständig sobald etwas implementiert wurde, um zu überprüfen, ob die empfangenen IR Daten auch ausgegeben wurden.

3.3.6 LED

Die Klasse ledState kann eine erst zur Laufzeit angegebene Menge von LEDs verwalten, die sie zunächst an den angegebenen Ports initialisiert, um dann über die Funktionen setLED und getLED ihren Status zu ändern und auszugeben. Zum Initialisieren muss nur die Anzahl der LEDs und die digital I/O Pins angegeben werden, an denen die Minuspole der LEDs angeschlossen sind. Wir nutzen sie, um drei LEDs anzusteuern, die bereits zur Kompilezeit festgelegt sind, jedoch ist die Klasse nutzbar um jegliche über die digitalen I/O Pins angeschlossenen Bauteile anzusteuern, solange sie kein pulsweitenmoduliertes Signal benötigen.

3.4 Besonderheiten und Probleme

Die von uns in der Version 1.0.5 [7] verwendete Arduino IDE hat einige Eigenarten, wie zum Beispiel reproduzierbare Abstürze sowie gelegentlich auftretende Abstürze die keine Regelmäßigkeit zu haben scheinen und nicht reproduzierbar sind. Außerdem muss man sich an strikte Namensgebung halten, was Dateien und Bibliotheken betrifft. Weiterhin treten Probleme auf, wenn man Funktionen in einer .h Header Datei implementiert und definiert, statt die Implementierung in eine .cpp Datei gleichen Namens auszulagern. Die dadurch vom Compiler ausgegeben Fehler lassen nur schwer auf das ursächliche Problem schließen.

Um auf Linuxsystemen mit der IDE auf USB Ports zugreifen zu können, um die Arduinos zu programmieren oder über die serielle Schnittstelle mit ihnen zu kommunizieren, ist es anscheinend unerlässlich der IDE Rootzugriff zu geben.

Das Debuggen der Software funktioniert ohne kostenpflichtige Software, wie das Plugin von Visual Micro für Visual Studio [6] nicht, was Speicherzugrifffehler und Memoryleaks schwer auffindbar macht und den Zeitaufwand für größere Projekte erhöht.

Nicht alle in der IDE gegebenen Funktionen entsprechen dem, was man von C++ gewohnt ist. Zum Beispiel gibt die String-Funktion „equals()“ in C++ 0 zurück, wenn die zu vergleichenden Strings gleich sind. Beim Arduino bekommt man bei Gleichheit true zurück, was zu Verwechslungen bei der Implementierung führen kann.

Da Arduinos kein Multithread unterstützen und über sehr begrenzten Arbeitsspeicher und Programmspeicher verfügen, muss bei der Programmierung der Software stark auf Echtzeitfähigkeiten, möglichst schnellen Code und speicherschonendes Design geachtet werden.

Bei der Nutzung des Ethernet / SD-Card Shields auf dem Mega 2560 fällt auf, dass man den Pin 53 und den Pin 10 manuell als Ausgang definieren muss. Vergisst man dies, so stürzt die Software beständig ab, ohne direkte Hinweise auf die Ursache. Diese beiden Pins werden zur Kommunikation zwischen dem Atmell Chip des Arduinos und dem Prozessor des SD-Lesers genutzt, obwohl der Pin 53 vom Arduino Mega 2560 keine mit bloßem Auge sichtbare Verbindung zum Ethernet/SD-Card Shield hat.

Der Compiler der IDE kennt leider in dieser Version keine Templates, was die Implementierung von Klassen wie universeller Vektorklassen verhindert. Die experimentelle Version 1.5 BETA unterstützt diese Funkionalität zwar rudimentär, weist aber viele andere Fehler auf, die die Nutzung für unsere Zwecke unmöglich machte.

Die verwendete IRRemote Bibliothek (Siehe 3.2) hat den Fehler, dass sie nach der Ausgabe eines Infrarotsignals den Infrarot-Emitter nicht ausschaltet, sondern leuchten lässt, was unnötig Energie verbaucht und eventuell andere IR-Signale stören kann. Dies lässt sich leicht unterbinden, indem nach jedem Aufruf der Funktion zur Ausgabe von IR-Signalen der digitale I/O Pin auf aus gesetzt wird.

4 Evaluation

In der theoretischen Ausarbeitung wurde das Ziel dieses Projekts festgelegt; es sollte eine Möglichkeit zur Weiterleitung von IR-Signalen geschaffen werden, die die Bedienung von infrarotansteuerbaren Geräten ermöglicht. In der nun folgenden Evaluation wird durch verschiedene durchgeführte Tests belegt, dass ARINA diese Problemstellung löst. Der Aufbau dieser Test sowie die beobachteten Ergebnisse werden dargelegt und erläutert. Zusätzlich wurden bei der Entwicklung aber auch vorher nicht spezifizierte Eigentschaften teil von ARINA, diese die Bedienung verbessernde jedoch auch teils sicherheitsrelevanten Eigenschaften werden ebenfalls vorgestellt.

4.1 Zusätzliche Eigenschaften von ARINA

Die zuvor genannte Notwendigkeit (Siehe 3.3.5, 2.2) zur Angabe einer IP-, MAC- und Empfänger-IP-Adresse sowie die Möglichkeit ein Passwort zum Authentifizieren anzugeben, ließ verschiedene Möglichkeiten diese Anforderung zu erfüllen. Eine naive jedoch für den Anwender sehr unpraktische Möglichkeit wäre das Festsetzen der entsprechenden Parameter im Quellcode des ARINA. Diese würde den Anwender bei jeder auch noch so simplen Änderung dazu nötigen, ein komplett neues Kompilat der Software zu erstellen und auf den jeweiligen ARINA zu flashen. Eine auch für einen weniger erfahrenen Anwender einfache Möglichkeit stellt daher das Erstellen einer einfach strukturierten Konfigurationsdatei dar, welche auf eine SD-Karte übertragen und dann dem gewünschten ARINA zugeführt wird. Ein per Tastendruck durchgeführter Reset des Adapters lässt ihn fortan mit den gewünschten Einstellungen agieren. Diese Funktionalität wird durch die Klasse Properties (Siehe 3.3.2) bereitgestellt und wurde eigens für den ARINA umgesetzt.

Als weiteres Feature wurde eine rudimentäre Authentifizierung implementiert, die ein Klartextpasswort austauscht und überprüft, ob beiden miteinander kommunizierenden ARINA das gleiche Geheimnis bekannt ist. Diese Funktion kann leicht durch Implementierung einer Verschlüsselung in der Crypt Klasse verbessert werden. Dazu werden im Ausblick (Siehe 5.2) noch weitere Ideen vorgestellt.

Außerdem ermöglichen angeschlossene LEDs, den Zustand der Verbindung zwischen zwei ARINA anzuzeigen. Eine weitere LED, die die Ausgabe sowie den Eingang von Daten signalisiert, gibt dem Anwender ein visuelles Feedback, ob überhaupt eine Funktion ausgeführt wird.

