Masterarbeit

# $ECO_2$-Manager

Mader Andreas, BSc

Institut für Technische Informatik
Technische Universität Graz
Vorstand: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß


Begutachter: O. A.o.Univ.-Prof. Dipl.-Ing. Dr. techn. Eugen Brenner
Betreuer: O. A.o.Univ.-Prof. Dipl.-Ing. Dr. techn. Eugen Brenner

Graz, im Mai 2012

„Unser größtes Problem ist, dass wir uns nicht mehr vorstellen, was wir anstellen."

Gunther Anders, Philosoph

„Es ist billiger den Planeten jetzt zu schützen, als ihn später zu reparieren."

José Manuel Barroso, Kommissionspräsident

„Die katastrophalen Folgen des Klimawandels müssen sichtbar gemacht, das heißt, sie müssen wirkungsvoll inszeniert werden, damit Handlungsdruck erzeugt wird."

Ulrich Beck

## Kurzfassung

Da in den letzten Jahren der Ausstoß von $CO_2$ immer mehr zum Thema wurde, entstand die Idee, eine $CO_2$-Messung für einen Haushalt zu realisieren. Dabei sollen diverse Werte, die den $CO_2$-Ausstoß im Haushalt bestimmen, gemessen und dargestellt werden, wodurch eine Visualisierung des Ausstoßes erreicht wird. Das Ziel könnte auch eine Regulierung der $CO_2$-Emissionen sein, entweder durch Anreize beim Einsparen von $CO_2$ oder auch durch Strafen für den erhöhten Ausstoß. Als Vorbild sind hier sicherlich die Klimazertifikate für die Industrie, die von der Europäischen Union vergeben werden, zu nennen.

In dieser Arbeit wird eine Auswertungseinheit für diese $CO_2$-Messung realisiert und dokumentiert, wobei das Hauptaugenmerk auf die Speicherung und Visualisierung der Daten mittels Embedded System gelegt wird. Die Daten können dabei in einfacher Form am Touchscreen direkt an der Einheit angezeigt werden oder genauer am PC, wo dies durch eine Java-Anwendung ermöglicht wird, welche per Ethernet mit dem Embedded System der Auswertungseinheit kommuniziert. Ein Auslesen von Datenbankdateien und die Einstellung der verschiedenen Sensoren wird ebenfalls über die Java-Anwendung ermöglicht.

Ein weiterer Teil der Arbeit soll die Ansteuerung von Aktoren auf den Sensorknoten ermöglichen, um einfache Schaltvorgänge über den bereits vorhandenen Kommunikationsweg ausführen zu können. Die Kommunikation wurde dabei schon im Masterprojekt [Mad11] behandelt und wird hier in die Software integriert und erweitert.

Als Prototyp wird die Auswertungseinheit auf einem ChipworkX-Development-System von GHI Electronics in C# mit Hilfe des .NET Micro Framework programmiert. Es stehen verschiedene Speichermöglichkeiten (intern, USB-Stick, SD-Karte) und eine Ethernet-Schnittstelle zur Verfügung. Über diese Schnittstelle werden Daten gesammelt und es können Daten darüber angefordert werden, welche am PC weiter verarbeitet werden. Bei der Datenübertragung wurde auf eine robuste und vor allem schnelle Übertragung geachtet, da es hier passieren kann, dass sehr viele Daten auf einmal angefordert werden und das Embedded System ist nicht besonders schnell beim Versenden von Daten. Die Anbindung von Smart-Metern wurde ebenfalls hinzugefügt, wobei diese über die vorhandenen Wege angeschlossen werden können oder mit USB-Umsetzern über eine optische Schnittstelle, über RS232 oder RS485. Das Protokoll dieser Smart-Meter wurde genau dokumentiert, da hier nur eine mangelhafte Dokumentation vorhanden ist.

## Abstract

In the last few years the $CO_2$ emissions where part in many political and economical discussions. Due to this, the idea to monitor the $CO_2$-Emissions of a household was born. Various data about factors, which influence the carbon dioxide emissions, are collected and visualised, to demonstrate the consumers, how much carbon dioxide they produce. The goal could be a regulation of the carbon dioxide emissions for every household, which could be done by financial incentives, if the amount of $CO_2$ emissions is less then a given limit, or a penalty, if the amount is exceeded. The climate-certificates given to the industry by the European Union are a good example for this, this model could also be applied to private households.

In this masterthesis a prototype of the $Eco_2$-Manager for $CO_2$ measurement is realized and documented. The main focus is the storage and visualisation of the collected data with an Embedded System. The data can be shown in a simple way on the touchscreen on the unit itself or on the pc, where a Java program can read the data via a network connection from the $Eco_2$-Manager. It is also possible to read a database file and set the sensors on the manager with this Java application.

Another chapter of the masterthesis enables the possibility to control some actors on the sensorboards which are connected via Ethernet or wireless XBee connection. The communication of the sensors and actors is also discussed in the masterproject [Mad11] and the existing software will be integrated and expanded in this thesis. Many improvements are made and new communication channels are added.

As a protype, the $Eco_2$-Manager is developed on a ChipworkX-Development-System from GHI Electronics in C# with the .NET Micro Framework. There are different ways to store the data (internal, USB flash drive, SD card) and an ethernet interface is available. Via this interface data can be gathered, but it is also possible to request data for calculations or visualisation on the PC. The communication is designed reliable and especially fast, because it is possible that many data is aquired at once, and the embedded system is slow sending ethernet packets. There is also a new connection added to aquire data from Smart-Meters. They can be connected via the existing interfaces or an optical interface, RS232 or RS485 via a USB powered converter. The protocol to communicate with the Smart-Meters is also documented, because there is only a faulty documentation available.

## EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz,am ..... .....

(Unterschrift)

# Danksagung

Ich möchte mich an dieser Stelle bei den Personen bedanken, die mich während meinem Studium und bei der Erstellung dieser Masterarbeit unterstützt haben. Mein besonderer Dank gilt **meinen Eltern Friedrich und Karin Mader**, die mir nicht nur das Studium erst ermöglicht haben, sondern auch immer großes Interesse für meine Arbeit zeigten und mich immer unterstützten. Ohne sie wäre mein Studium in dieser Form nicht möglich gewesen.

Ebenfalls bedanken möchte ich mich bei **Univ.-Prof. Dipl.-Ing. Dr.techn. Eugen Brenner** für die Möglichkeit, diese Diplomarbeit am Institut für Technische Informatik durchführen zu können. Vielen Dank für die freundliche und unkomplizierte Unterstützung und Betreuung während der gesamten Dauer der Arbeit.

Des Weiteren möchte ich noch allen **Freunden und Kollegen** aus meiner Heimat und aus Graz für die Geduld und die gemeinsamen Stunden während meinem Studium danken. Ein Ausgleich neben dem Studium ist für mich sehr wichtig, diesen habe ich in den Stunden mit euch und im Dienste des **Roten Kreuzes** gefunden.

Herzlichen Dank für alles!

Graz, im Mai 2012 Mader Andreas, BSc

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Kapitel 1.

# Einleitung

## 1.1. Motivation

In Zeit von Immissionsschutzgesetz-Luft (IGL), Geschwindigkeitsbegrenzungen auf Autobahnen und politischen Diskussionen über die Reduktion von Feinstaub bzw. anderen Belastungen wird immer wieder über Klimazertifikate diskutiert. Neue Autos dürfen einen bestimmten $CO_2$-Ausstoß pro Kilometer nicht überschreiten, dies wird von der Europäischen Union seit dem Jahr 2009 vorgeschrieben. Dabei soll eine Reduzierung des Ausstoßes von 30 Prozent vom Jahr 1990 bis zum Jahr 2020 erreicht werden, wobei derzeit hauptsächlich auf die Reduzierung im Straßenverkehr und der Industrie Wert gelegt wird. [Eur09]

In der Europäischen Union wurde auch bereits ein Emissionshandel eingerichtet, welcher derzeit hauptsächlich die Industrie betrifft, genannt Emission Trading System (ETS). Dabei wurde festgelegt, wie viel Emissionszertifikate, und damit die Erlaubnis Treibhausgase auszustoßen, den Unternehmen zur Verfügung gestellt werden. Die gesamte Anzahl an Zertifikaten wird von der EU an die einzelnen Staaten vergeben, diese verteilen sie auf die einzelnen Unternehmen im Land. Ab 2013 werden die Zertifikate von der Europäischen Kommission für alle Unternehmen der EU vergeben. Privathaushalte werden dort derzeit noch nicht erfasst.

Da es in Zukunft möglich bzw. wahrscheinlich ist, dass auch Privathaushalte von einer solche Regelung betroffen sind, kam die Idee, eine $CO_2$-Messung für einen Haushalt zu realisieren. Dabei sollen diverse Einflussfaktoren, wie Strom-, Heizöl- oder Erdgasverbrauch einbezogen werden und über diese der tatsächliche Ausstoß an $CO_2$ des Haushalts berechnet werden. Ebenso können umweltfreundliche Technologien, wie die Produktion von Ökostrom durch Photovoltaikanlagen, positiv bei der Berechnung des Kohlendioxidhaushaltes einberechnet werden. Um den eigenen Ausstoß immer im Blick zu haben wird ein System entwickelt, welches diese Faktoren laufend misst und darstellt - mit der zusätzlichen Option eine gewisse Menge als *Warngrenze* einzugeben bzw. auch den Rest darstellen zu können, der noch verbraucht werden darf.

In Zukunft sind Strafen für zu viel Kohlendioxidausstoß oder, in umgekehrter Richtung, Zuschüsse für weniger Ausstoß durchaus denkbar - dazu muss aber der Verbrauch zuerst erfasst werden. Ein Ausgleich des eigenen Ausstoßes kann sogar jetzt schon durch sogenannte Klimazertifikate (auch Klima-Aktie oder Klima-Schutzbrief genannt) "zurückgekauft" werden. Kauft man solche Zertifikate wird das Geld für umweltfreundliche Technologien zur Energiegewinnung (Wasserkraftwerke, Solarkraftwerke, Windkraftwerke,...) investiert oder auch für die Aufforstung von Wäldern verwendet. Auch bewusste Einsparungen von $CO_2$-Emissionen sind möglich, wenn man die Anteile diverser Einflussfaktoren betrachten

und minimieren kann. [Net11]

## 1.2. Zielsetzung

Im Zuge dieser Masterarbeit wird eine Auswertungseinheit ($ECO_2$-Manager) für eine $CO_2$-Messung in einem Haushalt entworfen und dokumentiert. Ein Embedded System Developmentboard (ChipworkX Development System v1.2 [GHI10b]) soll dabei als Hardwareplattform dienen, da dieses Board die für die Entwicklung nötige Peripherie enthält:

- einen Touchscreen,
- USB-Anschlüsse,
- einen SD-Karten-Slot,
- einen Ethernet-Anschluss,
- einen Steckplatz für XBee-Pro Module,
- eine Echtzeituhr,
- uvm.

Die eigentliche Hardware, wie sie später eventuell produziert wird, ist nicht Teil der Arbeit, diese kann auf Basis dieses Systems entwickelt werden. Auf Buttons wird verzichtet, da das System über den Touchscreen vollständig bedienbar ist und dadurch eine kleinere Bauform ermöglicht wird. Bei der Entwicklung wird zudem Wert auf eine einfache Bedienbarkeit gelegt - die erstmalige Einrichtung der Sensoren und Aktoren muss über den PC erfolgen - der Rest kann am System selbst eingestellt werden.

Programmiert wird das Embedded System in C#, wobei das **.NET Micro Framework** verwendet wird, welches bereits sehr viele Funktionen und Module für eine einfache Ansteuerung der Hardware enthält. Die Portierung auf andere Hardware wird dadurch ebenfalls erheblich erleichtert, sofern diese durch das Framework unterstützt wird. Am Markt tauchen inzwischen immer mehr Devices auf, welche mit dem .NET Micro Framework betrieben werden können. Wie das Framework genau aufgebaut ist, wird in Kapitel 2 genauer beschrieben.

Im Vordergrund des Projektes steht auch die Speicherung und Darstellung von Daten, die von Sensoren gesendet werden, wahlweise auf SD-Karte, USB-Stick oder im internen Speicher. Die Daten werden von den Sensoren per **Ethernet** oder **XBee** zur Auswertungseinheit geschickt, was bereits im Masterprojekt [Mad11] behandelt wurde und in dieser Arbeit verbessert und erweitert wird. Als weiteres Beispiel für die Vielseitigkeit der Sensoranbindung werden Smart-Meter der Firma Luna per USB angebunden und ausgelesen. Die Daten können anschließend in einfacher Form am Display ausgewertet werden oder genauer per Client-Programm am PC betrachtet werden. Dieses Programm wird in Java am PC entwickelt und kommuniziert über Ethernet mit dem $Eco_2$-Manager.

Die Ansteuerung von Aktoren soll ebenfalls über den Touchscreen in einfacher Art und Weise möglich sein. Jeder Aktor, der zuvor am PC erfasst wurde, kann über den Touchscreen aktiviert oder deaktiviert werden, sofern eine aktive Verbindung zu diesem besteht. Auch zum Beispiel Pulsweitenmodulation (PWM)-Ausgänge bzw. das Senden von einfachen ganzzahligen Werten kann mit der gewählten Kommunikationsschnittstelle übertragen werden. Am Touchscreen werden derzeit nur Schaltvorgänge ermöglicht, eine Umstellung auf dimmbare Regler ist bei Bedarf nur im Frontend umzusetzen, das Backend unterstützt dies bereits.

Die Vorgabe eines monatlichen bzw. jährlichen Limits ist ebenfalls möglich, inklusive Restanzeige, wie viel $CO_2$ noch verbraucht werden darf um unter der voreingestelltn Grenze zu bleiben. Die Grenze kann in *Kilogramm Kohlendioxid (kg$CO_2$)* angegeben und jederzeit angepasst werden. So wäre zum Beispiel eine Zuweisung einer gewissen Menge an $CO_2$ für einen Haushalt möglich und der Verbrauch kann für den Besitzer transparent dargestellt werden.

## 1.3. Gliederung

Am Beginn dieser Arbeit wird die verwendete Hardware für die Auswertungseinheit und das Framework erklärt, um Einsicht in die Funktionalität zu bekommen. Dabei werden auch besondere Konstrukte, die in der Arbeit später vorkommen, erklärt, um die Weiterentwicklung zu vereinfachen. Des Weiteren wird der Grundaufbau des Programmes und das zugehörige Graphical Userinterface erklärt. Auch hier wurde Wert auf Erweiterbarkeit gelegt, um für zukünftige Entwicklungen vorzusorgen.

Kapitel 4 beschreibt, wie das Management der Datenspeicherung umgesetzt wurde. Das Verwalten der Datenbanken, der Datenbankverbindungen und der Massenspeichermedien steht hier im Vordergrund. Die Datenbanken stellen einen zentralen Punkt dar, da sie sämtliche gesammelten Daten beinhalten und verwalten.

Im Kapitel über die Änderungen der Sensoranbindung werden die Anpassungen ausgehend vom Masterprojekt [Mad11] beschrieben, die durchgeführt wurden. Vor allem Optimierungen und Erweiterungen werden durchgeführt, um eine reibungslose Kommunikation zu gewährleisten. Gänzlich hinzugefügt wird das Senden und Empfangen von Daten via Ethernet, um mit dem $Eco_2$-Client (Parametrisierung und Visualisierung am PC) kommunizieren zu können.

Der $Eco_2$-Client selbst wird in einem eigenen Kapitel beschrieben. Hier stehen verschiedene Möglichkeiten zur Verfügung - Lesen der Daten über Ethernet oder aus einer Datenbankdatei und die Darstellung in verschiedenen Formen. Es können verschiedene Diagrammtypen gewählt werden, um eine möglichst übersichtliche Auswertung zu erreichen.

In Kapitel 7 wird die neu hinzugefügte Anbindung von Smart-Metern beschrieben. Im Speziellen werden Geräte der Firma Luna angebunden und deren Protokoll dokumentiert, da die vorhandene Dokumentation fehlerhaft ist. Ebenso wurde eine Übersetzung der Parametertabelle und der Bedienungsanleitung versucht, um die Funktion des Smart-Meters verstehen zu können, die Originale sind nur in türkischer Sprache erhältlich. Der genaue Ablauf des Auslesens eines Smart-Meters wird übersichtlich dargestellt und umgesetzt.

Das vorletzte Kapitel soll vor allem für die weitere Entwicklung hilfreich sein. Es beschreibt den verwendeten Codingstandard, welcher bestimmte Richtlinien vorgibt, die helfen sollen, zum Beispiel verschiedene Datentypen (Klassenmember, Konstanten...) zu unterscheiden. Javadoc und C#-XML-Dokumentation sollen eine Hilfe beim Verwenden der vorhandenen Funktionen geben und die verwendeten Debugging-Hilfen werden beschrieben, um diese weiter sinnvoll verwenden zu können.

# Kapitel 2.

# .NET Micro Framework / verwendete Hardware

## 2.1. Aufbau des .NET Micro Framework

Das .NETMF wurde früher bei Microsoft unter dem Namen *Smart Personal Objects Technology (SPOT)* geführt, was sich in manchen Core-Librarys heute noch im Namen wiederfindet. Es stellt kein traditionelles Betriebssystem mit gemanagtem Code oder eine virtuelle Maschine dar, das Resourcenmanagement wird zwar ähnlich wie bei einem Betriebssystem ausgeführt, ansonsten werden aber eher alternative Funktionen und Objekte angeboten, die helfen, die verwendete Hardware anzusprechen. Einen Überblick über die grobe Struktur des Frameworks bietet die Abbildung 2.1.

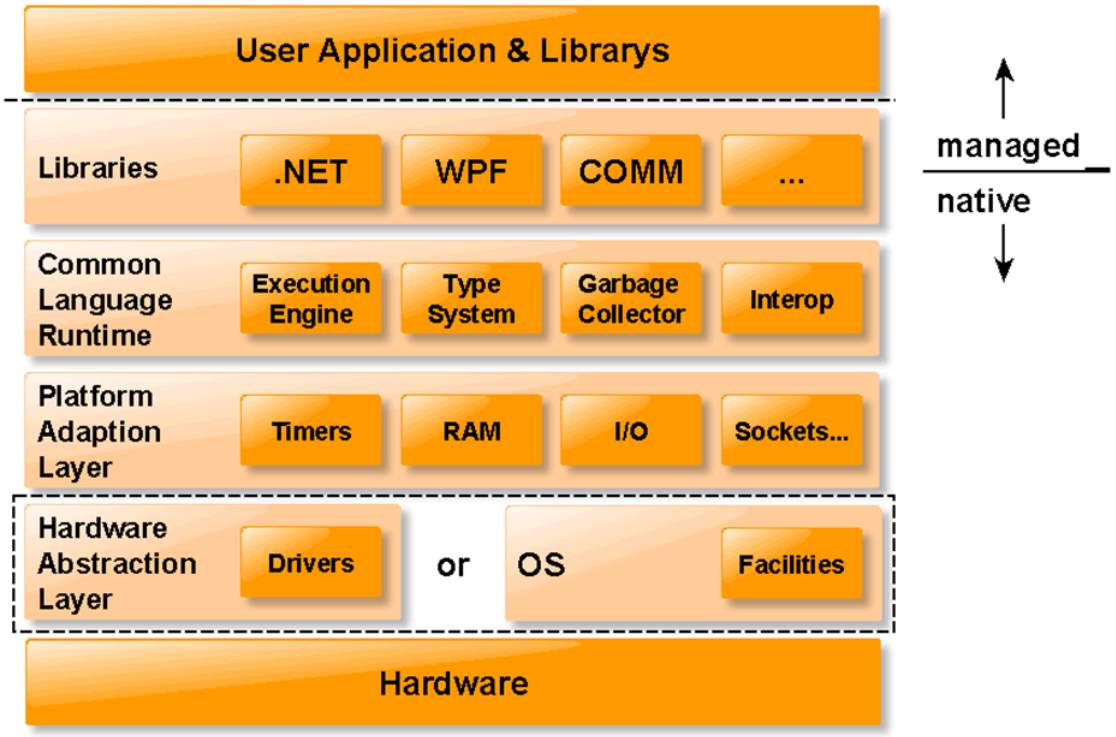



Abbildung 2.1.: Hierarchischer Aufbau .NET Micro Framework [DT07]

Im Gegensatz zu Programmen, die für Betriebssysteme (zum Beispiel Windows CE oder Windows XP Embedded) entwickelt wurden, können Programme, die mit dem .NETMF entwickelt wurden, direkt auf der Hardware ausgeführt werden und benötigen kein vorinstalliertes Betriebssystem. Das .NETMF wird dank dieser Eigenschaften auch als *Bootable Framework* bezeichnet, damit können Ressourcen eingespart und die Hardware günstiger gestaltet werden. Der Hardware Abstraction Layer (HAL), welcher im Framework integriert ist, stellt die entsprechenden Funktionen für die oberen Schichten zur Verfügung.

Sehr eng mit dem HAL arbeitet das Common Language Runtime (CLR) zusammen, welches den eigentlich Kern des Frameworks bildet. Teilweise geschieht diese Zusammenarbeit direkt, zum Teil aber auch über den entsprechenden Platform Adaption Layer (PAL). Der PAL wird speziell auf das verwendete System zugeschnitten, damit Timer, RAM, I/O-Ports und dergleichen für das jeweilige System optimal verwaltet und verwendet werden. Jedes .NET Framework bietet auch seine eigene Implementation des CLR, welches nicht compilierten Code ausführt, sondern *Code interpretiert*. Die Geschwindigkeitseinbußen, die durch den Interpreter entstehen, können allerdings durch die *Interop-Technik* wieder wett gemacht werden, wodurch sich klassische Bibliotheken wie normale .NET Programmfunktionen aufrufen lassen, was durch Kapselung in sogenannten *Wrappern* funktioniert. Die vollständige Kontrolle durch den CLR geht dabei allerdings verloren, die Ausführung wird dann rein von der entsprechenden DLL-Datei übernommen, was fehleranfällig sein kann, da die Programmflusskontrolle nicht wie gewohnt fortgeführt wird. Für Systemlibrarys macht dies jedoch Sinn, da wesentliche Geschwindigkeitsvorteile erzielt werden können und diese im Vorhinein bereits ausgiebig getestet werden. [DT07]

Libraries in der obersten Schicht sind bereits von Microsoft in großer Menge enthalten, werden allerdings auch von Herstellern der speziellen Hardware geschrieben, um einen besseren Support für die eigene Hardware bieten zu können. Die verwendeten Module des Frameworks werden in der jeweiligen Dokumentation der Codeblöcke genauer beschrieben. Zusätzlich zu den Libraries von Microsoft, gibt es auch speziellere Libraries von *GHI Electronics*, die für das ChipworkX-Modul (siehe Abschnitt 2.2) programmiert wurden.