4.2 Testaufbau und Ergebnisse

Um die Funktionstüchtigkeit bestätigen zu können, wurden verschiedenste Tests durchgeführt. Die jeweiligen Versuchsaufbauten und die beobachteten Ergebnisse werden im Folgenden beschrieben.

Ein mehrstufiger Testaufbau kam für die Untersuchung der Netzwerkfunktionalität zum Einsatz. In der ersten Stufe des Verfahrens wurden zunächst zwei ARINA direkt mit einem Twisted-Pair-Netzwerkkabel verbunden, um den korrekten Verbindungsaufbau zwischen beiden Adaptern zu prüfen. Dabei wurden zunächst die standardisierten Crossoverkabel eingesetzt, die bei einem ihrer beiden RJ45-Stecker gewisse Kabeladern vertauscht haben. Im Anschluss wurde ebenfalls eine Verbindung mittels Patchkabeln getestet. Es zeigte sich keinerlei Unterschied hinsichtlich der Konnektivität zwischen den jeweiligen Verbindungsarten. Wir konnten somit eine Untersützung von Auto-MDI(X) [5] bei den verwendeten Arduino Ethernet-Shields feststellen. Die nächste Stufe des Tests bestand darin, dass wir beide Adapter mittels eines 16-Port Netzwerkswitches verbanden. Auch dieser Test verlief erfolgreich. Auch eine Verbindung beider Adapter über das universitätsinterne Netzwerk verlief erfolgreich, ebenso wie ein Test über eine VPN-Verbindung. Zusammenfassend ist mit diesen Test gezeigt worden, dass das von uns entwickelte Netzwerkhandling eine korrekte Netzwerkkonnektivität gewährleistet.

Zum Abschluss unseres Projekts wurde eine weitere Reihe von Tests durchgeführt, um die Funktionsweise des ARINA mit verschiedensten Gerätetypen und Herstellern zu prüfen, und so die breite Vielfalt an Anwendungsmöglichkeiten zu bestätigen. Damit die Vergleichbarkeit der einzelnen Tests gewährleistet ist, wurde sich jeweils an ein zuvor entworfenes Versuchsschema (siehe Abbildung 5) gehalten. Der Aufbau eines Tests bestand demzufolge aus jeweils 6 Schritten: Im ersten Schritt wurden noch nicht im Netzwerk vertretene IP- und MAC-Adressen für die beiden verwendeten ARINA gesetzt. Der zweite Schritt bestand im Aufbau des Versuchsumfeld, wobei der infrarotsendende ARINA mit seiner infrarotemittierenden Diode im Abstand von einem Meter zum entsprechenden Gerät ausgerichtet wurde. Der zweite ARINA, welcher als Empfänger für das Signal der Fernbedienung fungierte, wurde unter Ausschluss jeglicher (auch reflektierter) Sichtverbindung platziert. Auch hierbei wurde ein Abstand von einem Meter zwischen Fernbedienung und der im infraroten Spektralbereich arbeitenden Diode eingehalten. Im dritten Schritt wurde die lokale Netzwerkkonnektivität beginnend mit einem Reset beider ARINA und dem anschließenden Übertragen beliebiger Infrarotsignale durch das Aufleuchten der grünen Status-LEDs bestätigt. Nachdem nun der vorbereitende Versuchsaufbau abgeschlossen war, konnte mit dem eigentlichen funktionalen Test begonnen werden. Somit wurde im vierten Schritt zunächst die Funktionalität von Standardtasten bei Multimediageräten - wie zum Beispiel „Power“, „Lauter / Leise“, „Kanal hoch / Kanal runter“ sowie jeweilige Programmtasten von null bis neun - sofern vorhanden überprüft. Daraufhin erfolgte ein ergänzender Test mit verbleibenden Sonderfunktionstasten - „OSD Menü“, „Zoom“, Pfeiltasten, etc - die erfolgten Beobachtungen finden sich in der Tabelle 11 wieder. Es zeigte sich, dass ARINA äußerst zufriedenstellende Ergebnisse lieferte und bei allen Tests jegliche Tastenbelegung unabhängig von Hersteller und Gerät unterstützte.

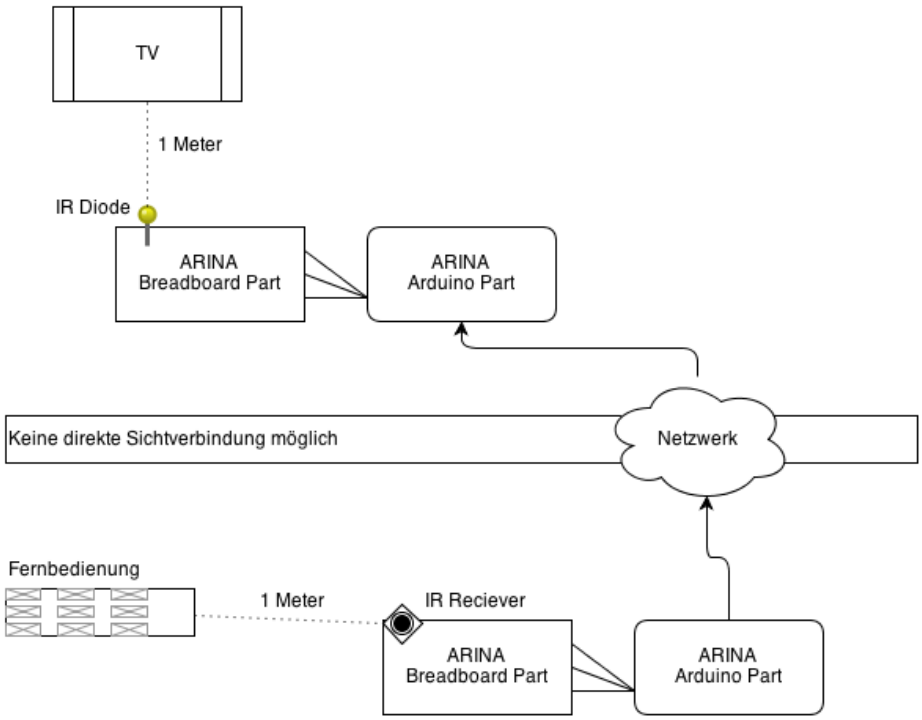

Abbildung 10: Schematische Darstellung des Versuchsaufbaus

Gerätename	Gerätetyp	Hersteller	Protokoll	alle Tasten	Kaufjahr
EB-1771W	Beamer	Epson	NEC	Ja	2012
UE40EH5000	TV	Samsung	RAW	Ja	2012
BD-E5500	3D-BR-Player	Samsung	RAW	Ja	2012
42LV579S	TV	LG	NEC	Ja	2012
DVX592H	DVD-Player	LG	NEC	Ja	2012
Connect ID 32	TV	Loewe	RC-5	Ja	2009
230E	DVD Player	Redstar	NEC	Ja	2001
55FU4243	TV	Thomsen	NEC	Ja	2012
Firestorm IR	Mini-Helekopter	AMEWI	RAW	Ja	2012