Für Echtzeitsysteme ist das .NETMF ebenfalls nicht gedacht. Speziell durch die Interpretation des Codes kann kein Echtzeitverhalten erzeugt werden, das Framework bietet allerdings Multithreading (auch ohne Unterstützung von Multithreading auf der Hardwareseite) bzw. auch Garbage Collection (GC), wofür ein eigener Algorithmus verwendet wird, der Arbeitsspeicher einspart und GC am Controller bzw. Embedded System Ressourcen schonend durchführen kann.

Das .NETMF ist auch nicht nur eine abgespeckte Version des .NET Frameworks, es wurden zum Teil Klassen in andere Libraries verschoben bzw. auch Methoden mit anderen Parametern versehen, um den Aufbau einfach und das Framework kompakt zu gestalten. [Jon07] Leider wurden zum Teil Methoden eingespart, die für ein Embedded System wichtig wären. So wurde zum Beispiel der *BitConverter* entfernt, welcher es ermöglichen würde, auf die einzelnen Bytes der Variablen und Strukturen zugreifen zu können. Der *BufferedStream*, welcher vor allem bei Übertragung via Ethernet oft sehr wichtig wäre, wurde ebenfalls eingespart. Leider werden diese Klassen sehr häufig auch auf Embedded Systems benötigt, stehen aber im Framework nicht zur Verfügung, der *BufferedStream* wurde daher als Code hinzugefügt und die Rohdaten der Variablen und Strukturen werden manuell gelesen. Um mit Pointern die Variablen direkt auslesen zu können, wird in

der Definition der Methode das Schlüsselwort *unsafe* eingefügt und der Sourcecode mit der Option */unsafe* compiliert. Ohne diese Schlüsselwörter werden Pointer in C# nicht akzeptiert.

Für dieses Projekt wird die Version 4.1 des .NETMF verwendet - die zur Zeit der Erstellung der Arbeit aktuellste Version. Eine Betaversion 4.2 kam gegen Ende auf den Markt, wobei es noch keine genauen Aussagen über die Änderungen gibt.

## 2.2. Aufbau der ChipworkX Development System Hardware

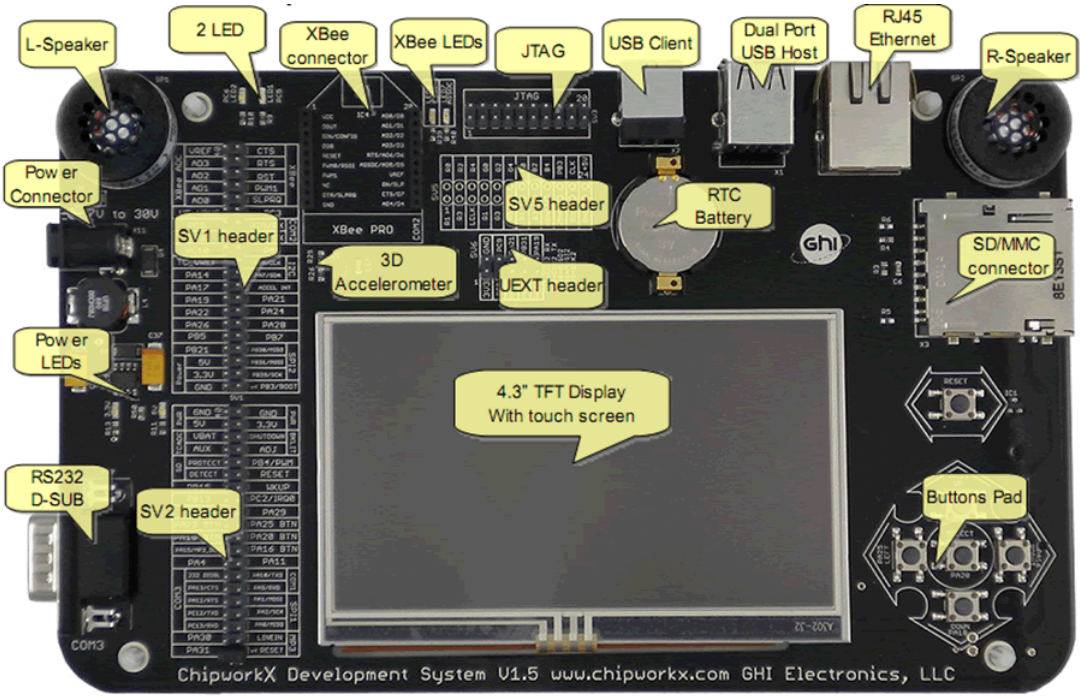



Abbildung 2.2.: ChipworkX Development System - Front View [GHI10a]

Das *ChipworkX Development-System* basiert auf dem *ChipworkX-Modul*, welches auf der Rückseite des Boards angebracht wird und führt diverse Hardwarepins als Steckplätze heraus bzw. bietet diverse Peripherie direkt am Board, um die Entwicklung zu erleichtern. Abbildung 2.2 und Abbildung 2.3 zeigen das Development-Board von vorne und hinten, die entsprechende Peripherie, die am Board vorhanden ist wurde direkt beschriftet. Speziell von Interesse für das Projekt $ECO_2$-Manager sind der XBee-Connector für die drahtlose Verbindung zu den Sensoren, der RJ45 Ethernet-Anschluss für die verdrahtete Kommunikation, USB-Steckplätze, SD/MMC Connector für die Speicherung der Daten und der Touchscreen für die Bedienung der Auswertungseinheit. Ebenso bietet das Board Debugging via USB - dies kann mit *Microsoft Visual Studio* oder *Microsoft Visual Studio Express* geschehen, letztere Version ist als kostenlose, allerdings etwas eingeschränkte Variante nach entsprechender Registrierung verfügbar.

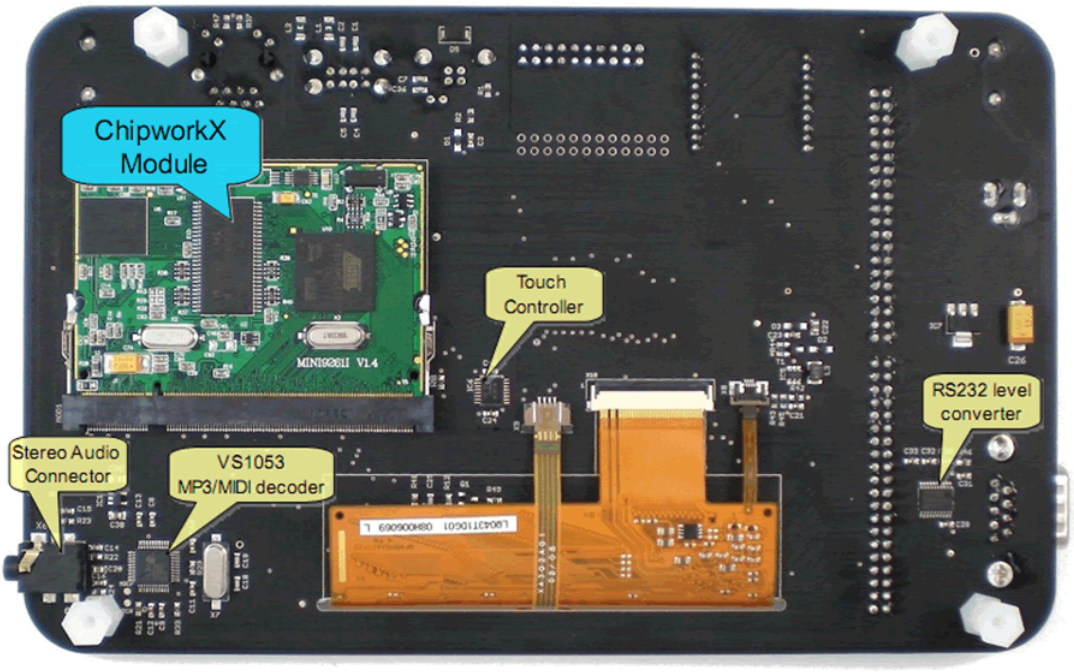



Abbildung 2.3.: ChipworkX Development System - Back View [GHI10a]

Für die Entwicklung von Software für dieses Board stehen sowohl das *.NET Micro Framework* mit den kompletten Libraries, als auch Assemblies von GHI Electronics, die speziell auf das Board zugeschnitten wurden, zur Verfügung. GHI bietet dabei Libraries für die Ansteuerung von USB-Clients oder USB-Hosts bis hin zu eigenen Core-Libraries mit diversen Klassen für Cyclic Redundancy Checks (CRC) oder auch mathematischen Klassen für diverse Berechnungen. Für die spezielle Hardware (Analog-Digital-Wandler, CAN, LCD, TouchScreen, PWM usw.) ist ebenfalls eine Library *Hardware* vorhanden. Die Assemblies sind nicht nur für das ChipworkX Development-System verwendbar, sondern für alle GHI-Produkte, das heißt, man kann jederzeit zu anderer Hardware von GHI wechseln, ohne Code verändern zu müssen - vorausgesetzt die gesamte verwendete Hardware ist auf diesen Boards vorhanden bzw. wird von diesen unterstützt. Die verwendeten Assemblies werden einfach dem Projekt in Visual Studio hinzugefügt und können dann dort verwendet werden. [GHI10a]

Für die .NETMF Libraries und die GHI Assemblies gibt es jeweils *frei zugängliche API-Dokumentationen*, wobei zu bemerken ist, dass sowohl die Dokumentation von *Microsoft*, als auch die Dokumentation von *GHI-Electronics* lückenhaft und zum Teil fehlerhaft ist. Diverse Methoden, die in der MSDN-Library vorhanden sind, wurden mit Umstellung auf die Version 4.1 des .NETMF verändert oder auch entfernt. In der Dokumentation scheint dies allerdings nicht auf.

- API Reference für das .NETMF in der MSDN-Bibliothek: `http://msdn.microsoft.com/en-us/library/ee435793.aspx`

- API Reference für GHI Assemblies findet man bei GHI online unter: `http://www.ghielectronics.com/downloads/NETMF/Library%20Documentation/Index.html`

Abbildung 2.4 zeigt eine Gegenüberstellung diverser GHI Hardware, wobei von rechts nach links die Kosten sinken, allerdings auch die enthaltenen Features geringer werden. Für eine Auswertungseinheit in der geplanten Komplexität ist sicherlich das ChipworkX-Modul die beste Wahl. Auf dem Modul befindet sich ein ARM9-Prozessor mit 200Mhz Systemtakt (AT91SAM9261S), 64MB RAM, 8MB Flash, 256MB interner Flash mit Dateisystem und jede Menge Peripherie. Die komplette Liste kann dem *ChipworkX User Manual* [GHI10b] entnommen werden.

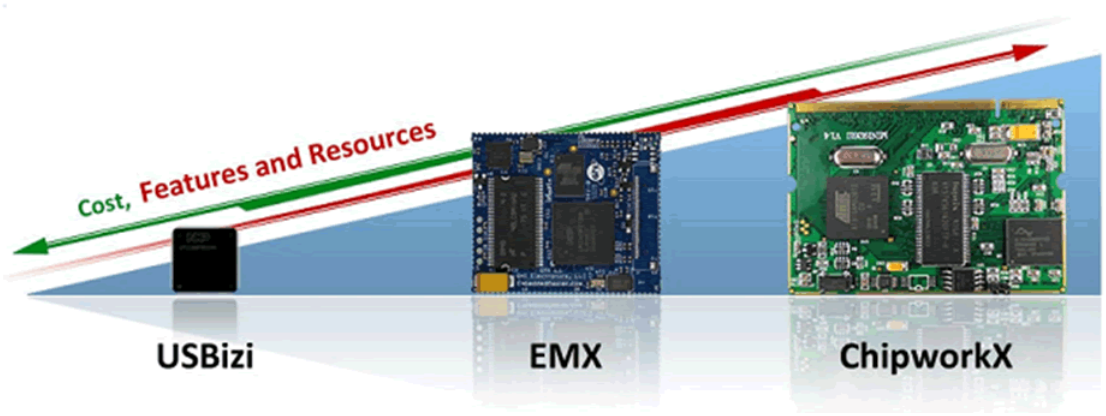



Abbildung 2.4.: Gegenüberstellung diverser GHI Hardware [GHI10b]

Das Zusammenspiel zwischen den Schichten des .NETMF und dem ChipworkX-Modul zeigt die Abbildung 2.5. Die Libraries von GHI Electronics, die speziell für diese Hardware entworfen wurden, und die Libraries des .NETMF arbeiten direkt mit dem CLR zusammen, welches den Code interpretiert. Für diese Interpretation des Codes verwendet das CLR den zur Verfügung gestellten RAM und lagert diverse Teile auch in den Flash aus, um RAM einzusparen. Die 256MB NAND-Flash können speziell zum Ablegen größerer Datenmengen, wie zum Beispiel Grafiken oder andere Ressourcen, verwendet werden. Der PAL und HAL stellen die Schnittstelle zum GHI-Developmentboard und dem 32-Bit ARM9 Prozessor her. Diese arbeiten direkt mit den jeweiligen Komponenten zusammen. Der 8MB große NOR-Flash-Speicher beinhaltet die Firmware des Moduls und darf vom Programmierer nicht verwendet werden. Der serielle Speicher (4MB) wird hauptsächlich für den Bootvorgang verwendet und steht für andere Anwendungen üblicherweise auch nicht zur Verfügung. Der GHI Runtime Loadable Procedure (RLP) Loader erlaubt es dem Benutzer, compilierten Code direkt im gemanagten Code auszuführen, was sich sehr ähnlich zu DLLs am PC verhält. Mit diesem Hilfsmittel können zeitkritische oder prozessintensive Funktionen in Assembler oder nativem C Code implementiert werden, wofür allerdings das Joint Test Action Group (JTAG) Interface benötigt wird. Dieses ist standardmäßig am Developmentboard deaktiviert und steht nur bestimmten Benutzern zur Verfügung. Ebenso steht auf den verwendeten Modulen *SQLite* zur Verfügung, was für die Datenspeicherung des $ECO_2$-Managers durchaus interessant erscheint. Die Daten können damit auf

SD-Karte, USB-Stick oder im internen NAND-Flash verwaltet werden und können mit SQL-Queries gespeichert und abgefragt werden. [GHI10b]

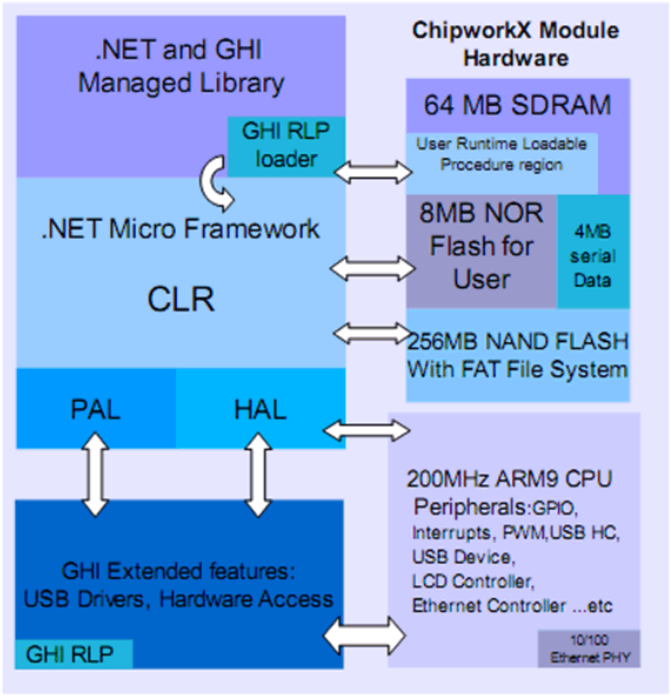



Abbildung 2.5.: Übersicht des Aufbaus des ChipworkX-Moduls im Zusammenspiel mit dem .NETMF [GHI10b]

Ein klarer Nachteil von .NET Micro Framework gegenüber Betriebssystem-basierenden Frameworks ist, dass man sich selbst um alle Dinge kümmern muss. Für Dinge, wie Dimmen der Displaybeleuchtung oder eine Kalibrierung des Touchscreens usw. gibt es zwar oft vorgefertigte Funktionen, um den Aufruf und die genaue Verarbeitung der zurückgelieferten Daten muss man sich allerdings selbst kümmern. Wenn ein Betriebssystem vorhanden ist, kümmert sich dies meistens um diese Funktionen, damit dem Programmierer diese Arbeit abgenommen wird. Für den $ECO_2$-Manager wurden einige solcher Funktionen implementiert, um eine komfortable Bedienung zu ermöglichen.

## 2.3. besondere verwendete Konstrukte

### Weak Reference und Extended Weak Reference

Das .NET Micro Framework CLR stellt sogenannte Extended Weak References (EWRs) zur Verfügung. Der Begriff *Reference* ist ein Verweis, welcher auf ein Objekt zeigt. Eine *Weak Reference* ist ein *schwacher Verweis*, welcher nicht verhindert, dass ein Objekt bei der *Garbage Collection* freigegeben wird. Sie wird meist verwendet, um große Objekte zu kennzeichnen, welche zwar gut sind, wenn sie vorhanden sind, allerdings gelöscht werden können, sollte mehr Speicher benötigt werden. Ein typisches Beispiel dafür wäre ein Cache-Speicher, da dieser jederzeit wieder neu geladen werden kann.

Eine *Extended Weak Reference* erweitert dieses Modell. Dabei werden die Daten automatisch im nicht flüchtigen Speicher (Flash) gespeichert und gelesen, damit diese auch einen Reboot des Systems bzw. einen Reset überstehen. Werden Daten an eine EWR übergeben, werden diese serialisiert und im Flash-Speicher abgelegt, aber zusätzlich im RAM gehalten. Sollte dieser Speicher im RAM benötigt werden, wird dieser vom Garbage Collector freigegeben, aber die Daten können jederzeit aus dem Flash wieder hergestellt werden. Mit der Priorität, die für jede EWR vergeben werden kann, kann dem Garbage Collector mitgeteilt werden, wie wichtig die Daten sind und welche vorher gelöscht werden können. Klassen die der EWR übergeben werden müssen auch serialisierbar sein, damit sie im Flash gespeichert werden können. [Kue08]

### Hashtable

Im .NETMF sind auch *Hashtables* verfügbar, welche über die Library *System.Collections* eingebunden werden können. Ein gravierender Nachteil bei dem *Hashtables* im .NETMF ist allerdings, dass diesen kein Typ hinzugefügt werden kann und die Werte daher gecastet werden müssen. In gewisser Weise ist die *Hashtable* aber trotzdem geschützt, dass kein falscher Cast ausgeführt werden kann, dies führt zur Laufzeit zu einer Exception. Während der Entwicklung im Visual Studio muss allerdings auf den Typ geachtet werden, dieser kann bei der Deklaration nicht angegeben werden.

### Dispatcher

In der Struktur von Programmen in .NETMF gibt es meist eine grafische Oberfläche, welche im *Main-Thread* bzw. *UI-Thread* läuft. Dort werden diverse Objekte wie Fenster, Buttons und dergleichen verwendet, um eine ansprechende Oberfläche zu erstellen. Die Verarbeitung der Daten sollte möglichst in einem separaten Thread passieren, damit die Ressourcen des Systems ausgenützt und quasi parallel die Oberfläche und die Datenverarbeitung ausgeführt werden.

Ein Problem stellt dann oft die Verwendung von Daten im Graphical User Interface (GUI) dar, welche in einem anderen Thread bearbeitet werden. Auch der Thread, der Daten verarbeitet, möchte eventuell auf die grafische Oberfläche zugreifen und dort etwas darstellen. Damit es hier nicht zu Problemen mit der Synchronisation kommt, ist ein direkter Zugriff von einem Thread auf Objekte, die in einem anderen Thread instanziert wurden, nicht erlaubt und führt zu einer *Exception*. Durch den *Dispatcher* kann herausgefunden werden, ob Zugriffsrechte existieren oder nicht und es kann über ihn zugegriffen werden. Um die Synchronisation kümmert sich ebenfalls der Dispatcher selbst.

Viele Dokumentationen verweisen noch auf ältere Methoden, bei denen eigene *Delegates* verwendet werden mussten, seit .NET Micro Framework 4.0 kann mittels der folgenden Aufrufe einer Methode das entsprechende Element verändert werden, wobei der Dispatcher auf Synchronisationsprobleme acht gibt.

- `DispatcherElement.Dispatcher.BeginInvoke(Methode, Parameter)`
  (Thread wird fortgesetzt)

- `DispatcherElement.Dispatcher.Invoke(Timeout, Methode, Parameter)`
  (Thread wartet auf fertige Ausführung der Methode oder bricht nach Timeout ab)

### Events und Delegates

Diese Begriffe spielen hauptsächlich in der ereignisorientierten Programmierung eine Rolle, wie sie bei der Umsetzung von grafischen Oberflächen mit Touchscreen-Input oft zum Einsatz kommt.

Ein *Delegate* ist einem Funktionspointer in C/C++ gleichzusetzen - die Referenz auf die Methode wird gekapselt in einem *Delegate-Objekt*, welches dann im Code verwendet werden kann, wobei zur Zeit des Kompilierens noch nicht bekannt sein muss, welche Methode genau aufgerufen wird. Es kann also ein *Delegate* mehrere verschiedene Funktionen gleichzeitig aufrufen.

*Events* werden auch sehr oft in der asynchronen Programmierung verwendet. Die eigentliche Idee ist es, Events zur Verfügung zu stellen, bei denen man Methoden anmelden kann, die beim Aufruf des Events ausgeführt werden. *Events* werden meist in Kombination mit *Delegates* verwendet, wie dies funktioniert soll hier kurz erläutert werden, da dies im Projekt des öfteren verwendet wird. Viele Events werden auch durch Klassen des Frameworks bereitgestellt. [Zah02]

- Anlegen eines *Delegates*
- Anlegen eines *Events* basierend auf dem *Delegate*
- Definieren und registrieren der entsprechenden *Event-Handler* (auch mehrere für ein Event möglich)
- Durch Auslösen des Events werden alle zugewiesenen Handler ausgeführt.

# Kapitel 3.

# User Interface

## 3.1. Touchscreen

Auf dem Development Board von GHI Electronics ist sowohl der Touchscreen, als auch der entsprechende Controller dafür verbaut. Da das ChipworkX-Modul standardmäßig den Touchcontroller **TSC2046** von Texas Instruments (TI) unterstützt wird dieser dort verwendet. Sollte ein anderer Controller verwendet werden, muss dafür ein Treiber für das .NETMF geschrieben werden, welcher die Koordinaten an die oberen Schichten des .NETMF weiter gibt. Empfohlen wird für das ChipworkX-Modul jedoch der angegebene Controller, da dieser bereits ein Preprocessing für den Serial Peripheral Interface (SPI) Bus besitzt, welches hilft, Traffic am Bus einzusparen. Speziell beim verwendeten Modul ist dies wichtig, da der SPI-Bus mit mehreren Komponenten geteilt wird (Touchscreen, SD-Karte, USB...).

Als Display wurde auf dem Developmentboard ein 4,3 Zoll großes TFT-Display von Sharp (**LQ043T1DG01**) mit resistivem Touchscreen verbaut. Es handelt sich dabei um einen Single-Touchscreen, welcher mit dem Finger oder mit einem Stift bedient werden kann. Empfohlen wird die Verwendung eines Stiftes, da damit die Koordinatenzuordnung besser funktioniert und auch kleinere Buttons getroffen werden können. Am Besten eignet sich ein Stift mit Plastikspitze, ein Kugelschreiber mit eingezogener Miene funktioniert ebenfalls. Zur Schonung des Bildschirms sollte jedoch kein Kugelschreiber mit ausgefahrener Miene verwendet werden, diese würde die Oberfläche des Touchscreens verletzen.

Die Auflösung des Displays beträgt 480x272 Pixel und kann über das .NET Micro Framework und den Treiber für den oben genannten Controller angesprochen werden. Die Touchscreenwerte werden mit 10 oder 12-Bit aufgelöst, sodass auch größere Displays ohne weiteres verwendet werden können. Von Sharp gibt es resistive Touchscreens von 2,2 Zoll bis zu 15 Zoll Größe, es ist also ein großer Spielraum gegeben. Auch andere resistive Touchscreens können mit diesem Controller verwendet werden, da die Displays mit vier Leitungen Spannungswerte von zwei elektrisch leitfähigen Schichten liefern, die über das Display gelegt werden. Diese Spannungen sind proportional zur gedrückten Position und ermöglichen so dem Controller (egal von welchem Hersteller oder wie groß das Display ist) durch einen Analog Digital Wandler (ADW) die Position zu bestimmen und digital weiterzugeben. Es sind auch Displays mit mehreren Anschlüssen vorhanden (5, 6, 7 oder sogar 8 Drähte), wobei diese Varianten zwar genauer sind, allerdings auch um einiges teurer und seltener, da sich der Aufwand nicht lohnt. Auch andere Technologien, wie kapazitive oder induktive Touchscreens stellen eine Alternative dar, sind aber meist teurer und können nicht mit simplen Plastikstiften bedient werden, was bei kleineren Displays mit kleinen

Symbolen sicher einen Nachteil darstellt. Die Möglichkeit einen Multitouch-Screen zu verwenden besteht im .NET Micro Framework ab Version 4.1 ebenfalls, allerdings ist dazu eine andere Hardware nötig, die wesentlich kostenintensiver ist. Für eine Neuentwicklung der Hardware ist auf jeden Fall ein Multitouch-Screen zu empfehlen, die Preise für diese werden in nächster Zeit wegen der weiten Verbreitung bei Mobiltelefonen weiter sinken.

### 3.1.1. Displayhelligkeit

Die Hintergrundbeleuchtung des LCD-Displays des *ChipworkX Development Systems* kann über einen PWM-Ausgang gedimmt werden. Dazu muss am System ein Kontakt zwischen **Pin SV2-31(PB4/PWM)** und **Pin SV2-33 (ADJ)** hergestellt werden, was mit einem einfachen Jumper realisiert werden kann. Werden diese Pins nicht verbunden, ist keine Regulierung der Helligkeit möglich, das Display ist dauernd eingeschaltet und auf 100% Helligkeit gesetzt. [GHI09]

Um das Display zu dimmen, wurde die Klasse **DisplayControl** geschrieben. Diese Klasse enthält nur statische Methoden zum Steuern der Displayhelligkeit und zum Deaktivieren des Displays (keine Hintergrundbeleuchtung aktiv). Gedimmt werden kann mit den Methoden *illuminationUp* und *illuminationDown*, wobei jeweils die Pulsweitenmodulation auf einen neuen Wert gesetzt wird. Die Frequenz bleibt immer gleich, aber die Pulsweite wird im Bereich von 0-80% geändert, wobei 100% Pulsweite die Hintergrundbeleuchtung deaktiviert ist, bei 0% erhält man die volle Helligkeit. Unter 20% Helligkeit dimmen ist allerdings nicht möglich, da darunter das Display ausschaltet, der Grund dafür ist der Transistor, der die Helligkeit regelt, der unter 20% Pulsweite sperrt. Die Schrittweite zum Dimmen kann in der Klasse **Constants** mit der Variable *DISPLAY_ILLUMINATION_STEP* eingestellt werden.

Wird der Helligkeitswert verändert, wird dieser in die EWR geschrieben, damit dieser Wert nach einem Neustart des Systems noch vorhanden ist. Beim Start des Systems wird der statische Konstruktor der Klasse aufgerufen, wodurch die entsprechende EWR ausgelesen wird und der *illuminationValue_* wird auf den gespeicherten Wert gesetzt. Sollte keiner vorhanden sein, wird die maximale Helligkeit verwendet.

Soll die Hintergrundbeleuchtung deaktiviert werden, kann die Methode *disableDisplayIllumination* verwendet werden, welche zuerst ein schwarzes Fenster für die Reaktivierung über den gesamten Bildschirm legt und anschließend die Pulsweite des PWM-Ausgangs auf 100% setzt. Soll das Display wieder aktiviert werden, reicht ein Klick auf irgendeine Stelle des Touchscreens, dem über den restlichen Inhalt gelegten Fenster wurde ein *TouchUp-Event* zugewiesen, welches die Methode *enableDisplayIllumination* aufruft, welche das Fenster schließt und die Displaybeleuchtung auf den eingestellten Helligkeitswert setzt. Ein Problem besteht, wenn auf den ausgeschaltenen Touchscreen doppelt geklickt wird. Dies passiert sehr oft unabsichtlich und sehr schnell hintereinander (prellen des Stiftes). Das Problem dabei liegt im doppelten Aufruf der Methode *enableDisplayIllumination*, welcher so schnell hintereinander passiert, dass nicht einmal eine Abfrage einer speziellen Variable funktioniert.

### 3.1.2. Kalibrierung des Touchscreens

Resistive Touchscreens benötigen meist eine Kalibrierung - Fertigungstoleranzen, eventuelle Alterserscheinungen, elektrisches Rauschen oder leichte Fehlpositionierung der Oberflächen sind ausschlaggebend dafür. Stimmen die Spannungswerte, die gemessen werden, nicht mit den Kalibrierungswerten überein, ergibt sich dadurch ein verschobenes Koordinatensystem der berührungsempfindlichen Fläche gegenüber der Anzeige am Display und Schaltflächen usw. können dann nicht richtig bzw. gar nicht angesprochen werden. Daher ist eine Kalibrierung bei einer Neuauslieferung und nach einiger Zeit der Benutzung unbedingt erforderlich.

Wie die Kalibrierung abläuft wird genauer in Unterabschnitt 3.4.7 beschrieben

## 3.2. Windows Presentation Foundation (WPF)

Für grafische Darstellungen mit dem .NETMF steht das Framework Windows Presentation Foundation (WPF) zur Verfügung. WPF ist aus dem *.NET Framework 3.0* bekannt und stellt dort eine Möglichkeit dar, den Code und das Design eines Programmes zu trennen.

Im .NET Micro Framework wird WPF allerdings in einer etwas anderen Art und Weise verwendet, eine Trennung von Design und Code wird hier leider nicht erreicht, dies wäre für den Interpreter zu aufwändig. Es existiert ein Projekt von *Jan Kucera*, bei dem versucht wird, ebenfalls Code und Design zu trennen. Dazu wird, wie in WPF des .NET Frameworks auch, Extensible Application Markup Language (XAML) verwendet. Das Projekt funktioniert zwar, bietet aber zu wenig Freiheiten beim Positionieren der Elemente und dem Verwenden von Buttons und aktiven Schaltflächen. Da die Bedienung in diesem Projekt aber rein über den Touchscreen erfolgen soll, ist diese Technik leider nicht ideal.

Des Weiteren bietet *GHIElectronics* eine Library namens **Glide** an. Dort kann mit der Hilfe eines Designers der Bildschirm gestaltet werden, dies wird dann in Extensible Markup Language (XML) abgebildet und mittels der Glide-Funktionen im Projekt eingebunden. Zur Verfügung stehen dort viele Elemente wie Radio Buttons, Checkboxen, Dropdown-Menüs, sogar eine Tastatur usw, eine Touchscreen-Bedienung ist dort auch vorgesehen. Der große Nachteil dabei - die Library für den kommerziellen Bereich kostet in etwa 1000 Dollar - dies war der Grund, warum das Projekt ohne diese Library entwickelt wurde. Zu finden ist die Library auf der Webseite von *GHI Electronics*: `http://www.ghielectronics.com/glide/`

Für dieses Projekt werden die einzelnen Klassen des .NETMF verwendet um ein GUI zu erzeugen. Es besteht die Möglichkeit mit verschiedenen Formen und Bitmaps direkt zu arbeiten und die Touchscreen-Aktionen händisch auszuwerten oder man kann das abgewandelte WPF-Framework verwenden. Das Framework ermöglicht es, User-Interface-Elemente (UI-Elemente) ausgehend von einer Basisklasse zu generieren und zu positionieren. Einige dieser Elemente werden in Folge beschrieben.

### 3.2.1. UIElement

Das **UIElement** stellt eine Basis-Klasse für Elemente der grafischen Oberfläche dar. Zu finden ist die abstrakte Klasse im Namespace *Microsoft.SPOT.Presentation* und stellt eine Reihe von Methoden, Properties und Events zur Verfügung, welche für die Positionierung, das Aussehen bzw. für die entsprechenden Aktionen bei Touchscreen-Events verwendet werden. Einige wichtige Methoden werden hier kurz beschrieben, da diese im Projekt des Öfteren verwendet werden:

- $<$ *Property* $>$ **Height,** $<$ *Property* $>$ **Width:** Zum Setzen der Höhe und Breite des Elements. Wird hier nichts angegeben wird immer die minimalste Größe angenommen, wodurch das Element eventuell nicht sichtbar wird.
- $<$ *Property* $>$ **Visibility:** Dieses Property muss auf *Visibility.Visible* gesetzt werden, da ansonsten das Element nicht angezeigt wird.
- **Invalidate():** Wird diese Methode aufgerufen, wird das Objekt auf "ungültig" gesetzt und wird schnellstmöglichst neu gezeichnet. Viele vorgefertigte Elemente rufen bei entsprechenden Änderungen diese Methode selbst auf.
- **OnRender():** Diese Methode muss vor allem bei abgeleiteten Elementen beachtet werden. Sie wird aufgerufen, wenn *Invalidate()* eines Objektes aufgerufen wurde. In diese Methode muss das neu Zeichnen für das aktuelle Objekt implementiert werden.
- **OnTouchDown(), On TouchUp(), OnTouchMove(), OnTouchGestureChanged():** Diese Methoden werden automatisch aufgerufen, wenn am entsprechenden Objekt eine Touchscreen-Aktion erkannt wurde. Die entsprechende Art der Aktion kann dem Methodennamen entnommen werden und die Methode kann je nach gewünschter Reaktion implementiert werden. Diese Dinge stehen auch als entsprechende Events zur Verfügung um von aussen entsprechende Aktionen durchführen zu können.

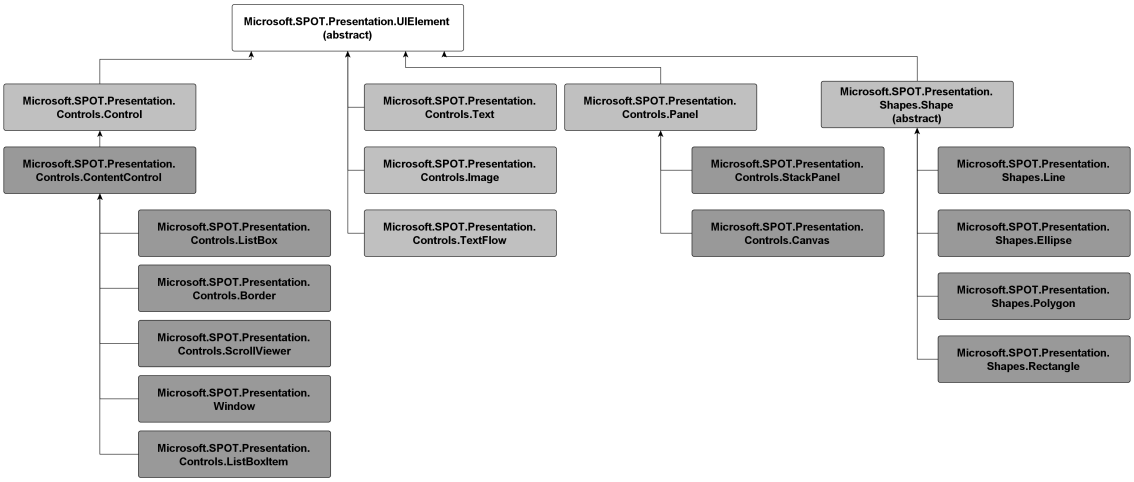



Abbildung 3.1.: Ableitungen von UIElement [Kue08]

Ausgehend von diesem Element werden im Projekt einige Elemente abgeleitet, welche zum Teil die Methoden überschreiben. Viele Systemklassen des WPF leiten ebenfalls von dieser Klasse ab. Einen guten Überblick zu den Ableitungen vom **UIElement** bietet die Abbildung 3.1, die Basiselemente werden in den Abschnitten unterhalb genauer beschrieben.

#### Static Controls

Statische Elemente, wie zum Beispiel Grafiken, Texte, Rechtecke, Linien usw. können keine Kindelemente annehmen, das heißt, sie bilden die *unterste Ebene* von Elementen bzw. es können keine weiteren Elemente zu den statischen mehr hinzugefügt werden. In ein **Image** zum Beispiel kann kein Strich gezeichnet werden. Dieser kann nur darüber gelegt werden. Vom Verhalten und der Positionierung stehen aber alle Varianten aus der Klasse **UIElement** zur Verfügung.

Eine Besonderheit stellt bei den statischen Elementen noch die Klasse **Shapes** dar. Diese Klasse ist abstrakt und kann daher nicht direkt instanziert werden. Von ihr werden aber einfache Formen wie Rechtecke, Linien, Polygone, Ellipsen usw. abgeleitet.

#### Content Controls

Diese Elemente können *genau ein weiteres Element* als Kindelement beinhalten, welches auf der Fläche positioniert werden kann. Zum Beispiel ein **Window** kann nur ein weiteres Element beinhalten, allerdings beinhalten die abgeleiteten Klassen einige weitere Funktionen, die für das Handling nützlich sein können.

#### Panels

Um auch mehrere Elemente in einem anderen Element positionieren zu können, gibt es zwei verschiedene Arten von Panels. Das **StackPanel** kann mehrere Kindelemente aufnehmen und positionieren. Dabei kann ausgewählt werden, ob man diese untereinander oder nebeneinander positionieren will, mehr Möglichkeiten hat man in diesem Fall nicht. Will man noch mehr Freiheit beim Positionieren, kann man ein **Canvas** verwenden, welches es erlaubt, die Elemente vollkommen frei zu positionieren. Diese beiden Arten werden in den verschiedenen Varianten bei diesem Projekt eingesetzt um Elemente an die entsprechende Position zu verschieben.

### 3.2.2. Fonts

Um Schriftarten in .NETMF verwenden zu können, müssen diese in den Typ *Tinyfont* konvertiert werden. Dazu ist im Internet ein Projekt namens *TinyFontToolGUI.exe* von *Jan Kucera* verfügbar, welches Windows-Schriftarten in Tinyfonts umwandelt. Im Framework selbst ist nur eine textbasierende Version dieses Tools enthalten. Abbildung 3.2 zeigt einen Screenshot des Tools. Es kann sowohl die Qualität der Schriftart angegeben werden, als auch die Zeichen, die inkludiert werden sollen. Wichtig zu erwähnen ist, dass man auch die Sonderzeichen (links unten die beiden Haken *Symbols* und *Special*) mit integrieren sollte, da diese des Öfteren verwendet werden und nur dann angezeigt werden können.

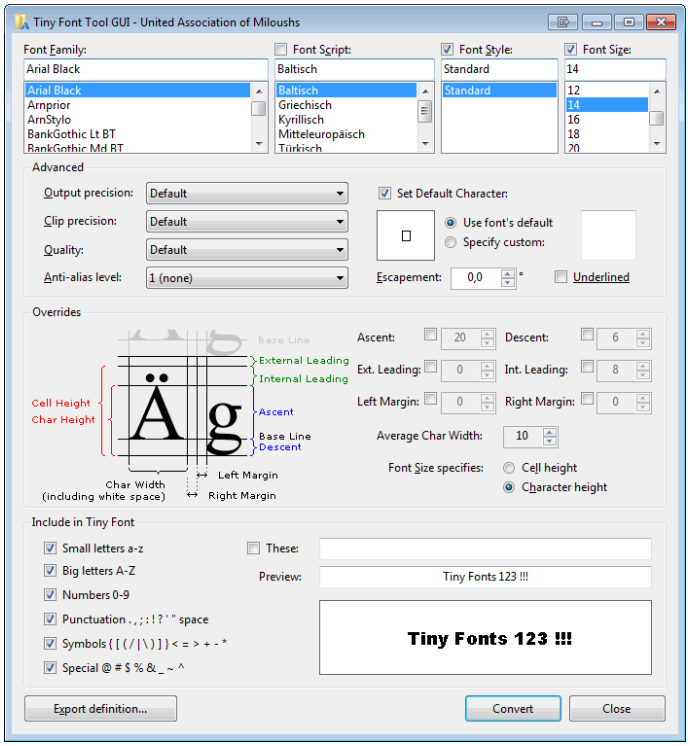



Abbildung 3.2.: Tiny Font Tool GUI Screenshot

### 3.2.3. Ressourcen

Damit Grafiken, die erzeugten Schriftarten usw. im Projekt verwendet werden können, müssen diese als Ressourcen dem .NETMF zur Verfügung gestellt werden. *Microsoft Visual C# 2010 Express* bietet dazu einen Ordner **Resources**, in dem diese Dateien abgelegt werden müssen, sie können dort hineingezogen werden. Allerdings muss man dem Ressourcenmanager (zu finden durch Doppelklick auf **Resources.resx**) mitteilen, welche Ressourcen er verwalten soll. Ruft man den Manager auf, können dort Zeichenfolgen, Bilder, Symbole, Audiodateien usw. hinzugefügt werden. In diesem Projekt werden hauptsächlich Bilder für die grafische Darstellung dort hinterlegt, in der Anwendung kann über die Klasse **Resources** auf die abgelegten Elemente zugegriffen werden.

## 3.3. grafische Grundstruktur und Start des Programms

Das System startet mit der Main-Methode, die in der Klasse **Program** definiert ist. Dort wird zuerst ein Startbild (siehe Abbildung 3.3) festgelegt, welches während dem Bootvorgang angezeigt wird. Dies wird jedesmal gesetzt um sicher zu stellen, dass es beim nächsten Start aktiv ist. Das Bild wird statt den Boottextinformationen angezeigt und wird in der maximal möglichen Größe dargestellt. Ebenso wird hier die Initialisierung für den Sound-Chip durchgeführt, indem **SoundControl.initialize()** aufgerufen wird.


Abbildung 3.3.: Bildschirm beim Starten des $ECO_2$-Managers

Im Anschluss an den Bootvorgang wird ein neues Objekt von der Klasse **Program** instanziert, der Touchscreen darauf initialisiert, wobei angegeben werden kann, ob der Treiber die Daten des Touchscreens auswerten soll oder der gemanagte Code. Hier wurde der Treiber gewählt, um eine bessere Performance zu erhalten, ein Debugging der Touchscreenaktionen ist nicht notwendig. Der *CollectionMode* wurde mit *InkAndGesture* festgelegt, was es erlaubt, sowohl Klick-Aktionen auszuwerten, als auch Gestures wie Schwenks nach rechts oder links auszuführen. Die Zeit, nach der ein Move-Events des Touchscreens erkannt wird, wurde auf mit der Konstante *Constants.UIBUTTON_PRESS_AND_HOLD_INCREASE_INTERVAL* festgelegt (derzeit 400ms), um Moves bei einem Klick zu verhindern und klar von diesen unterscheiden zu können. Mit einem Multitouch-Screen wären auch Gestures wie zum Beispiel Zoom oder Dreh möglich, diese müssen aber hier nicht beachtet werden, da keine Multitouch-Erkennung bei resistiven Touchscreens verfügbar ist. Die Sampling-Frequenz für Touchscreen-Aktionen wird auf 100ms festgelegt, es wird also zehn Mal pro Sekunde abgefragt, welche Touchscreenaktionen gesetzt wurden.