Abbildung 11: Tabelle mit den Ergbenissen der Praxisversuchen

4.2.1 Direktreflektor

Um den Aufbau eines ARINA zu testen, implementierten wir eine Funktionalität, die ein Infrarotsignal empfängt und für mehrere Sekunden zwischenspeichert, um sie dann wieder auszugeben. Dadurch wird beim mit der Infrarotfernbedienung bedienten Gerat das Phänomen hervorgerufen, dass jeder Tastendruck nach einigen Sekunden wiederholt wird. Bei diesem Versuch ist das in Abschnitt 2.2 beschriebene Problem der begrenzten Reichweite aufgefallen und konnte behoben werden. Zurerst gelang das Wiederholen von Signalen, die auf bekannten Protokollen beruhen, jedoch gab es Probleme mit dem Wiederholen von Signalen mit unbekanntem Protokoll, die intern als Rohdaten behandelt werden. Dadurch ist aufgefallen, dass die Ausgabe des RAW-Signals mit einer Pause anfängt aber mit einem „IR-Emitter an“ beginnen sollte, um durch die IRRemote Bibliothek (siehe 3.2) erfolgreich wiedergegeben werden zu können. Außerdem entdeckten wir hier die Existenz der in 3.2 beschriebenen Konstanten von 50 Mikrosekunden zum Multiplizieren der ausgegebenen Werte, da dies nicht in der Bibliothek geschieht. So erhält man die Rohdaten in Mikrosekunden, wie lang der IR-Emitter jeweils ein PWM-Signal ausgeben soll.

Nach dem Beheben der diversen Fehler funktionierte dieser Versuch mit den von uns getesteten Geräten:

Epson Beamer EB−1771W (Bekanntes NEC Protokoll)
Samsung Fernseher (Unbekanntes Protokoll)

4.2.2 Reflektor

Wie genau IR-Signale empfangen und wiedergegeben werden können, testeten wir mit zwei ARINA, die sich mit geringem Abstand gegenüber standen, und die die in 4.2.1 beschriebene Software aufgespielt hatten. Wurde jetzt mit einer Infrarotfernbedienung ein IR-Signal an einen der beiden ARINA gesendet, so sendete er dieses Signal nach kurzer Pause wieder aus, so dass der ihm gegenüberstehende ARINA dieses Signal empfing. Dies wurde solange wiederholt bis das neben dem Aufbau stehende Gerät, zudem die IR-Fernbedienung gehört, nicht mehr so auf die Signale reagierte, wie es nach Druck der jeweiligen Taste auf der Fernbedienung sollte. Im Durchschnitt konnten wir ein Signal drei bis vier mal wiederholen bevor es nicht mehr erkannt wurde. Als Spitzenwert wurde ein Signal noch nach 15 Reflexionen wiedererkannt. Die empfangenen IR-Signale veränderten sich in jedem Schritt, so dass manche Werte größer und manche kleiner wurden ohne das eine Regelmäßigkeit erkennbar ist.

5 Ausblick

Im folgenden, diese Ausarbeitung abschließenden, Ausblick werden zukünftige Erweiterungen und ausbaufähige Funktionen für den ARINA vorgestellt. Die Erweiterungen werden in hardware- und softwarespeziefische Thematik unterteilt, vorgestellt und knapp bewertet.Ebenfalls wird auf mögliche Projekte eingegangen, in denen die bei der Entwicklung des ARINA entstandene Software sowie das definierte Protokoll zur Übertragung von IR-Signalen einen grundlegenden Beitrag liefern können.

5.1 Hardware

Die Wahl eines Arduino Mega 2560 als Basiskomponente erlaubt durch die große Verfügbarkeit von Erweiterungsplatinen - so genannten "Shields" - eine beständig wachsende Auswahl an zusätzlichen Möglichkeiten für Erweiterungen. Ein Funkmodul für die Frequenzbereiche 315/433 MHz würde, nach ähnlichem Prinzip wie auch das Empfangen, Decodieren und anschließende Weiterleiten von IR-Signalen, das Weiterleiten von Befehlen zum Beispiel zum Ansteuern einer Garagensteuerung oder funkgesteuerter Steckdosen ermöglichen. Der ARINA kann auf einfache Art und Weise um die entsprechenden Hardwarekomponenten erweitert werden, es müsste eine entprechende Ausarbeitung der verwendeten Protokolle und die Erstellung einer diese Protokolle decodierende Bibliothek erfolgen. Ein schon verfügbares GSM-Shield kann nach entsprechender softwareseitiger Erweiterung ebenfalls zum Absetzen von Befehlen, die per IR-Befehl ausgegeben werden, genutzt werden. Um auch ein direktes kabelloses Übertragen zwischen zwei ARINA zu ermöglichen, wäre die Erweiterung um ein WLAN Shield ein möglicher Ersatz zu dem verwendeten Ethernet Shield. Wenn nur kleine Übertragungsstrecken zu überwinden sind, wäre ein Bluetooth Shield ebenfalls in der Lage diese Aufgabe zu erfüllen. Beide würden nur geringe Shield-spezifische Änderungen in der bisher verwendeten ourEthernet Klasse (siehe 3.3.5) notwendig machen, der allgemeine Programmablauf sowie das definierte Protokoll wären davon unberührt. Um die für jeden ARINA notwendigen Konfigurationsdaten bereitzustellen, kann alternativ zum Bereitstellen der Konfigurationsdetails per SD-Karte, auch eine Eingabe über ein abgeschlossenes Tastenfeld erfolgen. Sinnvoll wäre dann die zeitgleiche Ergänzung um ein Display, welches die eingegebenen Konfigurationsdaten zur Kontrolle anzeigt. Zur Umsetzung ist nur eine simple Routine zum Einlesen vom Tastenfeld sowie Anzeigen von ASCII Zeichen am angeschlossenen Display notwendig. Änderungen am allgemeinen Programmablauf des ARINA sind auch in diesem Fall nicht erforderlich.