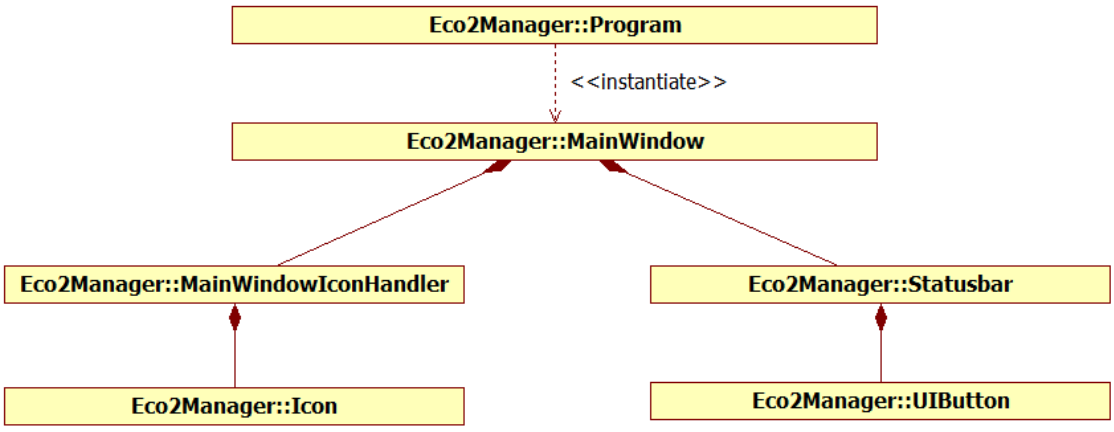



Abbildung 3.4.: Übersicht über die Klassen der grafische Grundstruktur

Zusätzlich stellt die Klasse **Program** eine Methode *setMainWindow(Window)* zur Verfügung,

mit der das Hauptfenster geändert werden kann. Grundsätzlich wird immer das **MainWindow** geladen, da aber für die Kalibrierung des Touchscreens (siehe Unterabschnitt 3.4.7) das Hauptfenster getauscht werden muss (auf **CalibrationWindow**), um alle anderen Fenster dahinter zu schließen, kann dieses hier gesetzt werden. Dies sollte allerdings mit allerhöchster Vorsicht verwendet werden, nach Möglichkeit nur für das Kalibrieren. Ansonsten sollte hier immer das **MainWindow** als Hauptfenster angegeben sein. Eine weitere Ausnahme für die Verwendung stellt auch das Ausschalten des Systems dar. Da alle Objekte im MainWindow gehalten werden, wird dieses durch ein leeres, schwarzes Fenster ersetzt, wodurch dem Garbage Collector ermöglicht wird, die Objekte zu entfernen.

Das Zusammenspiel zwischen den Klassen **Program**, **MainWindow**, **MainWindowIconHandler**, **Statusbar** und zwei weiterer Hilfsklassen ist in Abbildung 3.4 dargestellt.

### 3.3.1. Main Window

Das Fenster, das nach dem Start des Systems sofort angezeigt wird, ist eine Instanz der Klasse **MainWindow** und wird von der Klasse **Window** abgeleitet. Durch die Ableitung stehen diverse Methoden und Properties zur Verfügung, die wichtigsten, die hier verwendet werden, sind *Height*, *Width*, *Background* und *Visibility*. Damit wird das Fenster auf die Größe des Bildschirms gebracht, der Hintergrund gesetzt und das Fenster sichtbar gemacht.

Um Elemente positionieren zu können wird auf dem Fenster ein **StackPanel** platziert, welches eine Anordnung von Elementen untereinander ermöglicht (StackPanel mit vertikaler Ausrichtung). Damit wird die Trennung in den Fenster- bzw. Iconbereich und in die Statusleiste, wie in Abbildung 3.5 dargestellt erreicht. In diesem Fall ist ein StackPanel wesentlich einfacher zu verwenden als ein Canvas, da nur zwei Elemente untereinander positioniert werden müssen. Es genügt ein hinzufügen der Bereiche mittels *panel.Children.Add* und die Objekte werden richtig angezeigt. Die Größe legen die Objekte selbst fest.

Die Funktionen der Statusleiste und die Positionierung der Icons auf dem Bereich der dafür vorgesehen ist wird von zwei eigenen Klassen übernommen, diese werden in den folgenden Abschnitten beschrieben.

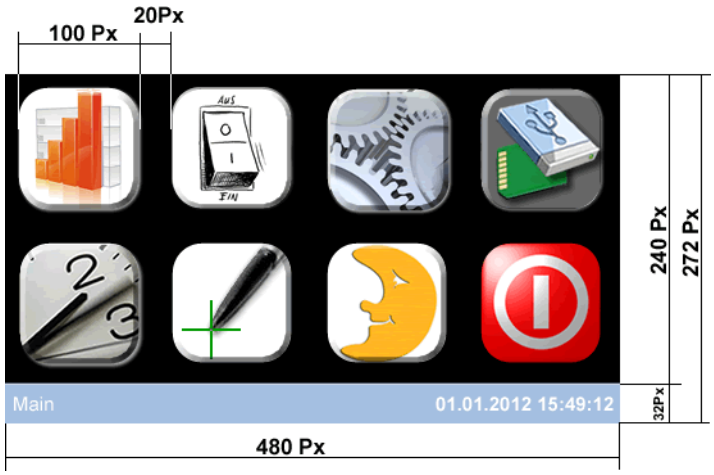



Abbildung 3.5.: MainWindow mit Bemaßung der Abstände

### Positionierung der Icons

Da die Symbole im Hauptbereich nicht fix angeordnet werden sollen und eine einfache Erweiterung möglich sein soll, wurde die Klasse **MainWindowIconHandler** (siehe Abbildung 3.6) geschaffen. Der Klasse können Icons hinzugefügt werden, die maximale Anzahl hängt von der Größe des verwendeten Touchscreens, der Größe der Icons und dem eingestellten Abstand zwischen den Icons ab. In den globalen Konstanten (Klasse **Constants**) kann die Einstellung für den Iconabstand (*ICON_MARGIN*) getroffen werden, die Icongröße wird automatisch aus der Grafik gelesen, wobei wichtig ist, dass die Icons *quadratisch* sein müssen, ansonsten funktioniert die Positionierung nicht richtig!

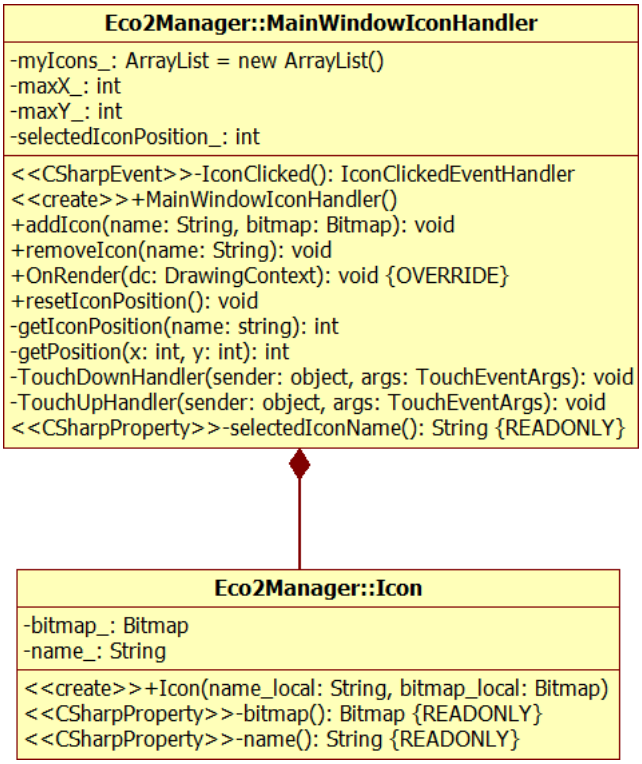



Abbildung 3.6.: Klassendiagramm MainWindowIconHandler und Icon

Erstellt man einen IconHandler, wird für den gesamten Hauptbereich (Bildschirmgröße ohne Statusleiste) ein *TouchDownHandler* eingerichtet, welcher Toucscreenaktionen an bestimmte Methoden weiterleitet. Nachdem dieser erstellt wurde, können Icons hinzugefügt werden, indem der Methode *addIcon* der Name und das Bitmap des gewünschten Icons übergeben werden. Um die Positionierung und die Touch-Aktionen des Icons kümmert sich die Klasse selbst. Auf die gleiche Art und Weise kann auch mit *removeIcon* ein Icon wieder entfernt werden. Dies kann jederzeit während der Laufzeit auf das entsprechende Objekt angewandt werden, die Icons können also auch bei Bedarf dynamisch während dem Betrieb hinzugefügt und entfernt werden. Es wurde absichtlich nicht jedem Icon ein Touchscreen-Event zugewiesen um eine Korrektur des gewählten Elementes durch gedrückt halten und aus dem Symbol herausziehen zu ermöglichen. Das Problem bei Touch-Down und -Up Handlern besteht darin, dass nicht unterschieden werden kann, ob der Cursor

aus dem entsprechenden Objekt entfernt wurde. Tippt man auf ein Symbol, hält dieses, bewegt den Finger/Stift aus dem Objekt und lässt dann los, wird immer noch ein Touch-Up-Event für den ursprünglich gewählten Button ausgelöst, die Koordinaten befinden sich ebenfalls noch innerhalb des Symbols. Wird auf ein Icon geklickt, der Finger/Stift aber noch während des Drückens aus dem Symbol gezogen, soll dieses nicht als Klick darauf gewertet werden.

In der Klasse **MainWindowIconHandler** wird eine ArrayList von Objekten der Klasse **Icon** angelegt, die Objekte enthalten Name und Bitmap - im Prinzip alle Infos die benötigt werden um die Icons positionieren zu können. In der Methode *OnRender()* in der Klasse **MainWindowIconHandler** werden die Icons auf den Bildschirm gezeichnet. Alle Einträge der ArrayList werden durchlaufen, zuerst werden die Objekte nebeneinander positioniert, dann folgt die nächste Zeile usw.

Um die Icons nun mit den entsprechenden Aktionen zu versehen, wird der *TouchDownHandler* und der *TouchUpHandler* verwendet. Beim Drücken auf den Touchscreen wird eine Position bestimmt, dabei werden die Koordinaten an die Methode *getPosition* übergeben und diese berechnet den Index des entsprechenden Icons. Dabei beginnt der Index mit 0 links oben und endet beim letzten Symbol rechts unten und wird immer berechnet, egal wohin man im Hauptbereich des Bildschirms drückt, auch wenn sich dort kein Icon befindet. Befindet sich ein Icon an dieser Stelle, wird der Index gespeichert und mit dem Index beim Entfernen des Stiftes vom Touchscreen verglichen. Stimmen diese beiden überein, gilt dieses Icon als gedrückt, es wird die Methode *IconClicked* aufgerufen und der Klick-Sound abgespielt. Mit dieser Methode wird ein EventHandler aufgerufen, welcher extern von der aufrufenden Klasse definiert sein muss. Als Info wird der Name des Buttons mitgeschickt, um extern zu erkennen, welcher Button getroffen wurde.

Hinzugefügt werden die Icons in der Klasse **MainWindow**, wobei auch ein Handler für die gedrückten Buttons hinzugefügt wird (*mainWindowIconClicked*). In diesem werden per Switch-Anweisung die Icons unterschieden und dementsprechend werden weitere Methoden aufgerufen bzw. das aktive Fenster richtig gesetzt.

### Bedeutung der Icons

Die Symbole, die am Startbildschirm positioniert wurden, sind absichtlich groß gehalten, um eine einfache Bedienung zu gewährleisten. Zudem wurde auf zu viele Spielereien verzichtet, um die Bedienung einfach zu halten. Die Symbole haben folgende Bedeutung (siehe auch Abbildung 3.7):

- **Auswertungen:** Ruft das **ReportingWindow** auf, um die verbleibende Monatssumme oder Diagramme für Tages-, Monats- oder Jahresauswertung anzuzeigen.

- **Aktoren:** Hier können mittels **ActorWindow** die vorhandenen Aktoren angesteuert werden. Gedacht ist dies für an Sensorknoten angeschlossene Geräte.

- **Einstellungen:** Displayhelligkeit, Lautstärke für die Töne bei Touchscreen-Aktionen, das Speicherintervall und das Monatslimit für den $CO_2$-Ausstoß können hier festgelegt werden.

- **Speichermedium:** Es kann zwischen SD-Karte, USB-Stick und interner Speicherung gewählt werden. Zudem kann die Datenbank geleert werden. Die Speichermedien können hier auch ausgeworfen werden, dies sollte vor dem Entfernen auf jeden Fall durchgeführt werden.
- **Uhrzeit:** Die Uhrzeit kann am Touchscreen eingestellt werden. Diese wird zwar von einer Real Time Clock gepuffert, aber hier kann die Umstellung von Sommer- und Winterzeit geschehen oder bei einer leeren Batterie die Zeit neu eingestellt werden.
- **Touchscreen kalibrieren:** Sollte nur ausgeführt werden, wenn wirklich notwendig. Einmal gestartet, muss die Kalibrierung beendet werden. Es werden die RAW-Daten des Touchscreens gelesen und der Touchscreen-Controller neu kalibriert.
- **Bildschirm deaktivieren:** Die Hintergrundbeleuchtung des Bildschirms wird deaktiviert.
- **System herunterfahren:** Es werden alle Datenverbindungen beendet.


Abbildung 3.7.: Bedeutung der Symbole am Startbildschirm

### Statusleiste

Die Statusleiste wird mit Hilfe der Klasse **Statusbar** (siehe Abbildung 3.8) gebildet, welche von der Klasse **Canvas** abgeleitet wurde, um eine beliebige Positionierungsmöglichkeit für Elemente in der Statusleiste zu haben. Die Taskleiste soll folgende Aufgaben erledigen:

- Überblick welches Fenster geöffnet ist
- Uhrzeit und Datum darstellen
- Zurückspringen in das Hauptmenü

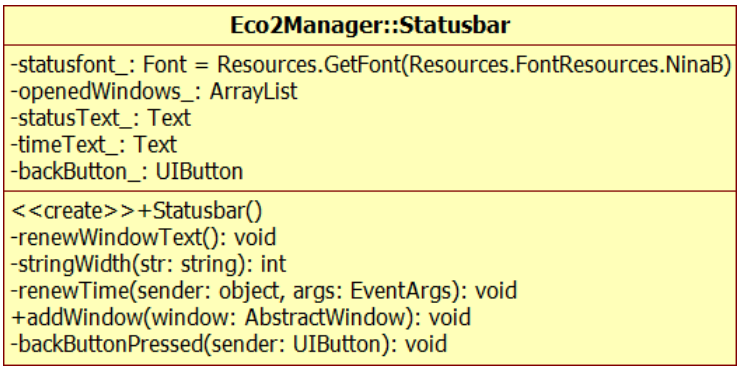



Abbildung 3.8.: Klassendiagramm Statusbar

Innerhalb der Klasse **Statusbar** wird zuerst der Hintergrund für die Taskleiste mit einer ein Pixel breiten Grafik (kann beliebig geändert werden) generiert und als Kindelement zur Klasse hinzugefügt. Im Anschluss wird der Statustext aufgebracht, wobei dieser am Anfang leer ist und nur positioniert wird. Das Objekt muss für spätere Änderungen allerdings zur Verfügung stehen und wird daher als Membervariable der Klasse ausgeführt. Selbiges gilt auch für die Zeitanzeige. Der Button zum Zurückspringen in das Hauptmenü wird immer am Ende des Statustextes positioniert (mit Hilfe der Klasse **UIButton** - siehe Unterabschnitt 3.5.1) und steht ebenfalls wieder als Membervariable zur Verfügung, damit er aktiviert und deaktiviert bzw. richtig positioniert werden kann.

Um die Zeit richtig darzustellen wird ein Timer gestartet, welcher in festen Zeitabständen (in Constants.REFRESH_INTERVAL_TIME_STATUSBAR festgelegt) die Methode *renewTime* aufruft. Damit wird der Zeitstring neu zusammengebaut und am Display angezeigt. Die Zeit wird zu Beginn immer von der Echtzeituhr (Real Time Clock (RTC)) geladen, um nach einem Neustart des Systems die aktuelle Zeit zu erhalten. Sollte die Zeit nicht richtig gespeichert werden, ist meist die Batterie der RTC am Board leer und muss getauscht werden.

Die Methode *renewWindowText* aktualisiert den Statustext, welcher angibt, welches Fenster gerade geöffnet ist. Dazu werden alle geöffneten Fenster an die Statusleiste gemeldet (mit *addWindow*) und der String aus den Namen aller Fenster zusammengebaut. Zusätzlich wird überprüft, ob geöffnete Fenster vorhanden sind, wenn ja, wird der Zurück-Button neben dem Statustext eingeblendet. Dazu steht die Hilfsmethode *stringWidth(String)* zur Verfügung, welche die Breite des Strings in Pixeln liefert, der Zurück-Button wird neben dem Text der geöffneten Fenster angezeigt. Wurde der Zurück-Button gedrückt, wird immer die *Close-Methode* des zuletzt geöffneten Fensters aufgerufen. Da ein Schließen des Fensters hier ermöglicht wird, muss man sich später nicht weiter um einen Button dafür kümmern und im Fenster selbst steht der ganze Platz für den Inhalt zur Verfügung.

Einzige Ausnahme für das oben beschriebene Vorgehen stellt das Kalibrierungsfenster dar, es wird nicht zur Statusleiste hinzugefügt und diese wird auch nicht angezeigt, da die Statusleiste während der Kalibrierung nicht sichtbar ist. Dort muss die Kalibrierung beendet werden, um zum Startbildschirm zurück zu kommen (siehe Unterabschnitt 3.4.7).

Die Statusleiste wird im **MainWindow** instanziert und dem dort vorhandenen Panel als zweites Element hinzugefügt.

#### sonstige Aufgaben des MainWindow

Das *MainWindow* stellt nicht nur die Funktionen für die Grafik des Startbildschirms zur Verfügung, sondern hält auch diverse andere Klassen, welche während der Ausführung immer wieder benötigt werden. Es werden der **DataStorageController**, der **UsbSdController** und die Klasse **Eco2IO** für die Kommunikation gehalten und im Konstruktor instanziert. Über Properties werden diese jeweils anderen Klassen zur Verfügung gestellt, da ein Zugriff von mehreren Punkten im Programm möglich sein muss. Eine statische Instanzierung dieser Klassen ist nicht möglich, da Objekte angesprochen werden, was aus einer statischen Klasse heraus nicht möglich wäre.

## 3.4. Beschreibung der Fenster und deren Funktionen

In diesem Abschnitt werden die verwendeten Fenster, welche über das **MainWindow** aufgerufen werden können kurz beschrieben. Da das **MainWindow** selbst als Grundgerüst dient und bereits im vorherigen Abschnitt beschrieben wurde, wird es hier nicht mehr erwähnt. Die weiteren Fenster werden im **MainWindow** als **activeWindow**_ geöffnet, damit bekannt ist, welches Fenster gerade aktiv ist und auf dieses zugegriffen werden kann.

### 3.4.1. AbstractWindow

Diese Klasse darf nicht instanziert werden, aber bietet eine Grundstruktur für alle Fenster, die vom **MainWindow** aus geöffnet werden. Der Konstruktor benötigt eine Referenz zum **MainWindow**, damit dies für die abgeleiteten Klassen zur Verfügung steht und einen *WindowName*. Dieser wird in der Statusleiste angezeigt. Die Klasse hat auch nur einen Konstruktor, der die beiden Werte nimmt, da eine Instanzierung ohne Fenstername und Referenz zum MainWindow keinen Sinn ergibt.

Das **AbstractWindow** legt auch die Grundeinstellungen für alle Fenster fest, es wird die Höhe und Breite festgelegt, der Hintergrund auf schwarz gestellt und das Fenster sichtbar gemacht. Für den Eintrag des Fensters in der Statusbar wird die Methode *addWindowNameToStatusbar* des **MainWindows** aufgerufen. Damit ist sichergestellt, dass die **Statusbar** über alle aktuell geöffneten Fenster informiert wird, und so eine Navigation ermöglicht. Da ein Verweis auf dieses Fenster vorhanden ist, kann die **Statusbar** dieses schließen. Alle Fenster, welche beim Schließen noch Aktionen ausführen wollen, müssen die Funktion *Close* überschreiben.

### 3.4.2. ReportingWindow

Das **ReportingWindow** dient der Auswertung der Daten am $ECO_2$-Manager. Da nur eine begrenzte Auflösung am Display zur Verfügung steht, wird die Auswertung nur für alle Sensoren gemeinsam angeboten. Es wird jeweils die gesamte Summe aller Sensoren angezeigt. Wird eine detaillierte Auswertung einzelner Sensoren gewünscht, kann dies am PC mit dem entsprechenden Java-Programm erledigt werden (siehe Kapitel 6).

#### Bedienung der Auswertung

Beim Start der Auswertung wird ein schwarzer Bildschirm mit den Wahlmöglichkeiten **Month**, **Week** und **Day** angezeigt und ein Hinweis, dass man einen Zeitraum auswählen soll. Darunter kann man sich schnell einen Überblick verschaffen, wie viel vom eingestellten Monatslimit noch übrig hat. Hier wird die Gesamtsumme des Monats angezeigt - hat man noch eine Reserve für den $CO_2$-Verbrauch, wird die noch übrige Menge in grün angezeigt, hat man das Limit bereits überschritten, wird die zu viel ausgestoßene Menge in rot angezeigt.

Durch einen Klick auf einen der oben angezeigten Buttons startet das System die Datenabfrage aus der Datenbank für die Darstellung des entsprechenden Diagramms. Dies kann je nach ausgewählter Periode und Anzahl an Datensätzen einige Zeit in Anspruch nehmen. Da dies in einem separaten Thread geschieht, wird die grafische Oberfläche nicht komplett geblockt, es kann allerdings zu kurzen Aussetzern bzw. Verzögerungen bei der Zeit oder der Reaktion auf andere Eingaben kommen.

Während dem Berechnen der Werte wird am Display "Processing, please wait..." angezeigt, bis die Daten berechnet wurden und anschließend wird die Grafik auf das Display gezeichnet. Standardmäßig wird der aktuelle Monat, die aktuelle Woche oder der aktuelle Tag dargestellt, der angezeigte Zeitraum kann rechts oben abgelesen werden. Will man einen anderen Zeitraum wählen, kann dies mit Hilfe von **Touch-Gestures** umgesetzt werden. Eine Gesture wird erkannt, wenn der Stift nicht gleich wieder vom Touchscreen entfernt wird, sondern der Druck aufrecht gehalten und der Stift in eine Richtung bewegt wird. Folgende Gestures werden bei den Auswertungen erkannt:

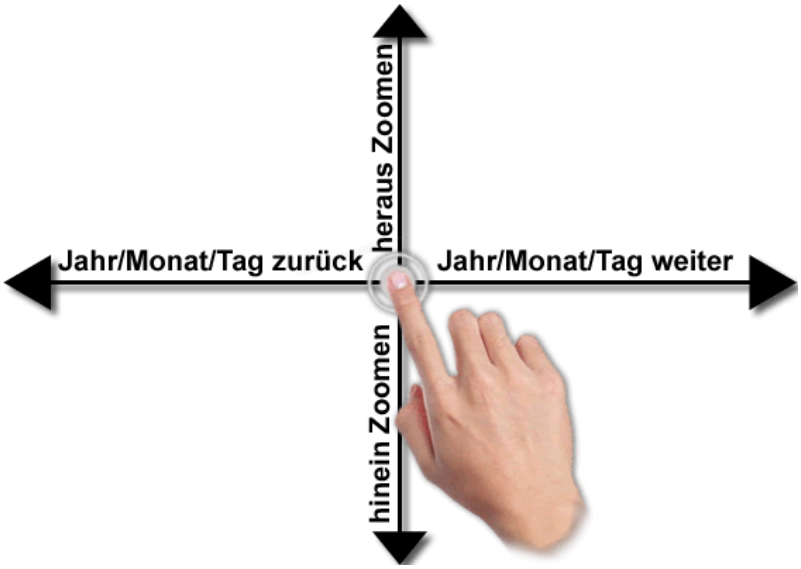



Abbildung 3.9.: Touch-Gestures im Auswertungsfenster

- **Wischen nach rechts:** Mit dieser Bewegung wird das vorherige Monat/die vorherige Woche/der vorherige Tag angezeigt. Es wird also 1 **Monat zurück geblättert**. Zu beachten ist, dass der Ladevorgang wieder einige Zeit in Anspruch nimmt, da die Daten für den geänderten Zeitraum erst neu berechnet werden müssen.
- **Wischen nach links:** Dies hat den umgekehrten Effekt als das Wischen nach rechts. Es wird der nächste Monat/die nächste Woche/der nächste Tag dargestellt **(1 Monat vor geblättert)** und wird ebenfalls wieder neu berechnet. Ist man im aktuellen Monat (Systemzeit) oder der aktuellen Woche angelangt, ist es nicht mehr möglich, weiter in die Zukunft zu scrollen, da dort keine Daten vorhanden sein können.
- **Wischen nach unten:** Diese Bewegung ermöglicht es, in das **Diagramm hinein zu zoomen**. Dies kann nur 1x geschehen, es wird aber für den angezeigten Zeitraum der optimale Zoom gewählt. Dabei wird die Y-Achse so skaliert, dass der minimale vorhandene Wert möglichst weit unten, der maximal vorhandene Wert möglichst weit oben liegt. Der minimale Wert pro Tag wird allerdings ohne Null-Werte berechnet, da ansonsten der Zoom keinen Sinn machen würde.
- **Wischen nach oben:** Der **Zoom wird wieder aufgehoben**. Der minimale Wert der Y-Achse wird wieder auf Null gesetzt, der maximale Wert bleibt gleich.

Wie die Auswertung am Display des Systems aussehen kann wird in Abbildung 3.10 gezeigt. Hier wurde als Beispiel die Monatsansicht für April 2012 gewählt und der Zoom aktiviert (Y-Achse beginnt nicht bei 0).

#### Technische Umsetzung

Im Konstruktor des **ReportingWindow** (Klassendiagramm siehe Abbildung 3.11) werden die beiden Buttons generiert und auf einem **Canvas** positioniert. Entsprechende


Abbildung 3.10.: Beispiel Monatsauswertung am Embedded System

Event-Handler für die Buttons und die Gestures werden ebenfalls dort registriert. Des Weiteren wird der Text zum Anzeigen der aktuell dargestellten Periode und ein **Image** angelegt und positioniert, auf welchem der Graph später gezeichnet wird. Die Initialisierung von diversen Membervariablen der Klasse wird ebenfalls noch im Konstruktor erledigt.

Im Konstruktor wird auch der Thread **displayMonthSum** aufgerufen, welcher die Monatssumme für das aktuelle Monat berechnet, es wird dazu die gleiche Methode (*recalculateValues_*) verwendet, die auch für das Diagramm angewandt wird. Angezeigt wird am Display die Monatssumme, der aktuell verbrauchte Wert und die Differenz der beiden Werte. Je nachdem ob eine Unterschreitung oder Überschreitung des Ausstoßes an $CO_2$ berechnet wurde, wird der Text der Differenz aus Limit und Verbrauch in grün oder rot dargestellt.

Wird auf einen Button geklickt, wird die Methode *buttonClicked* aufgerufen. Dort wird zwischen den drei Buttons unterschieden:

- **DayReport:** Hier wird die Tagesauswertung aktiviert. Die Variable *daysToDisplay_* für die Angabe der anzuzeigenden Tage wird auf einen Tag gesetzt. Somit wird ein Tag angezeigt, in dem Fall mit Stunden in der X-Achse.
- **WeekReport:** Wird eine wochenweise Auswertung gewünscht, wird der letzte Sonntag durch decrementieren der Tage gesucht und als erster Tag in der Membervariable *startingDay_* vermerkt. Die Anzahl der Tage die angezeigt werden sollen wird in *daysToDisplay_* auf 7 gesetzt.
- **MonthReport:** Bei einer monatlichen Auswertung wird der erste Tag des Monats als Starttag gespeichert und die Anzahl der Tage auf die Anzahl der Tage im Monat gesetzt. Mit Hilfe des Properties *DaysInMonth* der Klasse **DateTime** kann die Anzahl festgestellt werden.

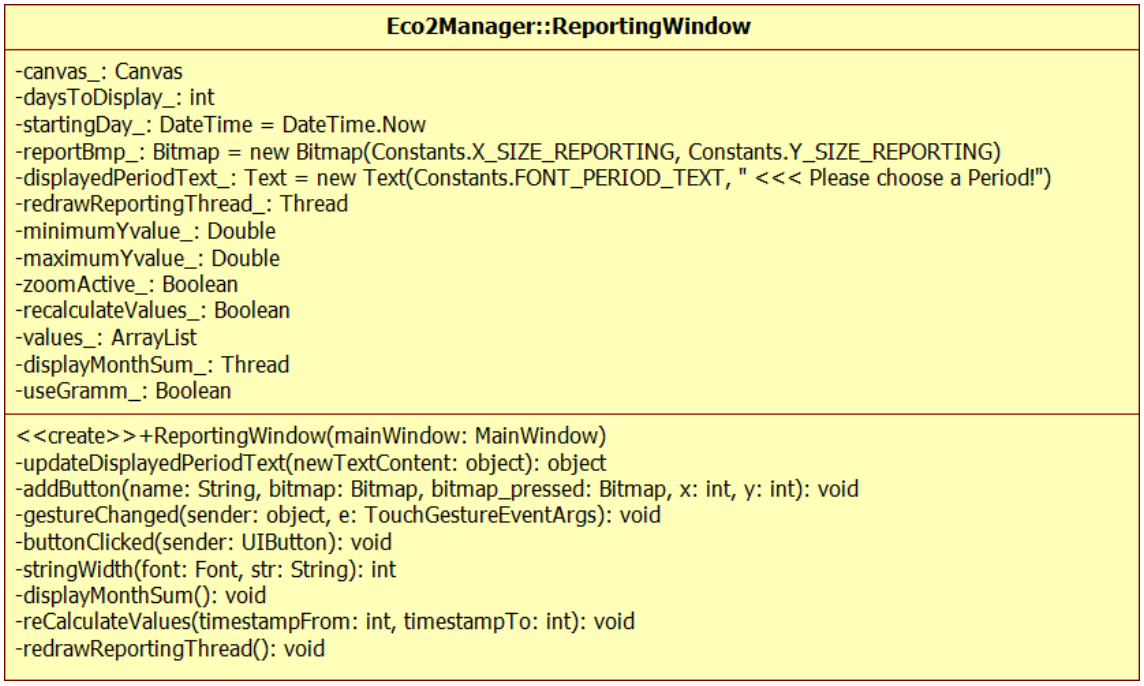



Abbildung 3.11.: Klassendiagramm ReportingWindow

Ebenso wird bei allen drei Buttons die Variable *recalculateValues_* auf `true` gesetzt, da die Werte für die Woche oder das Monat neu berechnet werden müssen. Um das Zeichnen des Graphen zu starten wird der Thread **redrawReportingThread** gestartet, sollte dieser nicht mehr aktiv sein. Ein doppeltes aktivieren wird verhindert, da es ansonsten zu Problemen mit der Ausführung kommt (paralleler Zugriff auf Variablen).

Beim Zeichnen einer Gesture am Touchscreen wird die Methode *gestureChanged* mit den *TouchGestureEventArgs* aufgerufen, welche Informationen darüber geben, welche Gesture genau erkannt wurde. Egal welche Gesture, wird der Sound für das Wischen am Display abgespielt und die Gesture anschließend unterschieden in:

- **Gesture left:** Es wird die vorige Woche/der vorherige Monat/der vorherige Tag angezeigt. Handelt es sich um eine Woche, müssen nur sieben Tage vom Starttag subtrahiert werden, handelt es sich um ein Monat (>= 28 Tage), muss der erste Tag des Monats neu berechnet werden. *recalculateValues_* wird auf `true` gesetzt, da die Datenwerte neu berechnet werden müssen.
- **Gesture right:** Im Prinzip wird hier gleich vorgegangen, allerdings wird eine zusätzliche Abfrage eingefügt, welche sicher stellt, dass keine Woche/kein Monat/kein Tag nach dem aktuellen Datum angezeigt werden kann.
- **Gesture down:** Hier wird nur *zoomActive_* auf `true` gesetzt, um den Zoom zu aktivieren. Eine Neuberechnung der Datenwerte ist hier nicht erforderlich.
- **Gesture up:** Der Zoom wird wieder zurückgesetzt. Eine Neuberechnung der Werte ist hier ebenfalls nicht erforderlich.

Zusätzlich muss bei jeder Gesture der **redrawReportingThread** wieder neu gestartet werden.

#### redrawReportingThread

Das Herzstück der Auswertungen stellt dieser Thread dar. Hier werden einerseits die Daten der Sensoren zusammengefasst und andererseits wird hier auch die Liniengrafik für das Diagramm erstellt. Zu Beginn des Threads findet sich schon eine Besonderheit. Hier wird ein Delegate verwendet, um dem UI-Thread einen neuen Text für die angezeigte Periode zu übergeben. Dies erfolgt mit Hilfe des Dispatchers, welcher den Aufruf automatisch per Delegate weiterleitet und sich um die Synchronisation kümmert. Genauer beschrieben werden Delegates und der Dispatcher in Abschnitt 2.3.

Im Anschluss folgt simple Mathematik. Es werden Minimal- und Maximalwerte für X- und Y-Achse (in Pixeln) und der Abstand zwischen den einzelnen Markierungen auf den Achsen berechnet. Für die Abfrage in der Datenbank wird auch noch der Timestamp des Starttages (00:00:00 Uhr) und des Endtages (23:59:59 Uhr) berechnet. Um zu visualisieren, dass ein Berechnung erfolgt, wird "Processing, please wait..." auf das zuvor vorbereitete und gelöschte Image für den Graphen geschrieben. Der weitere Ablauf kann in zwei Bereiche unterteilt werden:

- **Neuberechnung sämtlicher Werte (ausgelagert in Methode *reCalculateValues*):** Dies wird nur ausgeführt, sollte die Variable *recalculateValues_* auf `true` gesetzt sein, ansonsten wird die Berechnung übersprungen.
  Es werden alle Sensoren mittels eines Querys aus der Datenbank für die Einstellungen gelesen und in die ArrayList *sensors* als **SensorElement** eingefügt. Bei der Klasse **SensorElement** handelt es sich um eine reine Hilfsklasse zum Speichern von Daten in einem Objekt, wobei die sensorID, die valueID, der $CO_2$Offset und der $CO_2$Faktor dort gehalten werden. Im nächsten Block werden die Werte aller Sensoren von einem Tag oder einer Stunde aufaddiert. Dazu wird jeweils für einen Sensor ein Query abgesendet, welches die Summe aus den Werten des Sensors an dem entsprechenden Tag berechnet. Dies wird dann zur Tagessumme addiert. Sobald dies für alle Tage und Sensoren erledigt wurde, stehen die Werte in der Variable *values_* zur Verfügung. Während dem Speichern der Summen wird noch der maximale und der minimale Wert mitgespeichert, um eine entsprechende Skalierung der Y-Achse durchführen zu können.

- **Graph neu zeichnen:** Nach dem Berechnungsblock wird der Graph neu in das entsprechende Image gezeichnet. Dazu werden als erstes die Linien des X-Gitters und die X-Achsen-Markierungen gezeichnet und die Beschriftung der X-Achse hinzugefügt. Die gleiche Prozedur wird auch für die Y-Achse durchgeführt, die beiden Achsen selbst werden erst dann darüber gelegt. Die Koordinaten der Y-Achse verlaufen von oben nach unten, die Achse muss also quasi "am Kopf stehend" berechnet werden, was oft zu Verwirrungen führen kann.

  Die Liniengrafik wird in der folgenden for-Schleife gezeichnet. Dazu werden jeweils der Start-Y-Wert und der End-Y-Wert eines Linienabschnitts berechnet und die Linie über das vorhandene Gitternetz gezeichnet.

Die Farben der Achsen, der Linie und die Anzahl Unterteilungen etc. können in der Klasse **Constants** eingestellt werden.

### 3.4.3. ActorWindow

Dieses Fenster wird zum Schalten angeschlossener Aktoren verwendet. Derzeit wird *Ein-Aus-Funktionalität* unterstützt, auf Wunsch sind auch andere Anwendungen, wie Dimmer und dergleichen möglich, es müsste nur das Frontend verändert werden, übertragen werden bereits Integer-Werte, welche auch Dimmer, Ansteuerung von Analog-Digital-Wandlern etc. ermöglichen.

Zum Schalten der Aktoren wird eine *Listbox* verwendet, in der alle in der Datenbank *Actors* (siehe Unterabschnitt 4.2.4) verfügbaren Aktoren gelistet werden und für jeden Eintrag der Button **ON** und **OFF** zur Verfügung steht. Mit einem Klick auf den entsprechenden Button wird das Signal an den Sensor gesendet.

Zur Umsetzung dieser Funktionalität wurden zwei Klassen verwendet. Die Klasse **ActorWindow** ist für Erstellung einer *Listbox* und der entsprechenden *ActorListBoxItems* verantwortlich. Für jeden Aktor wird ein solches Item erstellt und der Listbox hinzugefügt. Die *Listbox* wird zusätzlich in einen *Scrollviewer* gepackt, welcher ein rauf und runter Scrollen ermöglicht. Die Funktion des Scrollens wurde den *Touch-Gestures "Up" und "Down"* zugewiesen. Diese schieben den Inhalt des Fensters jeweils eine Zeile hinauf oder hinunter, wobei die Höhe dieser Zeile in der Konstante *SCROLLING_AMOUNT* festgelegt werden kann.

Die Anordnung der Buttons für ON und OFF wird im **ActorListBoxItem** festgelegt. Für jeden Eintrag wird ein horizontales *StackPanel* angelegt, auf diesem wird zuerst der Text mit SensorID und ActorID platziert, anschließend daran jeweils ein ON und OFF Button. Um bei einem Klick auf einen solchen Button entsprechend reagieren zu können, wird die Methode *buttonClicked* implementiert, welche `0x00` für OFF und `0x01` für ON an den entsprechenden Aktor sendet. Sollte beim Senden ein Fehler auftreten (Socket-Fehler oder XBee-Fehler), wird neben den Buttons eine Fehlermeldung eingeblendet. Diese wird gelöscht, sobald ein Senden an den Aktor erfolgreich war. Eine Überprüfung, ob der entsprechende Aktor auch die gewünschte Aktion ausgeführt hat erfolgt nicht, dazu wäre ein Feedback von diesem notwendig.

Der Aufbau der beiden verwendeten Klassen kann Abbildung 3.12 entnommen werden.

### 3.4.4. SettingsWindow

Das **SettingsWindow** stellt ein simples Fenster da, über welches diverse Einstellungen getroffen werden können. Dies umfasst derzeit folgende Einstellungen:

- Helligkeit verringern/erhöhen
- Speicherintervall verringern/erhöhen
- Lautstärke verringen/erhöhen
- Monatslimit verringern/erhöhen

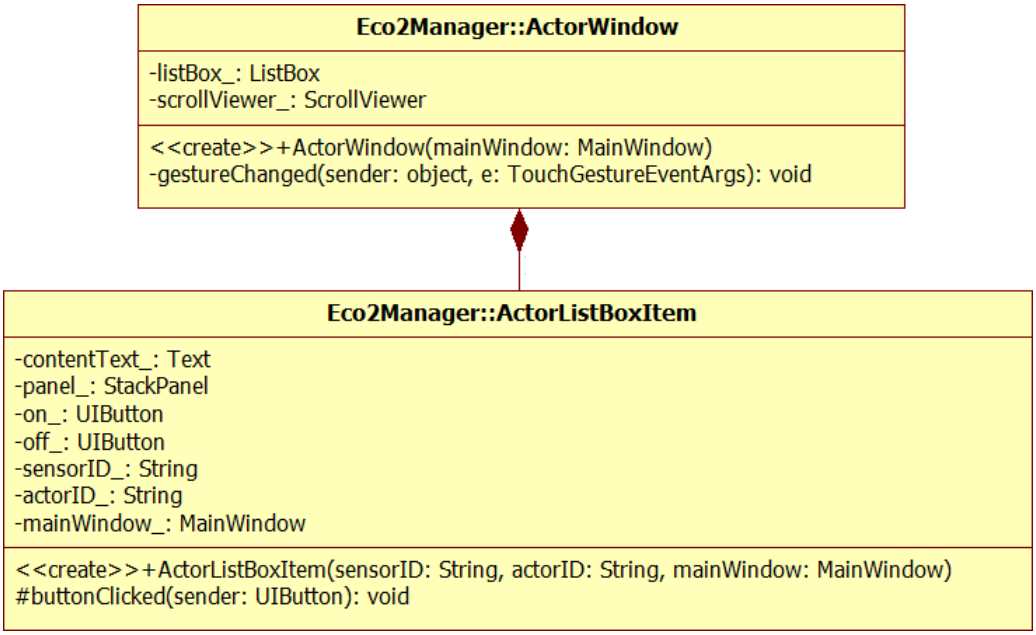



Abbildung 3.12.: Klassendiagramm ActorWindow und ActorListBoxItem

Um dies umzusetzen, werden die entsprechenden Up- und Down-Buttons positioniert und über die Methode *addButton* jeweils ein entsprechender Event-Handler für jeden **UIButton** festgelegt. Zwischen den Buttons wird noch der Text mit den aktuellen Einstellungen positioniert. Drückt man einen Button, wird der entsprechende Wert über die zuständige Klasse verringert oder erhöht und über die Methode *renewText* der Text mit den aktuellen Werten aktualisiert. Um die Maximal- und Minimalwerte kümmern sich jeweils die anderen Klassen selbst, daher muss dies hier nicht beachtet werden. Die Methoden und Membervariablen können auch Abbildung 3.13 entnommen werden.

### 3.4.5. Speichermedium

In den Anforderungen für das Projekt wurde festgelegt, dass mehrere Speichermedien zur Wahl stehen sollen. Am ChipworkX Development-Board steht sowohl ein SD-Karten-Slot als auch ein USB Anschluss und ein interner Speicher zur Verfügung. Daher wird eine Speicherung mit diesen drei Varianten ermöglicht und im Fenster **StorageDeviceWindow** können diese ausgewählt werden. Der interne Speicher (NAND-Speicher) kann immer gewählt werden, SD-Karte und USB-Stick nur, wenn diese auch wirklich angesteckt und korrekt erkannt wurden.

Für die Auswahl werden Objekte der Klasse **UIRadioButton** erstellt und untereinander angeordnet. Beim Erstellen dieser wird ein Name vergeben und ein *CheckedChangedEventHandler* hinzugefügt, um auf die Änderung einer Auswahl reagieren zu können. In der Methode *addRadioBox* der Klasse **UIRadioButton** wird auch ein Beschreibungstext rechts neben den Buttons positioniert.

Sobald ein Objekt ausgewählt wurde, wird in der Methode *radioBoxChanged*, welche als Listener für Änderungen hinzugefügt wurde, der richtige **UIRadioButton** ausgewählt

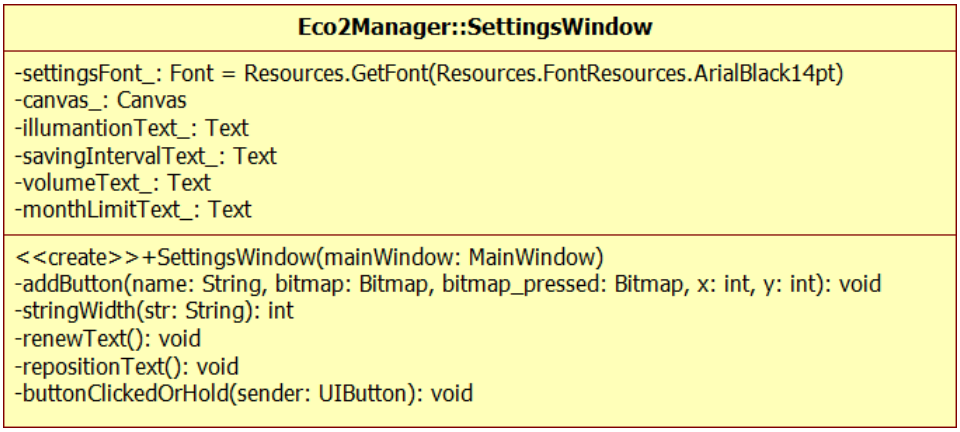



Abbildung 3.13.: Klassendiagramm des SettingsWindow

und dem **UsbSdController** mitgeteilt, welches Medium gewählt wurde. Dazu wird der Name des ausgewählten Punktes verglichen, der richtige Button aktiviert und die restlichen Buttons werden deaktiviert. Dies geschieht in der Methode *selectCorrectRadioBox*.

Ein weiterer wichtiger Bestandteil des Fensters sind die Buttons zum Auswerfen der Speichermedien. Wurde eine SD-Karte oder ein USB-Stick erkannt, werden im unteren Bereich des Fensters Buttons eingeblendet, welche es ermöglichen, die Speichermedien auszuwerfen. Dies wird wiederum über den **UsbSdController** erledigt. Sobald der Button gedrückt wurde, ist die Auswahl dieses Speichermediums nicht mehr möglich, sollte das ausgeworfene Speichermedium noch gewählt gewesen sein, wird automatisch auf den internen Speicher zurückgegriffen. Bei der Umsetzung der Funktionen zum Auswerfen wurde auf eine möglichst große Fehlertoleranz für Fehlbedienungen Wert gelegt. Wird ein Medium entfernt, ohne vorher ausgeworfen zu werden, werden die Datenverbindungen auch nach Möglichkeit beendet, sollte allerdings gerade ein Schreibzugriff aktiv gewesen sein, kann dies zur Zerstörung des Dateisystems führen. Daher wird das Auswerfen eines Mediums dringend empfohlen.