5.2 Software

Die Authentifizierungs- und Cryptographiefunktionalität ist bisher nur rudimentär implementiert, sodass bei der Version der Software, die zum Ende des Praktikums besteht, das Passwort unverschlüsselt zwischen den beiden ARINA ausgetauscht und dann verglichen wird. Dies lässt sich verbessern, indem Verschlüsselungsfunktionen eingebaut werden, wie etwa eine einfache XOR verschlüsselung mit einer in der Konfigurationsdatei angegebenen Zeichenfolge. Ein alternative Möglichkeit zur Verbesserung von Passwort und Veschlüsselung ließe sich schaffen, indem man die freien Pins des Arduinos nutzt, um durch Kabelbrücken oder DIP-Switches eine Kodierung vorzugeben, mit der die Verschlüsselung und das Passwort kombiniert werden. Die so konstruierte Verschlüsselung wäre nicht nur von in der Software wählbaren Einstellungen abhängig, was die Sicherheit erheblich verbessern würde. Mit der Schaffung einer zentralen Instanz, an der sich ARINAs anmelden, ergeben sich viele weitere Möglichkeiten. Sicherzustellen wäre zunächst jedoch, dass die einzelnen ARINAs aber auch die zentrale Instanz gegenüber dem ARINA eindeutig authentifizieren, um sicherzustellen, dass die so entstandene Vermittlungsstelle vertrauenswürdig ist. Mit dem Schaffen einer zentralen Instanz wäre es möglich, einzelne vorher erkannte Befehle an die verbundenen ARINAs zu übertragen und so zum Beispiel eine Ansteuerung eines TV-Gerätes über eine Weboberfläche zu realisieren. Alternativ zu einer Weboberfläche könnte auch eine App für ein Smartphone das Versenden des entsprechenden Befehlscode zum ARINA auf der zentralen Instanz anstoßen. Die in dieser Arbeit vorgestellte Software des ARINA wäre dazu nur geringfügig anzupassen, Änderungen am Hardwareaufbau und Protokoll sind nicht notwendig. Der Entwurf eines Servers und der entsprechenden Smartphoneapps stellen von daher den aufwändigeren Teil dieses neuen Projektes dar. Weitere Konzepte wie sie unter dem Oberbegriff „Smart Home“ immer populärer [2] werden, wären ebenfalls realisierbar. Die zuvor vorgestellte Erweiterbarkeit zur Ansteuerung von Funksteckdosen ermöglicht so, neben der in dieser Arbeit geschaffenen Möglichkeit TV- und andere Hifi-Geräte mittels Infrarotbefehlen anzusteuern, beste Grundlagen um eine preiswerte Alternative zu schon verfügbaren Smart Home Lösungen umzusetzen. Eine weitere sinnvolle Erweiterung der Software ist das Zurückfallen auf Standartwerte, wenn das Auslesen aus der „settings.txt“ von der SD-Karte nicht funktioniert. Jedoch würden dann alle Geräte, bei denen das der Fall ist, die gleiche MAC-Adresse und die gleiche IP-Adresse verwenden. Um das zu umgehen, kann man die MAC-Adresse zufällig generieren und die IP-Adresse per DHCP zuweisen lassen. Um sich dann zu finden, ist es nötig einen Multicast in das Netzwerk abzusetzen auf den der andere ARINA antworten kann.

6 Aufteilung

Kapitel	Autor
1	Rene
2	Rene
2.1	Thomas
2.2	Thomas
3	Rene
3.1	Thomas
3.1.1	Rene
3.1.2	Rene
3.1.3	Rene
3.2	Rene + Thomas
3.3	Thomas
3.3.1	Thomas
3.3.2	Rene
3.3.3	Thomas
3.3.4	Rene
3.3.5	Thomas
3.3.6	Thomas
3.4	Thomas
4	Rene
4.1	Rene + Thomas
4.2	Rene
4.2.1	Thomas
4.2.2	Thomas
5	Rene
5.1	Rene
5.2	Rene + Thomas

Abbildung	Autor
1	Thomas
4	Thomas
5	Thomas
6	Thomas
7	Rene
8	Rene
9	Rene
10	Rene
11	Rene + Thomas

7 Anhang

7.1 Software Dokumentation

Im Nachfolgenden befindet sich die 54 Seiten umfassende Dokumentation, die wir mit dem Tool Doxygen [3] erstellt haben. Sie wird vollständig automatisiert aus den im Quelltext angegebenen Kommentaren und dem Sourcecode selbst erstellt.

Literatur

[1] Arduino Mega 2560 Produktseite. http://arduino.cc/en/Main/ArduinoBoardMega2560.

[2] Deutschlandradio Kultur. http://www.dradio.de/dkultur/sendungen/ewelten/2077261/.

[3] Doxygen. www.doxygen.org.

[4] Ken Shirriff's blog. http://www.righto.com/2009/08/multi-protocol-infrared-remote-library.html.

[5] Medium Dependent Interface. http://de.wikipedia.org/wiki/Medium_Dependent_Interface.

[6] Visual Micro. www.visualmicro.com.

[7] Arduino project. http://www.arduino.cc/, 2013.

[8] Ken Shirriff. IRremote library for the Arduino. https://github.com/shirriff/Arduino-IRremote, 2012.

[9] René Neff & Thomas Trimborn. Theoretische Ausarbeitung zur Entwicklung einer Netzwerk-basierten Infrarot-Fernbedienung. 2013.

[10] René Neff & Thomas Trimborn. Theoretische Ausarbeitung zur Entwicklung einer Netzwerk-basierten Infrarot-Fernbedienung. Kapitel 2.2, Seite 5, 2013.

[11] René Neff & Thomas Trimborn. Theoretische Ausarbeitung zur Entwicklung einer Netzwerk-basierten Infrarot-Fernbedienung. Kapitel 4.3, Seite 14ff, 2013.

[12] WIZnet, http://www.wiznet.co.kr/UpLoad_Files/ReferenceFiles/W5100_Datasheet_v1.2.2.pdf. *W5100 Datasheet*.

Alle Websiten wurden zuletzt aufgerufen am 14.10.2013.

ARINA-ArduinoRemoteInfrarotNetworkAdapter

Erzeugt von Doxygen 1.6.1

Fri Nov 8 10:15:01 2013

Inhaltsverzeichnis

Kapitel 1

Datenstruktur-Verzeichnis

1.1 Datenstrukturen

Hier folgt die Aufzählung aller Datenstrukturen mit einer Kurzbeschreibung:

Kapitel 2

Datei-Verzeichnis

2.1 Auflistung der Dateien

Hier folgt die Aufzählung aller Dateien mit einer Kurzbeschreibung:

Kapitel 3

Datenstruktur-Dokumentation

3.1 Authentification Klassenreferenz

`#include <authentification.h>`

Öffentliche Methoden

- Authentification (String *password)

  Konstruktor.