Ein weiterer Button ermöglicht es, die Sensordaten auf dem gewählten Medium zu löschen, die Einstellungen der Sensoren und Aktoren bleiben erhalten. Ausgewertet werden alle Buttons über die Methode *buttonClicked*, eine Übersicht über die Klasse **StorageDeviceWindow** wird in Abbildung 3.14 dargestellt.

Wie die Wahl des Speichermediums und der weitere Ablauf der Massenspeicherverwaltung funktioniert wird in Abschnitt 4.1 beschrieben.

### 3.4.6. Uhrzeit einstellen

Das **ClockSettingWindow**, welches, wie alle Fenster, vom **AbstractWindow** abgeleitet wird, wird für das Einstellen der Uhrzeit verwendet. Da die Bedienung des $ECO_2$-Managers rein allein über den Touchscreen erfolgen soll, wurde ein eigenes Fenster zum Einstellen der Uhrzeit erstellt. Der ungefähre Aufbau ist in Abbildung 3.15 dargestellt, Screenshots vom Display des ChipworkX-Systems waren leider nicht möglich, daher werden nachgebaute Grafiken verwendet, die geringfügig von der tatsächlichen Darstellung abweichen können.

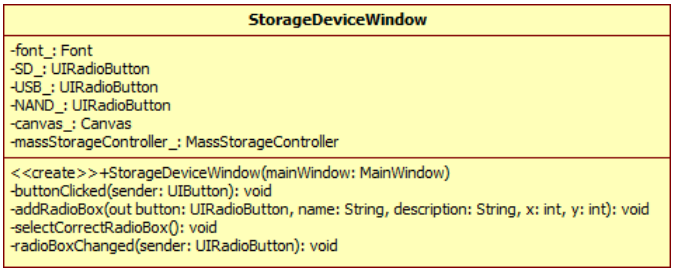



Abbildung 3.14.: Klassendiagramm des StorageDeviceWindow

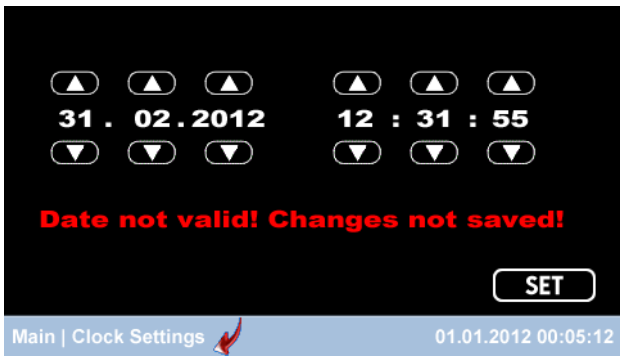



Abbildung 3.15.: Nachbildung des ClockSettingsWindow

Die einzelnen Buttons und Textelemente werden auf einem *Canvas* positioniert, dazu werden im Konstruktor zuerst alle Texte in einer Schleife an die richtige Position gebracht, vorerst bleiben die Texte aber leer. Mit der Methode *setTimeToMembers* wird die Zeit aus der RTC ausgelesen und in die entsprechenden Membervariablen der Klasse geschrieben, um diese dort ändern zu können. Sollte nach dem letzten Tausch der Batterie noch kein Datum eingegeben worden sein, wird der 1.1.2012 angezeigt, damit nicht von 1900 aus das Jahr gesetzt werden muss.

Nach dem Setzen der Membervariablen wird die jetzt vorliegende Zeit in die vorher positionierten Textfelder übernommen, im Anschluss werden die Up- und Down-Buttons positioniert, diese werden mit Hilfe der Methode *addButton* und der Klasse **UIButton** hinzugefügt. Zum Schluss werden mit *addStaticText* die Trennzeichen für Datum und Uhrzeit eingefügt, der SET-Button platziert und das *Canvas* als Kindelement zum Fenster hinzugefügt, womit die Positionierung der gesamten Elemente abgeschlossen ist.

Wird jetzt ein Button gedrückt, wird das *TouchDownEvent* an die Methode *buttonClicked* weitergeleitet und der Name des entsprechenden Buttons wird mit übergeben. Für Datum und Uhrzeit wird das entsprechende Increment oder Decrement durchgeführt, mit Beachtung des Überlaufs. Beim Drücken von `SET` wird das Datum überprüft, indem ein *DateTime-Element* mit dem neuen Datum erstellt wird und dies mit dem eingegebenen Tag und dem eingegebenen Monat verglichen wird. Sollte ein ungültiges Datum, wie der 31.2. eingegeben werden, stimmen Tag und Monat nicht überein, und es wird ein Fehler erkannt. Dieser Fehler wird durch einen roten Schriftzug unterhalb der Eingabe signalisiert. Wurde ein gültiges Datum eingegeben, werden die Membervariablen in die RTC und

die Systemzeit übernommen und das Datum und die Uhrzeit damit geändert. Bei jedem Tastendruck wird noch die Methode *updateTimeText* aufgerufen, um das Drücken des Buttons durch ändern des entsprechenden Textes zu signalisieren. Das Klassendiagramm für diese Klasse kann der Abbildung 3.16 entnommen werden.

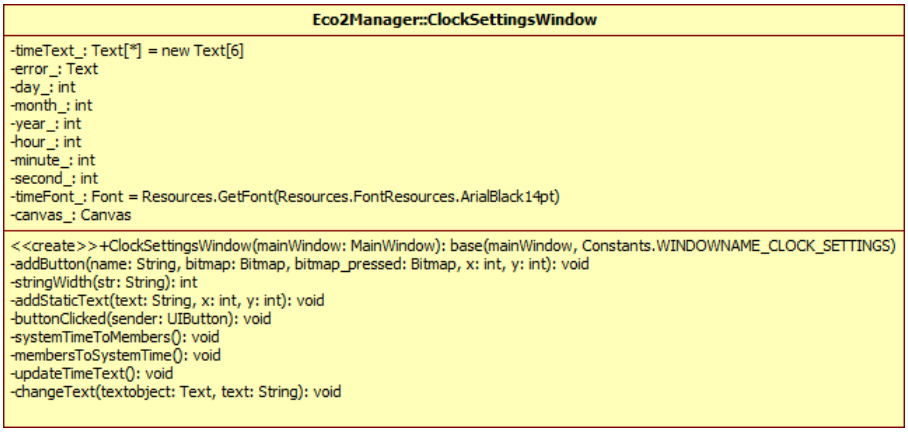



Abbildung 3.16.: Klassendiagramm des ClockSettingsWindow

### 3.4.7. Touchscreen kalibrieren

Da beim .NETMF zwar Funktionen für die Kalibrierung vorhanden sind, aber die Visualisierung davon selbst ausgeführt werden muss, wurde, ausgehend vom Beispiel von *Colin Millers Blog "Changing Display Size - the UI Thread, EWR, TouchCalibration"* [Mil10a], eine Visualisierung für die Kalibrierung des Touchscreens geschrieben. Aufgerufen wird diese durch das entsprechende Symbol am **MainWindow** (siehe Abbildung 3.5). Klickt man auf dieses Symbol wird das **MainWindow** durch das **CalibrationWindow**, welches mit den entsprechenden Hilfsklassen in Abbildung 3.17 dargestellt ist, ersetzt und die Kalibrierung muss bis zum Schluss durchgeführt werden, eine Unterbrechung ist nicht möglich. Ein Überblenden des **MainWindow** ist hier nicht möglich, da keine Elemente, bis auf das **CalibrationWindow** vorhanden sein dürfen, ansonsten funktioniert die Kalibrierung nicht. Das Fenster nimmt außerdem den gesamten Platz des Touchscreens ein und wird mit einem schwarzen Hintergrund versehen.

Noch im Konstruktor der Klasse **CalibrationData** werden die vom .NET Micro Framework vorgegebenen Kalibrierungspunkte geladen und entsprechend aufbereitet. Das Display wird dann mit Aufruf von *StartCalibration* in einen anderen Modus zum Sammeln der Koordinaten versetzt, da durch eine falsche Kalibrierung auch Punkte außerhalb des üblichen Bereichs von 480x272 Pixel auftreten können und verarbeitet werden müssen (RAW-Modus). Mit der Methode *OnRender* in der Klasse **CalibrationWindow** wird jeweils ein Kalibrierungspunkt als Kreuz auf das Display gezeichnet und die RAW-Koordinaten beim Drücken auf den Touchscreen wieder an die Klasse **CalibrationData** übergeben. Wurden alle Kalibrierungspunkte aufgenommen, werden diese an das .NETMF übergeben und die neuen Kalibrierungsparameter werden mit dem Aufruf der Methode *SetCalibration* übernommen. Die mathematische Berechnung der Korrekturdaten erfolgt automatisch im Framework.

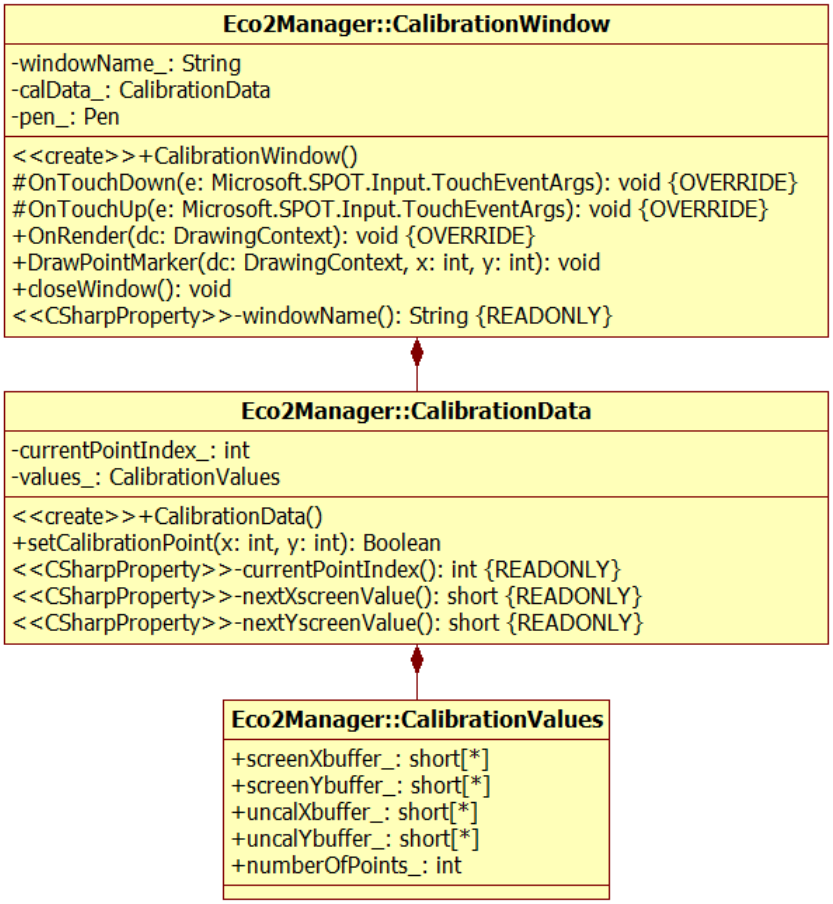



Abbildung 3.17.: Klassendiagramm des CalibrationWindow mit Hilfsklassen

Um die Kalibrierungsdaten beim nächten Neustart des Systems nicht zu verlieren, werden diese mittels einer *ExtendedWeakReference* nach Abschluss der Kalibrierung gespeichert (Klasse **CalibrationData**). Damit bleiben die Daten erhalten und können nach einem Neustart wieder geladen werden. In der Klasse **Program** wird versucht, diese Referenz wiederherzustellen, sollten Daten vorhanden sein, wird die gespeicherte Kalibrierung beim Start des Systems wieder auf das Display angewandt.

Nach Abschluss der Kalibrierung wird das **CalibrationWindow** wieder durch das **MainWindow** ersetzt, wodurch der normale Startbildschirm wiederhergestellt wird.

### 3.4.8. Shutdown

Um ein sauberes Beenden des Systems zu ermöglichen, wurde ein Button zum "Herunterfahren" des Systems hinzugefügt. Dabei ist wichtig, dass die Datenverbindungen sauber beendet werden, damit das Dateisystem nicht beschädigt wird. Bei einem Klick auf das Symbol zum Ausschalten des Systems, werden alle Datenverbindungen beendet und die Dateisysteme der Wechselspeichermedien ausgehängt und das Gerät beendet. Als Visualisierung wird ein schwarzes Fenster über den gesamten Bildschirm gelegt und das System

kann ausgeschalten bzw. ausgesteckt werden.

## 3.5. grafische Bausteine

### 3.5.1. UIButton

Die Klasse **UIButton** (siehe Abbildung 3.18) wurde erstellt, um eine einfache Möglichkeit bieten zu können, Buttons zu erstellen. Dabei wird dem Konstruktor ein Name, das Bitmap für den Button und ein Bitmap für den gedrückten Button übergeben. Diese übergebenenen Werte werden in den Membervariablen der Klasse für die spätere Verwendung gespeichert. Wichtig dabei ist, dass die Button-Grafik und die Grafik für den gedrückten Button die gleiche Größe aufweisen müssen, die Positionierung des Buttons wird der aufrufenden Klasse überlassen.

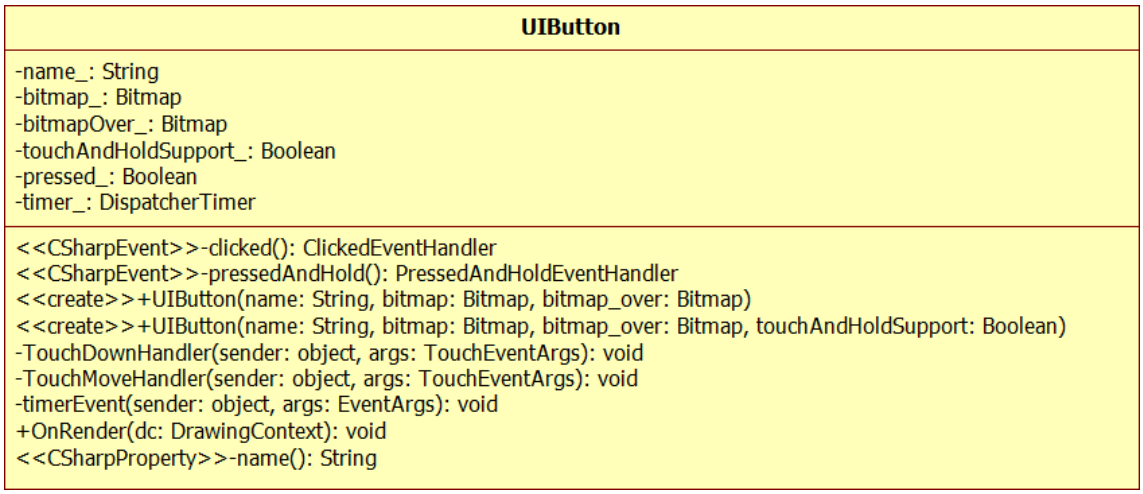



Abbildung 3.18.: Klassendiagramm UIButton

Das *TouchDown-Event* wird der Methode *TouchDownHandler* zugewiesen, um Actions am Touchscreen innerhalb des entsprechenden Elements zu erkennen. Wird der Button gedrückt, wird für eine gewisse Zeitdauer (*Constants.UIBUTTON_PRESSED_TIMEOUT*) die Grafik für den gedrückten Button angezeigt, anschließend wieder das Standard-Bild. Gleichzeitig wird der Sound für einen Klick abgespielt. Das Zurücksetzen der Grafik ist leider nicht mit einem *TouchUpHandler* möglich, da die Grafik sonst erhalten bleibt, sobald das *TouchUp-Event* außerhalb des Buttons stattfindet. Gezeichnet wird die Grafik des Buttons in der Methode *OnRender*, welche zwischen einem nicht gedrückten und einem gedrückten Button mittels der Variable *pressed* unterscheidet, welche im Handler gesetzt wird. Das Event, wenn ein Button gedrückt wurde, steht als *ClickedEventHandler* zur Verfügung, wobei der **UIButton** selbst als Sender mitgegeben wird und per Property der Name des entsprechenden Buttons bezogen werden kann. Damit wird ermöglicht, dass in der aufrufenden Klasse eine Methode alle Aktionen aller Buttons abarbeitet.

Zusätzlich wird das **TouchMove-Event** verwendet, um länger gedrückte Buttons zu erkennen. Diese Funktionalität kann im Konstruktor zusätzlich als letztes Argument (*touchAndHoldSupport*) angegeben werden. Wird dieses auf *true* gesetzt, wird bei jedem Aufruf

den *TouchMoveHandlers* das Event *pressedAndHold* aufgerufen. Beim Start des gesamten Systems (siehe Abschnitt 3.3) kann festgelegt werden, in welcher Geschwindigkeit *TouchMoveEvents* gesendet werden, entsprechend dieser Einstellung werden die Events weiter geleitet. Diese Einstellung kann in der Klasse *Constants* mit der Variable *UIBUTTON_PRESS_AND_HOLD_INCREASE_INTERVAL* in Millisekunden eingestellt werden. Derzeit ist dort ein Wert von 400ms als Standard eingestellt.

### 3.5.2. UIRadioButton

Mit Hilfe dieser Klasse (Übersicht siehe Abbildung 3.19), welche von **UIElement** abgeleitet wurde, können Auswahlfelder erstellt werden. Diese Auswahl ist einem *Radiobutton*, wie man ihn von Webseiten kennt, nachempfunden. Grundlage bieten zwei Grafiken - eine für den ausgewählten und eine für den nicht ausgewählten Punkt.

Innerhalb der Klasse werden diverse Methoden definiert:

- **TouchDownHandler:** Wird auf einen Button geklickt, wird dieser aktiv gesetzt, der Sound für einen Klick abgespielt und der externe *CheckedChangedEventHandler* aufgerufen - dort kann festgelegt werden, welche Aktionen auszuführen sind. Zur Unterscheidung der einzelnen Radiobuttons kann der Name ausgelesen werden. Das Handling, dass alle anderen Buttons deaktiviert werden (oder je nach Wunsch auch nicht), muss extern in der entsprechenden Methode erfolgen.
- **OnRender:** Beim Rendern des Buttons wird je nach Status der Variable *checked_* das richtige Bild gesetzt.
- $<$ *Property* $>$ **checkedValue:** Dieses Property setzt oder liest die Variable *checked_*. Wird diese gesetzt, wird zusätzlich *Invalidate* aufgerufen, um ein neues Rendern des Radiobuttons hervorzurufen, damit sichergestellt ist, dass die richtige Grafik geladen wird.

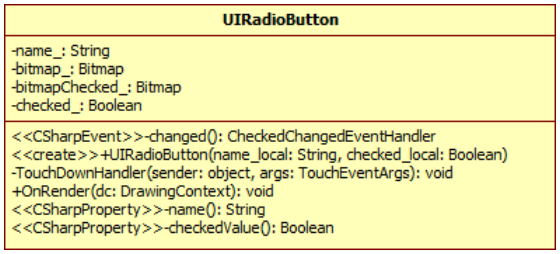



Abbildung 3.19.: Klassendiagramm UIRadioButton

## 3.6. Sounds

Um Eingaben am Touchscreen zu bestätigen wurde die Möglichkeit Sounds abzuspielen zum Projekt hinzugefügt. Ein entsprechender MP3/Midi-Decoder-Chip (VS1053) ist am ChipworkX-Development-Board bereits integriert, ebenso sind zwei Lautsprecher vorhanden. Da die von GHI Electronics zur Verfügung gestellten Treiber nicht richtig funktionierten, wurden die Treiber von *Thomas W. Holtquist* (`http://www.skewworks.com/`) verwendet, welche mehr Funktionen unterstützt und ohne Probleme funktioniert. Die Library wird mit der Datei **VS10XX.cs** eingebunden und der Treiber kann über den Namespace **Skewworks.Drivers** angesprochen werden.

### Klasse SoundControl

Ähnlich wie die Helligkeitssteuerung des Displays ist die Klasse für die Steuerung des Sounds ausgelegt. Es existiert eine EWR, welche die aktuelle Lautstärke beinhaltet (siehe Abbildung 3.20). In der Methode *initialize* wird zuerst der VS1053-Chip über den Treiber initialisiert, was hier nicht im Konstruktor erledigt werden kann, da zum Zeitpunkt des Ausführens des Konstruktors der Chip noch nicht angesprochen werden kann. Im Anschluss daran wird die EWR, sollte sie noch nicht vorhanden sein, angelegt und mit dem höchsten Wert (100) für die Lautstärke befüllt. Die Methode *updateValue* gibt den Wert an den Treiber weiter. Die Methoden *volumeUp* und *volumeDown* erhöhen bzw. senken die Lautstärke um den in den **Constants** definierten Wert pro Aufruf. Ebenso wird darauf geachtet, dass der Minimal- und Maximalwert nicht unter- beziehungsweise überschritten wird.

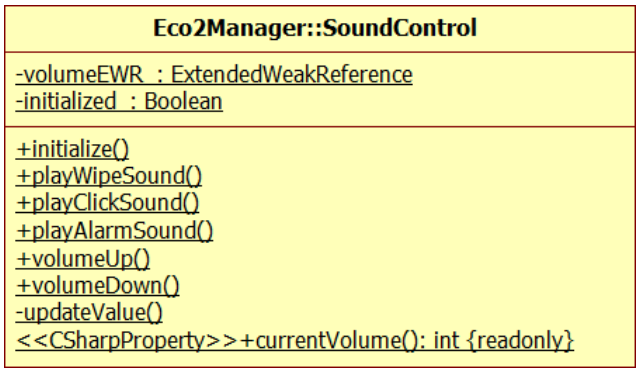



Abbildung 3.20.: Klassendiagramm SoundControl

### Sound abspielen

Um einen Sound abzuspielen, muss vom Treiber **VS10XX** die Methode *Play* mit der entsprechenden Sounddatei aufgerufen werden. Die Sounddatei muss unter den Resources hinzugefügt werden und kann dann von dort geladen werden. Um nicht jedes Mal die Referenz auf die Ressource angeben zu müssen, wurden in dieser Klasse drei Methoden geschrieben, die das Abspielen erledigen:

- **playWipeSound:** Spielt den Sound für Gestures am Touchscreen ab.
- **playClickSound:** Spielt den Sound für Klicks am Touchscreen ab.
- **playAlarmSound:** Spielt den Sound für einen Alarm ab.

# Kapitel 4.

# Speicherung der Daten

Für die Speicherung der Sensornamen, Aktoren und Daten, die von den Sensoren empfangen werden wurden, verschiedene Konzepte in Betracht gezogen, wobei in den Anforderungen festgelegt wurde, dass die Daten am $Eco_2$-Manager selbst gespeichert werden sollen. Eine Übertragung auf einen Client (zum Beispiel via Ethernet) steht nicht dauerhaft zur Verfügung und kann daher nicht verwendet werden. Eine externe Verbindung zu Servern etc. kann zwar zur Funktionalität hinzugefügt werden, die primäre Speicherung übernimmt aber dennoch der Manager.

Wie in Unterabschnitt 3.4.5 bereits beschrieben, stehen am ChipworkX-Development-Board ein SD-Karten-Slot und ein USB-Anschluss zur Verfügung, zudem hat das ChipworkX-Modul selbst einen internen NAND-Speicher (256MB), welcher auch als Datenspeicher verwendet werden kann. Um die Speicherung möglichst benutzerfreundlich zu gestalten, wird das Dateisystem auf den jeweiligen Speichermedien automatisch gemountet und das Medium kann dann über das entsprechende Fenster (siehe Unterabschnitt 3.4.5) ausgewählt werden. Die Funktionalität für die Verwaltung der Speichermedien wird in Abschnitt 4.1 genauer beschrieben. Eines haben alle Klassen gemeinsam - sie halten jeweils eine Referenz auf das **MainWindow**, damit der Zugriff auf die jeweilige andere Klasse und sonstige Komponenten möglich ist.

Es gibt verschiedene Arten von zu speichernden Daten:

- **Sensor- und Aktor-Informationen:** Diese Daten werden über den $Eco_2$-Client (siehe Kapitel 6) definiert und immer im **internen Speicher** gespeichert, da diese auch vorhanden sein sollen, wenn kein externer Speicher vorhanden ist. In regelmäßigen Abständen bzw. beim Entfernen des Massenspeichermediums werden die Informationen über die Sensoren auf das entsprechende Medium kopiert, um die Daten auch am PC auswerten zu können.
- **Sensordaten:** Die Daten, die von den Sensoren empfangen wurden, werden auf das **gewählte Speichermedium** gespeichert. Die Daten können auch mit dem Papierkorb-Symbol im **StorageDeviceWindow** bei Bedarf gelöscht werden. Die Sensor- und Aktor-Informationen bleiben dabei erhalten. Wird der **interne Speicher** für diese Speicherung gewählt, werden die Daten dort **maximal für eine Woche** aufgezeichnet, um eine Überfüllung zu verhindern.

Für die Art der Datenspeicherung wurden mehrere Möglichkeiten angedacht und zum Teil ausprobiert:

- **Eigene Dateistruktur, Speicherung in Dateien:** Der erste Gedanke, die Daten in einer eigens dafür angelegten Dateistruktur in Textdateien zu speichern wurde zu Beginn betrachtet und ausprobiert. Dabei wurde versucht, eine Klasse für die Daten zu schreiben und diese mittels XML in eine Datei zu schreiben. Dabei traten allerdings sehr rasch einige Probleme auf, beim Zugriff auf die Daten zum Beispiel müssten diese erst aus den Dateien geparst werden oder bereits zu Beginn in den Arbeitsspeicher geladen werden. Da dies eine Menge Overhead bedeutet, wurde die Methode nach den ersten Testversuchen wieder verworfen. Das Parsen zu Beginn war nicht sonderlich performant und das System würde zu lange für einen Startvorgang benötigen. Ebenso ist das Parsen in Threads nicht sinnvoll, da die Daten nach dem Start schnell zur Verfügung stehen sollen.

  Bei Datendateien mit mehreren Datensätzen traten zudem sehr rasch Probleme mit maximaler Pufferlänge und der Lesegeschwindigkeit beim Parsen auf. Dies würde sich durch verteilen auf mehrere Dateien sicherlich etwas in den Griff bekommen lassen, allerdings sehr umständlich und aufwändig. Am PC müsste dieser Parser ebenfalls gebaut werden, was zu einem nicht vertretbaren Mehraufwand jeglicher Entwicklung zum Auswerten der Daten führen würde. Dieser Ansatz wurde nach den ersten Versuchen nicht weiter Verfolgt, da er für zu aufwendig und langsam empfunden wurde.

- **Extended Weak Reference:** Eine weitere Möglichkeit wäre, die Daten mittels einer Extended Weak Reference (EWR) zu speichern. Die Daten würden dabei automatisch in den Flash-Speicher gesichert werden (siehe Abschnitt 2.3), allerdings hätte man keinerlei Kontrolle, welche Daten durch den Garbage Collector entfernt werden, sollte der Speicher voll werden. Bei Extended Weak References garantiert das System nicht, dass die Daten noch vollständig vorhanden sind, eine eigene Struktur ist eine zusätzliche Voraussetzung, welche für eine EWR sicherlich sehr kompliziert zu verwalten ist. Der Ansatz wurde daher nie umgesetzt oder ausprobiert.

- **SQLite:** Die wohl beste Variante ist die Speicherung der Daten mittels SQLite. Von *GHI Electronics* wird eine Library zur Verfügung gestellt, welche **SQLite** in Version 3.6.13 in verringertem Umfang unterstützt. SQLite legt seine Daten in einer Datei ab und Daten können mit normalen SQL-Queries gelesen, gespeichert und gelöscht werden. Die Daten werden dabei effizient in einer einzelnen Datenbankdatei abgelegt und können dort wieder gelesen werden. Die Verwaltung innerhalb der Datei wird dabei komplett von der Library übernommen, darum muss sich der Benutzer nicht kümmern. SQLite wird auch in den meisten anderen Programmiersprachen und Frameworks als Stand-Alone-Datenbanksystem unterstützt und implementiert. Da dies die sinnvollste Variante für die Datenspeicherung am $Eco_2$-Manager darstellt, wird diese Methode verwendet und in Abschnitt 4.2 genauer beschrieben.

## 4.1. UsbSdController

Diese Klasse wurde geschrieben, um das Handling der Massenspeichermedien zu erledigen. Der Benutzer soll sich nicht um das Mounten des Dateisystems oder die Erkennung der Speichermedium kümmern müssen, daher werden diese Dinge in dieser Klasse automatisch erledigt. Die Klassen **StorageDeviceWindow** (siehe Unterabschnitt 3.4.5) und **DataStorageController** (siehe Abschnitt 4.2) arbeiten sehr eng mit dieser Klasse zusammen. Die Klasse **DataStorageController** muss zwingend vor dem **UsbSdController** instanziert werden, da die Methoden von dieser Klasse bereits im Konstruktor benötigt werden! Instanziert werden beide Klassen im **MainWindow** um den Zugriff untereinander zu ermöglichen.

Im Konstruktor wird eine *ExtendedWeakReference* für das ausgewählte Speichermedium erstellt bzw. geladen, sollte bereits ein Wert im Speicher vorhanden sein. Dieser Wert gibt vor, welches Speichermedium ausgewählt war, als das System resettet oder ausgeschalten wurde. Dieses Speichermedium wird nach Möglichkeit wiederhergestellt. Sollte kein solcher Wert vorhanden sein wird, wie auch sonst, wenn ein gewähltes Speichermedium entfernt wurde, der interne Speicher (NAND) ausgewählt, da dieser immer vorhanden ist. Der NAND-Speicher wird auch gleich gemountet, bei Bedarf formatiert und in einer Membervariable gehalten, da dieser Speicher immer zur Verfügung stehen soll. Eine Übersicht der Klasse wird in Abbildung 4.1 dargestellt.

### 4.1.1. USBHostDevice-Events

Diese Handler dienen der Erkennung von Vorgängen am USB-Port. Wird ein Gerät eingesteckt, wird das **DeviceConnectedEvent** aufgerufen, welches an die Methode *USBHostDeviceConnected* weitergeleitet wird. Dort wird überprüft, ob es sich um ein Massenspeichermedium handelt, wenn nicht, ist das Gerät für diese Klasse uninteressant. Handelt es sich um ein neu erkanntes Speichermedium, wird ein *PersistentStorage-Objekt* erzeugt und in der dafür vorgesehenen Membervariable *usbMassPersistentStorage_* gespeichert. Zudem wird noch das Dateisystem dafür gemountet und das Medium formatiert, sollte sollte dies noch nicht geschehen sein.

Das **DeviceDisconnectedEvent** wird aufgerufen, wenn ein USB-Gerät entfernt wurde. Meistens ist es zwar schon zu spät, ein *Unmount* auf das Dateisystem auszuführen, es wird aber trotzdem versucht und das Objekt in der Membervariable wird verworfen und wieder auf *null* gesetzt. Idealerweise sollte ein *Unmount* bzw. das Auswerfen des Mediums vom User im **StorageDeviceWindow** erledigt werden, bevor das Medium entfernt wird. Dazu kann im **UsbSdController** die Methode *removeUsbMassStorageDevice* verwendet werden.

### 4.1.2. SDMountThread

Da es für die SD-Karte **keine Events** wie für die USB-Massenspeichermedien gibt, wird hier ein **Thread** eingesetzt, welcher in bestimmten Abständen im Hintergrund einen **Pin abfragt**, ob eine Speicherkarte vorhanden ist oder nicht. Dazu wurde im Konstruktor der *sdDetectPin_* definiert, welcher mit dem Schalter für die **Kartenerkennung** des SD-Kartenslots verbunden ist. Beim ChipworkX-Development-Board ist die der *Port B15*,

welcher mit einem Jumper am Connector *SV2* mit dem *SD DETECT* Pin verbunden werden muss (neben der Steckerleiste mit einem kleinen, weißen Balken markiert).

Der Thread liest am Anfang den Status vom Pin - dieser wird in der Variable *cardStatus* gespeichert, im Anschluss werden verschiedene Zustände unterschieden. Wenn eine Karte erkannt wurde, wird noch einmal einige Millisekunden gewartet und der Status erneut gelesen, um sicher zu gehen, dass es sich nicht um eine Störung handelt, bzw. die Karte derweil wirklich eingelegt ist (Entprellung). Anschließend werden folgende Zustände unterschieden:

- **Karte erkannt, noch kein *sdPersistentStorage_* vorhanden und *sdAutomountInactive_* nicht gesetzt:** Es wurde seit der letzten Ausführung des Threads eine Karte eingelegt, daher wird ein *PersistentStorage-Objekt* erstellt und das Dateisystem gemountet und formatiert, sollte es sich um ein unformatiertes Medium handeln. Zusätzlich wird *cardInserted_* gesetzt. Die Variable *sdAutomountInactive_* wird verwendet, um nach dem Auswerfen des Speichermediums durch den Benutzer das Dateisystem nicht automatisch wieder zu mounten.

- **keine Karte erkannt, aber ein *sdPersistentStorage_-Objekt* vorhanden:** Es wurde seit der letzten Ausführung des Threads die Karte entfernt. Es wird noch versucht ein Unmount auf das Dateisystem auszuführen und das Objekt *sdPersistentStorage_* wird verworfen und auf *null* gesetzt. Zudem wird *cardInserted_* wieder auf `false` gesetzt, damit kein weiterer Zugriff auf die Karte mehr erfolgt.

Im Anschluss an diese Unterscheidung wird noch, sollte die Karte entfernt worden sein, *cardInserted_* auf false gesetzt und *sdAutomountInactive_* wieder zurückgesetzt. Mit diesem Aktionen erkennt der Thread eingelegt Speicherkarten und mountet das Dateisystem, der Benutzer muss sich um dies nicht mehr kümmern.

Um händisch ein *Umount* für die SD-Karte auszuführen, kann die Funktion *removeSDCard* verwendet werden. Dabei wird die Variable *sdAutomountInactive_* gesetzt, um zu verhindern, dass nach dem Auswerfen des Mediums dieses sofort wieder erkannt wird. Zusätzlich können noch Event-Handler für eine erfolgreich gemountete/ungemountete SD-Karte definiert werden: *RemovableMedia.Insert (InsertEventHandler)* und *RemovableMedia.Eject (EjectEventHandler)*. Die Namen können leicht zur Verwechslung führen - die Handler werden nicht nach dem Einlegen oder Auswerfen einer SD-Karte, sondern nach dem **Mounten oder Unmounten des Dateisystems** darauf aufgerufen. Derzeit sind diese Handler nicht in Verwendung und wurden daher im Konstruktor auskommentiert.

### 4.1.3. gewünschtes Speichermedium auswählen

Um dem **UsbSdController** mitzuteilen, welches Massenspeichermedium (USB / SD / NAND) verwendet werden soll, kann mit dem Property *selectedPersistentStorageText* erledigt werden. Diesem wird ein String übergeben, welcher dann in der entsprechenden EWR Variable gespeichert wird. Dazu wird der Wert als Target an die EWR übergeben und *PushBackIntoRecoverList* aufgerufen, damit der Wert gleich in den Flash geschrieben

wird und bei einem Neustart wieder zur Verfügung steht. Genauso kann das ausgewählte Speichermedium über dieses Property gelesen werden. Die Methode *setSelectedPersistentStorage* wird zusätzlich aufgerufen, diese erledigt die Überprüfung, ob das gewählte Speichermedium gültig ist. Für jedes Speichermedium steht auch eine eigene Membervariable zur Verfügung, um Manipulationen, wie *mount* oder *unmount* auf das Medium ausführen zu können.

Sollte ein Speichermedium gewählt worden sein, dessen Medium nicht (mehr) verfügbar ist, wird dies in der Methode *setSelectedPersistentStorage* behandelt. Es wird dann automatisch zum internen Speicher (NAND) gewechselt. Im Anschluss muss unbedingt die Methode *persistentStorageDeviceChanged* aufgerufen werden, um dem **DataStorageController** mitzuteilen, dass sich das Speichermedium geändert haben könnte. Ob es letztendlich geändert wurde oder nicht ist nicht von Bedeutung, der **DataStorageController** behandelt dies beim Aufruf der Methode.

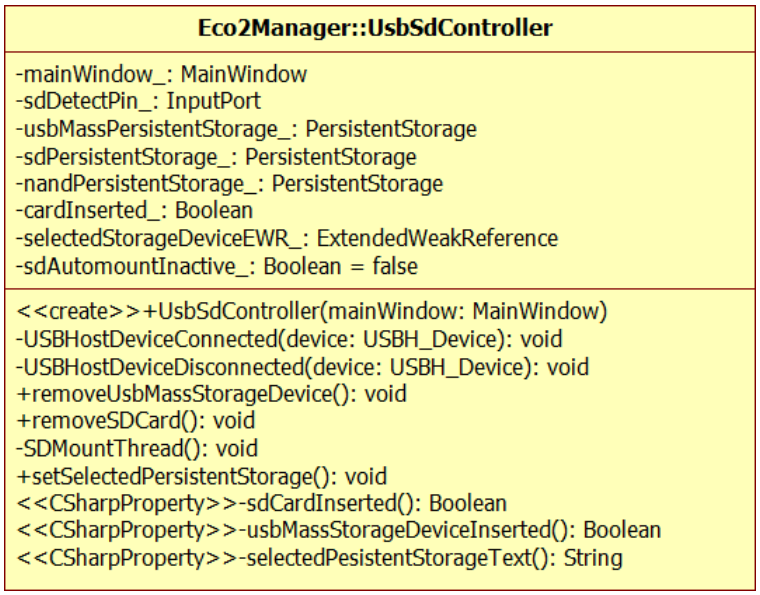



Abbildung 4.1.: Klassendiagramm des UsbSdControllers

## 4.2. DataStorageController

Der **DataStorageController** (Klassendiagramm siehe Abbildung 4.4) ist verantwortlich für die **Speicherung der Daten in SQLite Datenbanken**. Im Konstruktor wird eine Extended Weak Reference (EWR) für das Speicherintervall (*savingIntervalEWR_*) erstellt, sollte bereits ein Wert vorhanden sein wird dieser geladen, ansonsten wird standardmäßig das kleinste Speicherinterval aus den Constants (*MINIMUM_SAVING_INTERVAL*) angenommen.

### 4.2.1. SQLite

Folgende Ziele werden von SQLite verfolgt: [sql12]:

- **Self-Contained:** Soll keine (oder wenige) externe Libraries und kein Betriebssystem benötigen.
- **Serverless**: SQLite soll ohne Server auskommen, um eine einfache Anwendung zu garantieren.
- **Zero-Configuration:** Es soll keine Konfiguration benötigt werden, was durch das Einsparen eines Servers erreicht wird.
- **Transactional:** Einheitliche, konsistente, beständige Änderungen und Queries. Zum Beispiel sollen bei einem Abbruch eines Queries (Systemstemabsturz oder ähnliches) die Daten erhalten bleiben und die Datenbank im Anschluss nutzbar bleiben.

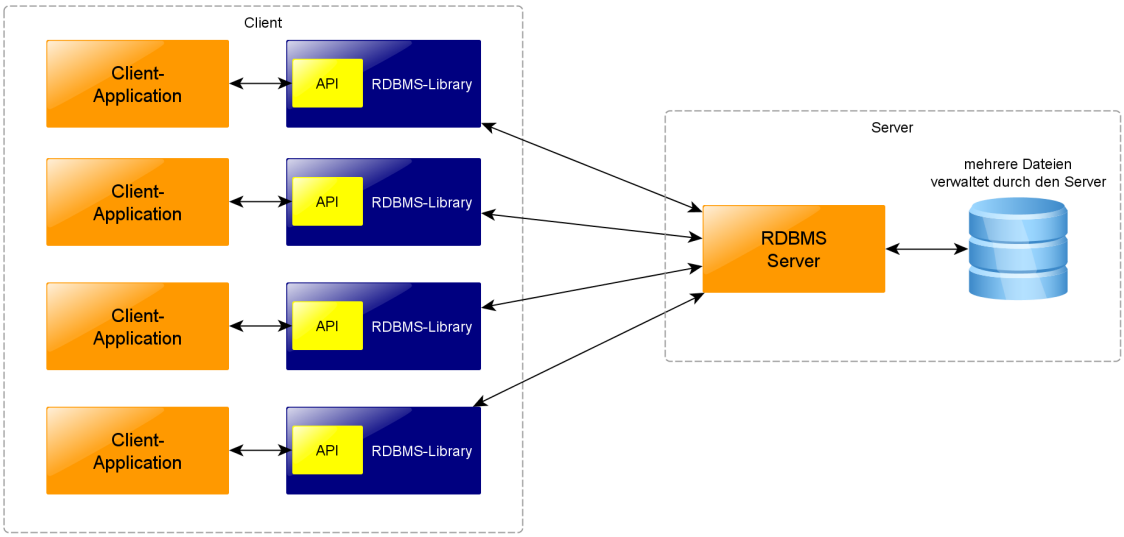



Abbildung 4.2.: traditionelle Struktur für Datenbanken

Bei *SQLite* handelt es sich um ein freies *Relational Database Management System (RDBMS)*, welches es, wie alle anderen dieser Systeme (MYSQL, DB2, Microsoft SQL...), ermöglicht, große Mengen von Daten in Datenbanktabellen abzulegen, wobei die genaue Verwaltung durch das System selbst übernommen wird. Der Benutzer bzw. Programmierer muss sich nicht um die komplette Verwaltung der Daten kümmern. Das *Lite* im Namen bezieht sich nicht auf den Funktionsumfang, sondern auf ein einfaches Setup, wenig administrativen Aufwand und geringe Ressourcennutzung. Im Gegensatz zu anderen RDBMS hat SQLite keine Client/Server-Architektur - es wird also kein eigener Serverdienst benötigt, welcher eine Schnittstelle zur Verfügung stellt. Solche Systeme haben auch meist eine eigene Struktur, eventuell mehrere Dienste und legen die Daten in mehreren Dateien ab, welche für einen Zugriff auf die Datenbank alle vorhanden sein müssen. Um auf die Datenbank zuzugreifen werden Libraries verwendet, welche ein Application

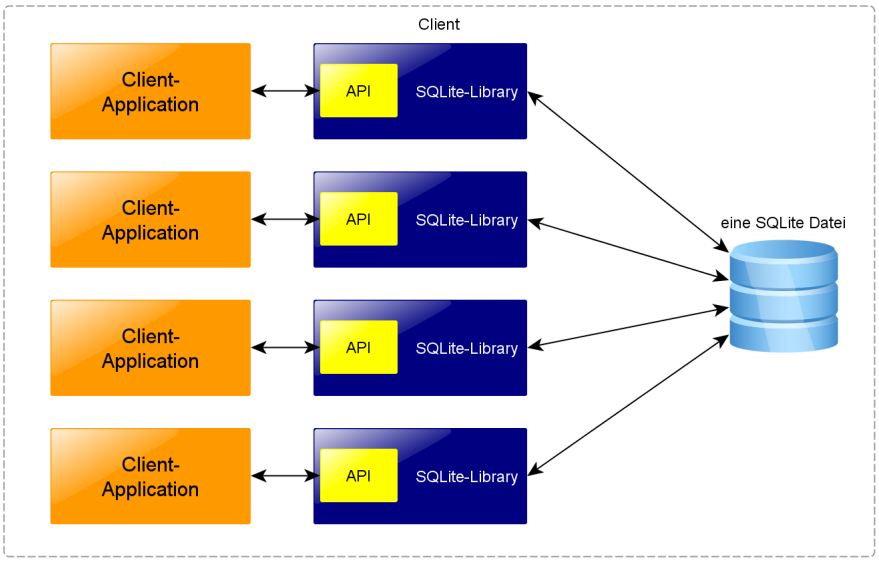



Abbildung 4.3.: Struktur für SQLite-Datenbanken

Programming Interface (API) zur Verfügung stellen und zum Beispiel über das Netzwerk mit dem Datenbank-Server bzw. Datenbank-Service kommunizieren (siehe Abbildung 4.2). Im Gegensatz dazu benötigt SQLite keinen Server. Die komplette Funktionalität für den Zugriff auf die Datenbank ist in der Library enthalten. Es werden keine externen Komponenten mehr benötigt, die Datenbank wird in einer einzelnen Datei abgelegt (siehe Abbildung 4.3). Da kein Server mehr benötigt wird, kann viel Komplexität eingespart werden und das RDBMS kann auch ohne Betriebssytem zum Beispiel auf einem Embedded System ausgeführt werden, da auch kein Multitasking oder hohe Performance benötigt werden. [Kre10]

### SQLite und .NETMF

Von *GHI Electronics* wird eine Library für *SQLite* auf ChipworkX-Modulen zur Verfügung gestellt. Von dieser Library (*GHIElectronics.NETMF.SQLite* Namespace) werden simple Methoden zur Verfügung gestellt, welche es ermöglichen, SQLite-Files zu öffnen, zu schließen und Queries darauf auszuführen. Eine Einschränkung besteht jedoch - die möglichen Datentypen. Es werden nur *Integer*-, *Double*- und *Text*-Werte unterstützt, was aber für dieses Projekt ausreichend ist. Zeit- und Datumsangaben zum Beispiel können durch einfache Integer-Repräsentation gespeichert werden. Ansonsten werden keine direkten Einschränkungen in der Dokumentation der GHI-Library angegeben, daher wird angenommen, dass der SQLite-Standard implementiert wurde. Die Queries am ChipworkX-System werden nach Möglichkeit sehr einfach gehalten, um wenig Overhead zu erzeugen. Die Queries entsprechen Standardqueries, wie diese auch in anderen Structured Query Languages (SQLs) verwendet werden, die Reihenfolge der Schlüsselwörter muss allerdings genau eingehalten werden und kann [Kre10] entnommen werden. In diesem Buch werden die Zugriffe auf SQLite-Datenbanken genau beschrieben und sämtliche Queries genau erklärt und dargestellt.