- boolean checkPassword (String *externPassword)

  ueberprueft das Password

- boolean isAuthentificated ()

  gibt an ob die Authentifizierung bereits erfolgreich durchgeführt wurde

Private Attribute

- String * password
- boolean authentificated

3.1.1 Beschreibung der Konstruktoren und Destruktoren

3.1.1.1 Authentification::Authentification (String * *password*)

Konstruktor. sichert das uebergebene Password in die private variable

Parameter:

String password Passwort, das noetig ist um eine Verbindung mit diesem Agenten aufzubauen

3.1.2 Dokumentation der Elementfunktionen

3.1.2.1 boolean Authentification::checkPassword (String ∗ *externPassword*)

ueberprueft das Password Ueberprueft ob das angegebene Password mit dem gespeicherten ueberein stimmt

Parameter:

String externPassword Vom anderen Agenten uebertragenes Password

Rückgabe:

true wenn das Password richtig ist, sonst false

3.1.2.2 boolean Authentification::isAuthentificated ()

gibt an ob die Authentifizierung bereits erfolgreich durchgeführt wurde

Rückgabe:

true wenn ja, sonst false

3.1.3 Dokumentation der Datenelemente

3.1.3.1 boolean Authentification::authentificated `[private]`

3.1.3.2 String∗ Authentification::password `[private]`

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- arina/authentification.h
- arina/authentification.cpp

3.2 Crypt Klassenreferenz

`#include <crypt.h>`

Öffentliche Methoden

- Crypt ()
- ~Crypt ()
- byte * encrypt (byte *data)
  encrypts given data
- byte * decrypt (byte *data)
  decrypts given data

3.2.1 Beschreibung der Konstruktoren und Destruktoren

3.2.1.1 Crypt::Crypt () [inline]

3.2.1.2 Crypt::~Crypt () [inline]

3.2.2 Dokumentation der Elementfunktionen

3.2.2.1 byte * Crypt::decrypt (byte * *data*)

decrypts given data

Parameter:

byte *data data to decrypt

Rückgabe:

decrypted data

3.2.2.2 byte * Crypt::encrypt (byte * *data*)

encrypts given data

Parameter:

byte *data data to encrypt

Rückgabe:

encrypted data

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- arina/crypt.h
- arina/crypt.cpp

3.3 LEDState Klassenreferenz

`#include <ledstate.h>`

Öffentliche Methoden

- LEDState (int num,...)

  Konstruktor.

- ~LEDState ()

  Destruktor.

- int setLED (int number, boolean state)

  setzt eine LED in einen angegebenen Zustand

- boolean getLED (int number)

  gibt den Status einer LED aus

Private Attribute

- int numberOfLED
- int * ledPin
- boolean * ledState

3.3.1 Beschreibung der Konstruktoren und Destruktoren

3.3.1.1 LEDState::LEDState (int *num*, ...)

Konstruktor. Erzeugt die notwendigen Arrays und fuellt sie mit inhalt

Parameter:

int num Anzahl der LED's die genutzt werden sollen

... Pinnummern der zu nutzenden LED's

3.3.1.2 LEDState::~LEDState ()

Destruktor. Zerstoert die im Konstruktor angelegten arrays

3.3.2 Dokumentation der Elementfunktionen

3.3.2.1 boolean LEDState::getLED (int *number*)

gibt den Status einer LED aus gibt den status einer mit number spezifizierten LED als boolean aus, wenn diese Nummer existiert, sonst gibt sie immer false zurueck

Parameter:

int number LED, deren Status abgefragt wird

Rückgabe:

Status der LED, oder false, wenn die LED nicht existiert

3.3.2.2 int LEDState::setLED (int *number*, boolean *state*)

setzt eine LED in einen angegebenen Zustand setzt eine LED, die mit number angegeben wird in den Zustand der in state uebergeben wird wurde das gemacht gibt die Funktion 1 zurueck, andernfals 0

Parameter:

int number nummer der LED die geaendert werden soll

boolean state status der fuer die LED absofort gelten soll

Rückgabe:

wenn setzen erfolgreich, dann 1 sonst 0

3.3.3 Dokumentation der Datenelemente

3.3.3.1 int* LEDState::ledPin [private]

3.3.3.2 boolean* LEDState::ledState [private]

3.3.3.3 int LEDState::numberOfLED [private]

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- arina/ledstate.h
- arina/ledstate.cpp

3.4 OurEthernet Klassenreferenz

`#include <ourEthernet.h>`Zusammengehörigkeiten von OurEthernet:

Öffentliche Methoden

- OurEthernet ()

  Konstruktor.

- int initEthernet (byte ∗mac, byte ∗myip, byte ∗targetip, int port, String ∗Password)

  initialisiert das Ethernet

- int startServer (int port)

  startet den Server

- int startClient (byte ∗ip, int port)

  startet den Client

- int getConnection ()

  Versucht eine Verbindung aufzubauen.

- int sendIR (Vector ∗code)

  kümmert sich um den übertragunsweg und stößt das Senden an

- Vector ∗ recvIR ()

  Holt Daten vom Ethernetport und gibt sie entschlüsselt zurück.

- int sendAnswer (Vector ∗answer)

  Sendet Antwort an anderen Agenten.

- boolean isConnection ()

  Gibt zurück ob es eine Verbindung zu einem anderen Agenten gibt.

Private Methoden

- int sendData (Vector ∗code)

  sendet daten

- boolean doAuth ()

  Führt die Authentifikation durch.

- int sendEncData (Vector ∗code)

  sendet die verschlüsselten Daten an den Client

Private Attribute

- EthernetClient tmpclient
- EthernetClient * client
- EthernetServer * server
- Crypt * crypt
- Authentification * auth
- String * password
- byte * targetIP
- int port

3.4.1 Beschreibung der Konstruktoren und Destruktoren

3.4.1.1 OurEthernet::OurEthernet ()

Konstruktor. setzt Pointer auf NULL und initialisiert crypt und Authentification

3.4.2 Dokumentation der Elementfunktionen

3.4.2.1 boolean OurEthernet::doAuth () `[private]`

Führt die Authentifikation durch. Sendet sein Passwort an den anderen Agenten und holt von der Netzwerkschnittstelle Daten bis zum ersten Trennzeichen ab um diese mit dem gespeicherten Password zu vergleichen

Rückgabe:

true wenn die Authentifizierung erfolgreich war, sonst false;

3.4.2.2 int OurEthernet::getConnection ()

Versucht eine Verbindung aufzubauen.

Rückgabe:

1 Wenn erfolgreich, sonst 0

3.4.2.3 int OurEthernet::initEthernet (byte * *mac*, byte * *myip*, byte * *targetip*, int *port*, String * *pass*)

initialisiert das Ethernet initialisiert die Schnitstelle und speichert ip und port fuer eine verbindung

Parameter:

byte *mac mac fuer die Ethernetschnittstelle

byte *myip ip fuer die Ethernetschnittstelle

byte *targetip ip zu der eine Verbindung aufgebaut werden soll

int port port zu dem eine Verbindung aufgebaut werden soll

String *password passwort für die authentification

Rückgabe:

gibt 1 zurueck

3.4.2.4 boolean OurEthernet::isConnection ()

Gibt zurück ob es eine Verbindung zu einem anderen Agenten gibt.

Rückgabe:

true wenn es eine Verbindung gibt, sonst false

3.4.2.5 Vector * OurEthernet::recvIR ()

Holt Daten vom Ethernetport und gibt sie entschlüsselt zurück. Ließt Daten Byteweise vom Ethernetport, bis zu einem Trennzeichen, entschlüsselt sie und speichert sie im Ergebnisbuffer. Das tut sie 2 mal, dann wird das Ergebnis zurück gegeben;

Rückgabe:

Empfangene Daten als String

3.4.2.6 int OurEthernet::sendAnswer (Vector * *answer*)

Sendet Antwort an anderen Agenten. Sendet die Antwort an den anderen Agenten, der diesem Daten gesendet at die ausgegeben werden sollen.

Parameter:

unsigned long *answer Antwort die gesendet werden soll

Rückgabe:

Anzahl der gesendeten bytes

3.4.2.7 int OurEthernet::sendData (Vector * *code*) `[private]`

sendet daten sendet daten, nachdem die authentifikation entweder ueberprueft oder durchgefuehrt wurde und die daten durch das crypt modul verschluesselt wurden

Parameter:

Vector *code enthält den zu sendenen code entweder die zahl der erfolgreich gesendeten byte oder 0 wenn nicht erfolgreich

3.4.2.8 int OurEthernet::sendEncData (Vector * *code*) `[private]`

sendet die verschlüsselten Daten an den Client um Ressourcen zu sparen wird hier in nur einer Zeile Code das Protocoll + Trennzeichen + Code Trennzeichen verschlüsselt und an den anderen Agent gesendet um dann die Anzahkl der gesendeten Bytes zurück zu geben wobei die Trennzeichen nicht verschlüsselt werden

Parameter:

Vector ∗code enthält den zu sendenen code

Rückgabe:

die Menge der erfolgreich gesendeten Bytes

3.4.2.9 int OurEthernet::sendIR (Vector ∗ *code*)

kümmert sich um den übertragunsweg und stößt das Senden an Ueberprüft ob bereits eine Verbidnung zu einem anderen Agenten besteht, wenn ja wird das senden sofort ausgeführt, wenn nicht wird eine Verbindung aufgebaut und dann das senden gestartet

Parameter:

Vector ∗code enthält den zu sendenen code

Rückgabe:

Zahl der gesendeten byte oder 0 wenn nicht erfolgreich

3.4.2.10 int OurEthernet::startClient (byte ∗ *ip*, int *port*)

startet den Client erzeugt und initialisiert das Client Objekt um dann eine Verbindung zu einem Server aufzubauen

Parameter:

byte ∗ip ip adresse zu der eine Verbindung aufgebaut werden soll

int port port nummer zu der eine Verbindung aufgebaut werden soll

Rückgabe:

gibt 1zurueck wenn erfolgreich, sonst 0

3.4.2.11 int OurEthernet::startServer (int *port*)

startet den Server erzeugt und initialisiert den server auf dem listenport port zu horchen

Parameter:

int port portnummer auf der der listenport des Servers liegt

Rückgabe:

1 wenn erfolgreich, sonst 0

3.4.3 Dokumentation der Datenelemente

3.4.3.1 Authentification* OurEthernet::auth [private]

3.4.3.2 EthernetClient* OurEthernet::client [private]

3.4.3.3 Crypt* OurEthernet::crypt [private]

3.4.3.4 String* OurEthernet::password [private]

3.4.3.5 int OurEthernet::port [private]

3.4.3.6 EthernetServer* OurEthernet::server [private]

3.4.3.7 byte* OurEthernet::targetIP [private]

3.4.3.8 EthernetClient OurEthernet::tmpclient [private]

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- arina/ourEthernet.h
- arina/ourEthernet.cpp

3.5 Properties Klassenreferenz

`#include <properties.h>`

Öffentliche Methoden

- Properties ()
  Konstruktor.
- ~Properties ()
  Destruktor.
- String * getLocalPassword ()
  Gibt das Password zurück.
- byte * getRemoteIP ()
  Gibt die ZielIP zurück.
- byte * getLocalIP ()
  Gibt Eigene IP zurück.
- byte * getLocalNM ()
  Gibt Netzwerkmaske zurück.
- byte * getLocalGW ()
  Gibt Gateway zurück.
- byte * getLocalDNS ()
  Gibt DNSServer zurück.
- byte * getLocalMAC ()
  Gibt MAC Adresse zurück.
- void getSettings ()
  ließt die Einstellungen von der SD-Karte aus
- int getLocalPort ()
  Gibt Port zurück.
- int getRemotePort ()
  Gibt Port zurück.
- int getPort ()
  gibt Port zurück

Private Methoden

- void intToByte (unsigned int *source, byte *target, int length)
  Konvertiert unsigned int array in byte array.

Private Attribute

- File myConfig
- int port
- byte locMac [6]
- byte remIP [4]
- byte locNM [4]
- byte locGW [4]
- byte locDNS [4]
- byte locIP [4]
- String locPW
- String remPW

3.5.1 Beschreibung der Konstruktoren und Destruktoren

3.5.1.1 Properties::Properties ()

Konstruktor. Initialisiert die SD-Karte für Lesezugriff

3.5.1.2 Properties::~Properties ()

Destruktor.

3.5.2 Dokumentation der Elementfunktionen

3.5.2.1 byte * Properties::getLocalDNS ()

Gibt DNSServer zurück.

Rückgabe:

Pointer auf locDNS

3.5.2.2 byte * Properties::getLocalGW ()

Gibt Gateway zurück.

Rückgabe:

Pointer auf gateway

3.5.2.3 byte * Properties::getLocalIP ()

Gibt Eigene IP zurück.

Rückgabe:

Pointer auf myIP

3.5.2.4 byte * Properties::getLocalMAC ()

Gibt MAC Adresse zurück.

Rückgabe:

Pointer auf mac

3.5.2.5 byte * Properties::getLocalNM ()

Gibt Netzwerkmaske zurück.

Rückgabe:

Pointer auf netmask

3.5.2.6 String * Properties::getLocalPassword ()

Gibt das Password zurück.

Rückgabe:

Pointer auf Password

3.5.2.7 int Properties::getLocalPort ()

Gibt Port zurück.

Rückgabe:

Pointer auf port

3.5.2.8 int Properties::getPort ()

gibt Port zurück

Rückgabe:

Portnummer

3.5.2.9 byte * Properties::getRemoteIP ()

Gibt die ZielIP zurück.

Rückgabe:

Pointer auf ZielIP

3.5.2.10 int Properties::getRemotePort ()

Gibt Port zurück.

Rückgabe:

Pointer auf port

3.5.2.11 void Properties::getSettings ()

ließt die Einstellungen von der SD-Karte aus Einstellungen aus der "settings.txt" auf der SD-Karte werden ausgelesen

3.5.2.12 void Properties::intToByte (unsigned int * *source*, byte * *target*, int *length*) [private]

Konvertiert unsigned int array in byte array.

Parameter:

unsigned int *source Quellarray

byte *target Zielarray

int length Länge der arrays

3.5.3 Dokumentation der Datenelemente

3.5.3.1 byte Properties::locDNS[4] [private]

3.5.3.2 byte Properties::locGW[4] [private]

3.5.3.3 byte Properties::locIP[4] [private]

3.5.3.4 byte Properties::locMac[6] [private]

3.5.3.5 byte Properties::locNM[4] [private]

3.5.3.6 String Properties::locPW [private]

3.5.3.7 File Properties::myConfig [private]

3.5.3.8 int Properties::port [private]

3.5.3.9 byte Properties::remIP[4] [private]

3.5.3.10 String Properties::remPW [private]

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- arina/properties.h
- arina/properties.cpp

3.6 remote Klassenreferenz

`#include <irremote.h>`

Öffentliche Methoden

- void sendIR (Vector *data)

  übergibt IR-Signale aus einem Vector an die IRRemote-Bibliothek.

- Vector * recvIR ()

  erzeugt vector zur Übertragung per Netzwerk

- remote (int)

  erzeugt einen localReciever der IR-Bibliothek

- ~remote ()

Private Methoden

- int detectRAWData ()

Private Attribute

- IRrecv * localReciever

3.6.1 Beschreibung der Konstruktoren und Destruktoren

3.6.1.1 remote::remote (int *recvPIN*)

erzeugt einen localReciever der IR-Bibliothek erzeugt einen localReciever der IR-Bibliothek mit einem gewählten Input-Pin

3.6.1.2 remote::~remote ()

3.6.2 Dokumentation der Elementfunktionen

3.6.2.1 int remote::detectRAWData () `[private]`

3.6.2.2 Vector * remote::recvIR ()

erzeugt vector zur Übertragung per Netzwerk initialisiert einen zu übertragenden vector, entsprechend des Übertragungsprotokolls

Rückgabe:

vector *data enthält die eingelesenen Daten in protokollform

3.6.2.3 void remote::sendIR (Vector * *data*)

übergibt IR-Signale aus einem Vector an die IRRemote-Bibliothek. initialisiert eine IRsend-Klasse und übergibt die per Vektor übergebenen Befehle entsprechend

Parameter:

vector *data enthält die Daten in protokollform

3.6.3 Dokumentation der Datenelemente

3.6.3.1 IRrecv* remote::localReciever `[private]`

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- arina/irremote.h
- arina/irremote.cpp

3.7 Return Klassenreferenz

`#include <return.h>`

Öffentliche Methoden

- Return ()
- ~Return ()
- String * createReturnMessage (boolean worked)

3.7.1 Beschreibung der Konstruktoren und Destruktoren

3.7.1.1 Return::Return () [inline]

3.7.1.2 Return::~Return () [inline]

3.7.2 Dokumentation der Elementfunktionen

3.7.2.1 String* Return::createReturnMessage (boolean *worked*)

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Datei:

- arina/return.h

3.8 Vector Klassenreferenz

`#include <vector.h>`

Öffentliche Methoden

- Vector ()
  Konstruktor.
- ∼Vector ()
  Destruktor.
- void push_back (byte add)
  Fügt neues Byte hinten an.
- unsigned int size ()
  Gibt die Größe des Arrays zurück.
- byte ∗ at (unsigned int i)
  Zugriff auf einzelne Positionen des Vektors.

Private Attribute

- byte ∗ vector
- unsigned int vectorSize

3.8.1 Beschreibung der Konstruktoren und Destruktoren

3.8.1.1 Vector::Vector ()

Konstruktor. Initialisiert die Vriablen

3.8.1.2 Vector::∼Vector ()

Destruktor. Löscht das angelegte Array und gibt so den reservierten Speicher wieder frei

3.8.2 Dokumentation der Elementfunktionen

3.8.2.1 byte ∗ Vector::at (unsigned int *i*)

Zugriff auf einzelne Positionen des Vektors. Gibt einen Pointer auf die angefragte Position im Array zurück

Parameter:

unsigned int i Stelle des Vektors

Rückgabe:

Pointer auf das gefragte byte

3.8.2.2 void Vector::push_back (byte *data*)

Fügt neues Byte hinten an. Reserviert neuen Speicher, der so groß ist wie der alte plus eins Kopiert dann den Inhalt des alten arrays in das neue, und fügt hinten das neue Byte an Dann wird noch der Speicherplatz vom alten Array freigegeben.

Parameter:

byte data Byte das hinten angefügt werden soll

3.8.2.3 unsigned int Vector::size ()

Gibt die Größe des Arrays zurück.

Rückgabe:

Größe des Arrays

3.8.3 Dokumentation der Datenelemente

3.8.3.1 byte* Vector::vector `[private]`

3.8.3.2 unsigned int Vector::vectorSize `[private]`

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- arina/vector.h
- arina/vector.cpp

Kapitel 4

Datei-Dokumentation

4.1 arina/arina.ino-Dateireferenz

```
#include <Arduino.h>
#include <util.h>
#include <Dns.h>
#include <Dhcp.h>
#include <Ethernet.h>
#include <EthernetServer.h>
#include <EthernetUdp.h>
#include <EthernetClient.h>
#include <SPI.h>
#include <SD.h>
#include <ctype.h>
#include <IRremote.h>
#include "properties.h"
#include "ledstate.h"
#include "ourEthernet.h"
#include "irremote.h"
#include "vector.h"
```

Include-Abhängigkeitsdiagramm für arina.ino:

Funktionen

- void setup ()
  Initialisierung.
- void loop ()
  Main loop.

Variablen

- Properties * prop
- OurEthernet * oureth
- LEDState * led
- remote * irrem
- int irReadCounter = 0
- Vector * result = NULL

4.1.1 Dokumentation der Funktionen

4.1.1.1 void loop ()

Main loop. Fragt abwechselnd IR Empfänger und Netzwerk ab um dann entsprechend zu reagieren

4.1.1.2 void setup ()

Initialisierung. Initialisiert Serial, LED, Properties, OurEthernet und Remote

4.1.2 Variablen-Dokumentation

4.1.2.1 int irReadCounter = 0

4.1.2.2 remote* irrem

4.1.2.3 LEDState* led

4.1.2.4 OurEthernet* oureth

4.1.2.5 Properties* prop

4.1.2.6 Vector* result = NULL

4.2 arina/authentification.cpp-Dateireferenz

```
#include "authentification.h"
```

Include-Abhängigkeitsdiagramm für authentification.cpp:

4.3 arina/authentification.h-Dateireferenz

Verarbeitet die Authentifikation. `#include <Arduino.h>`

Include-Abhängigkeitsdiagramm für authentification.h:Dieser Graph zeigt, welche Datei direkt oder indirekt diese Datei enthält:

Datenstrukturen

- class Authentification

4.3.1 Ausführliche Beschreibung

Verarbeitet die Authentifikation.

Autor:

Thomas Trimborn

Datum:

30.08.13

Version:

0.1

Einfache implementierung, zur Zeit wird nur das Password ueberprueft

4.4 arina/crypt.cpp-Dateireferenz

```
#include "crypt.h"
```

Include-Abhängigkeitsdiagramm für crypt.cpp:

4.5 arina/crypt.h-Dateireferenz

implements a en and decryption funktion for kommunikation `#include <Arduino.h>`

Include-Abhängigkeitsdiagramm für crypt.h:Dieser Graph zeigt, welche Datei direkt oder indirekt diese Datei enthält:

Datenstrukturen

- class Crypt

4.5.1 Ausführliche Beschreibung

implements a en and decryption funktion for kommunikation

Autor:

Thomas Trimborn

Datum:

30.08.13

Version:

0.1

Has to be implemented in later work Now it just takes a byte array an gives it back

4.6 arina/irremote.cpp-Dateireferenz

```
#include "irremote.h"
```

Include-Abhängigkeitsdiagramm für irremote.cpp:

4.7 arina/irremote.h-Dateireferenz

bereitet die Daten zum Senden per Netzwerk bzw. Ausgabe per IR auf `#include <Arduino.h>`

`#include "vector.h"`

`#include <IRremote.h>`

Include-Abhängigkeitsdiagramm für irremote.h:Dieser Graph zeigt, welche Datei direkt oder indirekt diese Datei enthält:

Datenstrukturen

- class remote

4.7.1 Ausführliche Beschreibung

bereitet die Daten zum Senden per Netzwerk bzw. Ausgabe per IR auf

Autor:

Rene Neff

Datum:

10.10.13

Version:

1.0

überführt die empfangenen Befehle aus der IR Bibliothek in das definierte Protokollformat und übergibt Befehle einer empfangenen protokollkonformen Sequenz an die IR Bibliothek

4.8 arina/ledstate.cpp-Dateireferenz

```
#include "ledstate.h"
```

Include-Abhängigkeitsdiagramm für ledstate.cpp:

4.9 arina/ledstate.h-Dateireferenz

Kümmert sich um das an und abschalten von LED's. `#include <Arduino.h>`

Include-Abhängigkeitsdiagramm für ledstate.h:Dieser Graph zeigt, welche Datei direkt oder indirekt diese Datei enthält:

Datenstrukturen

- class LEDState

4.9.1 Ausführliche Beschreibung

Kümmert sich um das an und abschalten von LED's.

Autor:

Thomas Trimborn

Datum:

30.08.13

Version:

1.0

Diese Klasse kümmert sich um das ansteuern der LED's, die im Konstruktor in beliebiger Anzahl definiert werden können Der Konstruktor muss mit mehreren Argumenten Aufgerufen werden: Zuerst die Anzahl der LED's als Integer Dann die Pins an denen die LED's haengen, auch als Integer Jeweils durch Kommata getrennt

4.10 arina/ourEthernet.cpp-Dateireferenz

```
#include "ourEthernet.h"
```

Include-Abhängigkeitsdiagramm für ourEthernet.cpp:

4.11 arina/ourEthernet.h-Dateireferenz

Zuständig für das senden und empfangen über ethernet. `#include <Arduino.h>`

```
#include <util.h>
#include <Dns.h>
#include <Dhcp.h>
#include <Ethernet.h>
#include <EthernetServer.h>
#include <EthernetUdp.h>
#include <EthernetClient.h>
#include <SPI.h>
#include "authentification.h"
#include "crypt.h"
#include "return.h"
#include "vector.h"
```

Include-Abhängigkeitsdiagramm für ourEthernet.h:Dieser Graph zeigt, welche Datei direkt oder indirekt diese Datei enthält:

Datenstrukturen

- class OurEthernet

Makrodefinitionen

- #define TRENNZEICHEN ';'

4.11.1 Ausführliche Beschreibung

Zuständig für das senden und empfangen über ethernet.

Autor:

Thomas Trimborn

Datum:

10.10.13

Version:

1.0

Kann listen ports aufmachen und sich als client zu listenports verbinden Ausserdem können verbindungen authentifiziert sowie verschlüsselt genutzt werden um protocoll und code daten zu uebertragen sowie um antworten zurueck zu schicken

4.11.2 Makro-Dokumentation

4.11.2.1 #define TRENNZEICHEN ';'

4.12 arina/properties.cpp-Dateireferenz

```
#include "properties.h"
```

Include-Abhängigkeitsdiagramm für properties.cpp:

4.13 arina/properties.h-Dateireferenz

Auslesen von Einstellungen auf der SD-Karte. `#include <SD.h>`

`#include <ctype.h>`

`#include <Arduino.h>`

Include-Abhängigkeitsdiagramm für properties.h:Dieser Graph zeigt, welche Datei direkt oder indirekt diese Datei enthält:

Datenstrukturen

- class Properties

4.13.1 Ausführliche Beschreibung

Auslesen von Einstellungen auf der SD-Karte.

Autor:

Rene Neff

Datum:

08.10.13

Version:

1.0

"settings.txt" auf der SD-Karte liefern Einstellungen zum Betrieb

4.14 arina/return.cpp-Dateireferenz

```
#include "return.h"
```

Include-Abhängigkeitsdiagramm für return.cpp:

Funktionen

- String ∗ createReturnMessage (boolean worked)
  Erzeugt aus den gegebenen Daten eine Rueckgabenachricht

4.14.1 Dokumentation der Funktionen

4.14.1.1 String∗ createReturnMessage (boolean *worked*)

Erzeugt aus den gegebenen Daten eine Rueckgabenachricht∗

Parameter:

boolean worked Ob das empfangen und ausgeben der IR Nachricht funktioniert hat ∗

Rückgabe:

Pointer zum String mit den angaben

4.15 arina/return.h-Dateireferenz

Erzeugt, was zum anderen Sender zurueck gegeben werden soll. `#include <Arduino.h>`

Include-Abhängigkeitsdiagramm für return.h:Dieser Graph zeigt, welche Datei direkt oder indirekt diese Datei enthält:

Datenstrukturen

- class Return

4.15.1 Ausführliche Beschreibung

Erzeugt, was zum anderen Sender zurueck gegeben werden soll.

Autor:

Thomas Trimborn

Datum:

30.08.13

Version:

0.1

simple implementierung, da bisher kein rueckkanal eingerichtet ist

4.16 arina/vector.cpp-Dateireferenz

```
#include "vector.h"
```

Include-Abhängigkeitsdiagramm für vector.cpp:

4.17 arina/vector.h-Dateireferenz

implements a small version of a vector class `#include <Arduino.h>`

Include-Abhängigkeitsdiagramm für vector.h:Dieser Graph zeigt, welche Datei direkt oder indirekt diese Datei enthält:

Datenstrukturen

- class Vector

4.17.1 Ausführliche Beschreibung

implements a small version of a vector class

Autor:

Thomas Trimborn

Datum:

09.10.13

Version:

1.1

Just a few small functions to implement a vector

Index