### 4.2.2. Threadsicherheit SQLite im .NETMF

Die Library von .NETMF wurde von GHI Electronics wirklich sehr einfach gehalten und beinhaltet die wichtigsten Funktionen für den Zugriff auf eine SQLite-Datenbank. Da die Datenbanken hier in Threads verwendet werden, wurden, mangels Dokumentation dieser Umstände, einige Tests bezüglich Threadsicherheit durchgeführt, wobei festgestellt wurde, dass parallele Zugriffe zu teilweise falsch gelesenen Daten und teilweise zur Zerstörung der Datenbankdateien führen. Wurde aus zwei Thread parallel auf die gleiche Datei zugegriffen führte dies meist zur Unlesbarkeit der Datenbankdateien, daher ist dies unbedingt zu verhindern.

Aus diesen Gründen wurde die Klasse **ThreadingDatabase** eingeführt, welche einige Methoden von der Klasse **Database** ableitet und diese threadsicher macht. Die Synchronisierung erfolgt mit Hilfe eines Locks, welcher im erstellten Objekt gehalten wird. Bei einem Zugriff auf dieses Objekt wird immer zuerst der Lock-Status überprüft, und es darf nur jeweils ein Thread auf die Datenbank zugreifen. Folgende Methoden wurden in der Ableitung ersetzt:

- **ExecuteNonQuery:** Führt ein Query aus, welches keine Daten zurück liefert. Es wird die übergeordnete, gleichnamige Funktion aufgerufen, jedoch wird dies nach einem Lock ausgeführt, wodurch die Abfrage sicher für Threading wird.
- **ExecuteQuery:** Führt ein Query aus, welches Daten in Form einer **SQLiteDataTable** zurück liefert. Die Funktionsweise der Sicherheit für Threads funktioniert hier gleich. Der Lock beachtet zusätzlich, dass sich die beiden Methoden nicht untereinander beim Zugriff stören können.

### 4.2.3. Erstellen der Datenbanken

Wie zu Beginn dieses Kapitels bereits angesprochen, werden zwei Datenbanken für zwei verschiedene Datentypen verwendet:

Eine Datenbank im internen NAND-Speicher, welche Informationen über Sensoren und Aktoren speichert. Zum Erstellen dieser Datenbank wird die Methode *openSettingsDatabase* zur Verfügung gestellt. Wird diese Methode aufgerufen, wird eine Datenbankdatei im NAND-Speicher angelegt, deren Name der Variable *DATABASE_FILENAME_SENSORS_ACTORS* in der Klasse **Constants** definiert ist. Sollte die Datei noch nicht vorhanden sein, wird sie durch den *Open*-Befehl automatisch angelegt und geöffnet.
Im Anschluss werden die beiden Tabellen für die Sensoren und Aktoren erstellt, sollten diese bereits vorhanden sein, wird dieses Statement wegen dem Zusatz *IF NOT EXISTS* ignoriert. Die genaue Datenbankstruktur kann Unterabschnitt 4.2.4 entnommen werden.

Die folgende Statements werden zum Erstellen der Datenbanken ausgeführt:

- ```
  CREATE TABLE IF NOT EXISTS
  actors
  (sensorid TEXT, actorid TEXT,
  PRIMARY KEY (sensorid, actorid))
  ```
  ```

- `CREATE TABLE IF NOT EXISTS`
  `sensors`
  `(id INTEGER PRIMARY KEY AUTOINCREMENT, sensorid TEXT, valueid TEXT,`
  `co2factor DOUBLE, co2offset DOUBLE, absoluteValue INTEGER)`

Die zweite Datenbankdatei wird auf dem ausgewählten Speichermedium (siehe Unterabschnitt 3.4.5) angelegt. Dies geschieht jedes Mal, wenn das Massenspeichermedium geändert wurde und beim Start des Systems. Dazu wird die Methode *persistentStorageDeviceChanged* aufgerufen, welche kontrolliert, ob sich das Speichermedium geändert hat (wenn nicht ist kein erneutes Erstellen notwendig) und versucht die Tabelle für die Sensorwerte anzulegen (Struktur siehe Unterabschnitt 4.2.4). Sollte die Tabelle bereits vorhanden sein wird das `CREATE`-Statement ignoriert.

Das Statement zum Erstellen sieht folgendermaßen aus:

- `CREATE TABLE IF NOT EXISTS`
  `sensorvalues`
  `(id INTEGER, value DOUBLE, absoluteValue DOUBLE, timestamp INTEGER,`
  `PRIMARY KEY(id, timestamp))`

### 4.2.4. Datenbankstrukturen

In diesem Abschnitt wird die Struktur der Datenbanken erklärt und die Verwendung der entsprechenden Felder wird kurz beschrieben. Zu beachten ist, dass *Primary-Keys* (PK) unterstrichen dargestellt werden und keine *Foreign-Keys* (FK) verwendet werden, da dies über zwei Datenbankdateien hinweg nicht möglich ist. Primary Keys werden verwendet, damit die Datenbank einen Index erstellen kann und damit die Suche beschleunigt wird. Des Weiteren müssen die Spalten, welche als Primary Key definiert wurden, in Kombination einheitlich sein, dürfen also nicht doppelt vorkommen.

#### Tabelle für Aktoren

Diese Tabelle verwaltet die Aktoren, welche im System vorhanden sind. Diese müssen vom Benutzer einmal definiert werden. Die Struktur kann Tabelle 4.1 entnommen werden, die einzelnen Werte werden hier kurz erklärt:

- **sensorid:** Die ID des Sensorboards, auf dem sich der Aktor befindet. Hier wird keine numerische ID angegeben, da die Einträge in keiner anderen Datenbanktabelle relevant sind und die numerische ID daher nur Platz verschwenden würde.
- **actorid:** Die ID, über die der entsprechende Aktor angesprochen werden kann. Bei XBee Anbindung (siehe [Mad11]) kann dies zum Beispiel ein I/O Port sein. Bei Netzwerkanbindung kann diese ID frei gewählt werden, sofern diese auch dem Controller am Sensorboard bekannt ist. Diese beiden IDs werden in Kombination zum Ansteuern der Aktoren im **ActorWindow** angezeigt, wie in Unterabschnitt 3.4.3 beschrieben.

| TABLE: actors |
|---|
| sensorid TEXT |
| actorid TEXT |

Tabelle 4.1.: Datenbankstruktur für Aktoren

### Tabelle für Sensoren

Über diese Tabelle werden die Sensoren verwaltet, die an den $Eco_2$-Manager angeschlossen wurden.

| TABLE: sensors |
|---|
| id INTEGER AUTOINCREMENT |
| sensorid TEXT |
| valueid TEXT |
| co2factor DOUBLE |
| co2offset DOUBLE |
| absoluteValue INTEGER |

Tabelle 4.2.: Datenbankstruktur für Sensoren

- **id:** Eine eindeutige, numerische ID für diesen Sensor. Diese wird beim Speichern der Sensordaten verwendet, um nicht jedes Mal zwei Strings speichern zu müssen. Die ID wird, wenn nicht angegeben, automatisch auf die nächste höhere ganze Zahl gesetzt.
- **sensorid:** Die ID eines Sensorboards bzw. die Seriennummer eines Smart Meters. Die Seriennummer muss **genau acht Stellen** haben, IDs von Sensorboards können frei gewählt werden, sollten aber keine ganzzahligen Werte mit acht Stellen sein, um Verwechslungen mit Smart Metern zu verhindern.
- **valueid:** Die ID des entsprechenden Sensorwertes. Auf einem Sensorboard können mehrere Sensoren sitzen, daher wird eine weitere Unterscheidung durch diese ID angeboten. Wird eine unbekannte Value-ID eines Sensors oder ein neuer Sensor gefunden, wird dies in die Datenbank mit Standardwerten eingetragen. Der Offset wird dabei auf 0 und der Faktor auf 1 gesetzt. Für Smart Meter wird hier der OBIS-Code eingetragen (siehe Abschnitt 7.3).
- **co2factor:** Der Faktor, mit dem der aufgezeichnete Wert des Sensors multipliziert wird, damit man den $CO_2$-Ausstoß in $kgCO_2$ erhält.
- **co2offset:** Der Wert, der zum bereits mit dem Faktor multiplizierten Wert addiert wird, um den $CO_2$-Ausstoß in $kgCO_2$ zu erhalten.
- **absoluteValue:** Gibt an, ob es sich bei den Werten vom Sensor um einen Absolutwert (1) oder Relativwert (0) handelt. Diese Spalte kann als Boolean betrachtet werden, dieser Datentyp ist aber in SQLite nicht vorhanden, daher wird ein Integer verwendet.

Folgende Berechnung wird durchgeführt, um den gemessenen Wert in $kgCO_2$ umzurechnen: $kgCO_2 = (gemessenerWert \cdot co2factor) + co2offset$

#### Tabelle für gemessene Sensorwerte

Diese Tabelle beinhaltet die gemessenen Sensorwerte, wobei für jeden Wert ein Timestamp mitgespeichert wird. Da es in der SQLite-Library von GHI-Electronics keine *DATETIME*- oder *TIMESTAMP-Datentypen* gibt, wird der entsprechende Unix-Timestamp selbst berechnet (siehe Unterabschnitt 4.2.5) und als Integer gespeichert.

Der Wert selbst wird als *value* gespeichert, in der Spalte *absoluteValue* wird der absolute Wert eines Sensors gespeichert, bei relativen Werten ist dieser 0.

| TABLE: sensorvalues |
|---|
| id INTEGER |
| value DOUBLE |
| absoluteValue DOUBLE |
| timestamp INTEGER |

Tabelle 4.3.: Datenbankstruktur für gemessene Sensorwerte

### 4.2.5. Hilfsfunktionen

In der Klasse **DataStorageController** wurden noch einige weitere Funktionen definiert, welche hier in Folge erklärt werden. Eine Übersicht kann Abbildung 4.4 entnommen werden.

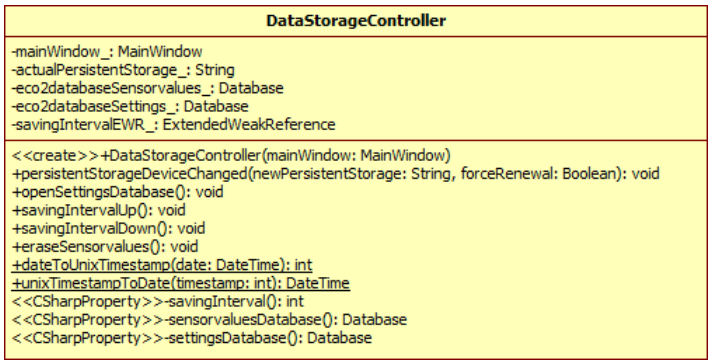



Abbildung 4.4.: Klassendiagramm des DataStorageController

#### Speicherintervall

Die beiden Funktionen *savingIntervalUp* und *savingIntervalDown* werden verwendet, um das Speicherinterval, in dem die Werte in die Datenbank geschrieben werden, zu verändern. Dabei wird darauf geachtet, dass das Minimum (*Constants.MINIMUM_SAVING_INTERVAL*) nicht unterschritten und das Maximum (*Constants.MAXIMUM_SAVING_INTERVAL*)

nicht überschritten wird. Wurde einer dieser Werte erreicht ist kein Increment bzw. Decrement der Werte mehr möglich.

#### Datenbankwartung

Um zu alte Sensorwerte (eine Woche) aus der Datenbank im **internen Speicher** zu löschen und den Speicherplatz für andere Zwecke zur Verfügung zu stellen, wird die Funktion *doDatabaseMaintaince* verwendet. Die Funktion wird jede Stunde aufgerufen um die alten Werte zu löschen, zusätzlich wird die **Datenbank mit den Sensoreinstellungen** auf die SD-Karte oder den USB-Stick kopiert, damit diese für eine Auswertung am PC zur Verfügung steht. Die Daten der Sensoreinstellungen etc. bleiben dabei vollständig erhalten. Um sicher zu stellen, dass diese Datenbank sich beim Auswerfen auf dem Wechselmedium befindet, wird die Funktion auch beim Entfernen eines Wechselmediums aufgerufen. Dazu muss allerdings der Benutzer das Medium vorschriftsmäßig auswerfen.

#### Timestamp-Funktionen

Da es, wie bereits erwähnt, in der SQLite-Library von GHI Electronics keinen Datentyp für Zeitstempel gibt, wurden zwei Funktionen geschrieben, welche eine Umwandlung von einem *DateTime-Objekt* in einen Unix-Timestamp (die Anzahl der Sekunden seit dem 1.1.1970, 0 Uhr) und umgekehrt ermöglichen. Dieser Timestamp kann als Integer in der Datenbank abgelegt werden und kann in den meisten Programmiersprachen sehr einfach verwendet werden.

- *dateToUnixTimestamp:* Wandelt ein *DateTime-Objekt* in einen Timestamp. Dabei wird die Zeitspanne zwischen dem Referenzdatum 1.1.1970 und dem übergebenen Datum berechnet (in Sekunden).
- *unixTimestampToDate:* Wandelt einen Timestamp in ein *DateTime-Objekt*, indem zum Referenzdatum die übergebene Anzahl der Sekunden dazu addiert wird.

## 4.3. Eco2IO

Im Namespace **Eco2IO** wurden einige Klassen geändert. Prinzipiell handelt es sich hierbei um das Masterprojekt [Mad11], dieses wurde aber erweitert und angepasst.

### 4.3.1. Klasse Eco2IO

Die in der Projektarbeit [Mad11] vorgesehene Routine zum Verarbeiten der Daten, die über den seriellen Port über ein XBee-Pro-Funkmodul empfangen werden, wurde abgeändert. Da das Escaping der empfangenen Zeichen zuerst im Event-Handler des seriellen Ports erledigt wurde (siehe [Mad11, Seite 71]) kam es zu Problemen beim Ausführen der grafischen Oberfläche, da ein Event-Handler eine sehr hohe Priorität besitzt und damit die Ausführung des Main-Threads unterbricht und eine Interaktion am GUI unterbricht. Um dies zu verhindern, werden die Daten im Event nur in eine Queue geschrieben und durch einen eigenen Thread verarbeitet.

Der Thread *ProcessReceivedDataThread* wurde bereits in der Projektarbeit für die Verarbeitung der empfangenen Daten geschaffen. Dieser wird regelmäßig durchlaufen und verarbeitet alle Daten, die in dieser Zeit empfangen wurden. Wurden keine Daten empfangen, wird der Thread pausiert, bis neue Daten an den $Eco_2$-Manager gesendet wurden. Näheres dazu siehe Abschnitt 5.2.

Sobald der Thread gestartet wurde, wird für alle empfangenen Daten die selbe Prozedur abgearbeitet:

- Zu Beginn wird in der Hashtable *valueMemory_* nachgesehen, ob der aktuelle Sensor mit seiner ID (`sensorID/sensorSubID`) bereits vorhanden ist. Ist dies nicht der Fall, wird ein neuer Eintrag mit der vollen ID für den Sensor und einem Objekt der Klasse **SensorValue** (siehe Unterabschnitt 4.3.2) angelegt. Dem Konstruktor werden die nötigen Informationen über den Sensor mitgegeben, um zu verhindern, dass bei jedem Durchlauf für jeden Sensor eine Abfrage in der Datenbank stattfinden muss, indem der letzte gespeicherte Zeitstempel, der letzte gespeicherte Wert und die Information, ob es sich um einen absoluten Wert handelt, in einer Variable gehalten wird. Durch eine Abfrage bei jedem Durchlauf würde der Prozess wesentlich langsamer, da dies mit sehr viel Overhead verbunden wäre. Die Tabelle *valueMemory_* wird absichtlich erst beim Abarbeiten der Daten aufgebaut, um nicht aktive Sensoren auch nicht in der Tabelle zu halten.
- Im Anschluss wird der gerade empfangene Wert dem **SensorValue**-Objekt hinzugefügt, wobei der Wert nur mittels *addValue*-Methode hinzugefügt werden muss, der Rest wird von der Klasse selbst erledigt.
- Nach einer Abfrage, ob der letzte gespeicherte Wert bereits so alt ist, dass eine neue Speicherung erforderlich ist, wird entweder der relative Wert direkt, oder der absolute Wert abzüglich des letzten gespeicherten Absolutwertes (ergibt dann ebenfalls einen relativen Wert) in die Datenbank geschrieben. Die Berechnung des relativen Wertes erfolgt hier, da es dann egal ist, wenn ein Absolutwert in den Sensoreigenschaften auf einen Relativwert umgestellt wird. Zudem erleichtert dies die Auswertung der Daten, da nur mehr Relativwerte verarbeitet werden müssen.

### 4.3.2. Klasse SensorValue

Diese Klasse dient als Speicher für die vorher in der Datenbank gespeicherten Werte und zur Berechnung des nächsten zu speichernden Wertes. Sie wurde eingeführt, um das Speichern von mehr Information in der Hashtable der Klasse **Eco2IO** (siehe Unterabschnitt 4.3.1) zu ermöglichen und die Funktionalität des Berechnens der Werte in einer eigenen Klasse zu kapseln.

Der *Konstruktor* verlangt folgende Informationen:

- **isAbsoluteValue:** Wenn es sich um einen absoluten Wert handelt, muss hier *true* eingetragen werden, bei einem relativen Wert *false*.
- **lastSavedTimestamp:** Der letzte in der Datenbank vorhandene Zeitstempel für diesen Sensor.
- **lastSavedAbsoluteValue:** Der letzte in der Datenbank vorhandene Absolutwert.

Mit der Methode *addValue* können dann beliebig viele Werte hinzugefügt werden, welche mit *getValue* wieder abgefragt werden können, wobei diese Abfrage bereits beachtet, ob es sich um einen Absolutwert oder einen Relativwert handelt. Ebenso wird bei einem Relativwert die Summe der Werte und die Anzahl der Werte wieder auf Null gesetzt. Bei einem Absolutwert bleibt der letzte Wert immer vorhanden, welcher auch mit der Methode *getAbsoluteValue* abgefragt werden kann.

Ob eine erneute Speicherung nötig ist, kann mit der Methode *TimeSpanSinceLastSave* und einem Vergleich mit der eingestellten Zeitspanne für das Speicherintervall (siehe Abschnitt 4.2) festgestellt werden.

Wurde eine Speicherung in die Datenbank durchgeführt muss zwingend *setLastSaved* aufgerufen werden, damit die Werte in der Klasse aktualisiert werden und das neue Speicherinterval berechnet werden kann.

# Kapitel 5.

# Änderungen in der Sensoranbindung

## 5.1. Allgemeine Anpassungen

### Objekte statt statischen Klassen

In [Mad11, Kapitel 3.2.1] wurde beschrieben, dass statische Variablen und Methoden verwendet wurden, um eine Instanzierung und das Halten der Instanzen innerhalb der Kommunikationsklassen zu vermeiden. Es stellte sich im Laufe der weiteren Entwicklung heraus, dass die Einschränkungen durch statische Methoden und Klassen doch groß sind und der Zugriff dadurch eher erschwert wird. Aus diesem Grund wurden sämtliche Klassen der Kommunikation umgestellt, diese müssen nun instanziert werden. Die Instanzen werden alle in der Klasse **Eco2IO** gehalten und erzeugt, von dort aus kann auf die einzelnen Module wie **Ethernet**, **XBee**, **XBeeNodeTable**, **XBeeFrametable** etc. zugegriffen werden. Eco2IO wird vom **MainWindow** aus instanziert, dies kann auch Abbildung 5.1 entnommen werden.

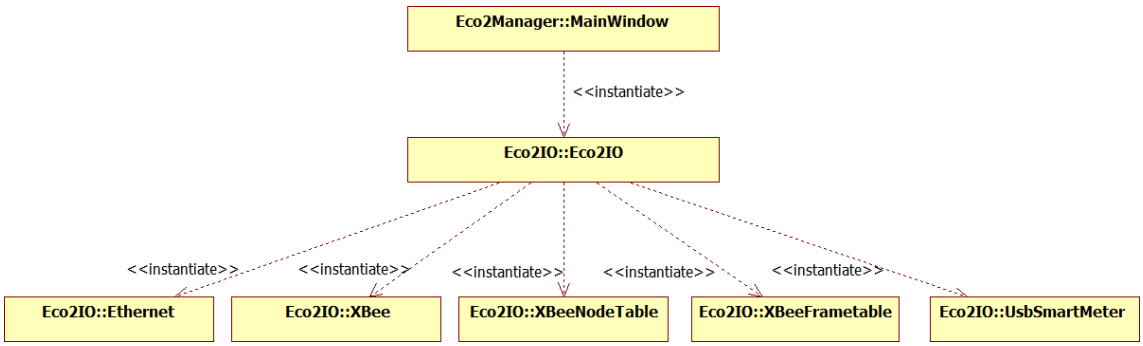



Abbildung 5.1.: Instanzierungen der Kommunikationsklassen

### Debugging

Das Debugging wurde ebenfalls umgestellt auf die Klasse **DebugHelper**, hier muss diese allerdings zusätzlich mit dem Namespace **Eco$_2$-Manager** angesprochen werden. Es können alle drei Stufen des Debuggings (info/warn/error) verwendet werden. Der Vorteil dieser Methode liegt im einheitlichen Output mit Klassenname und Priorität.

#### Hinzufügen von USB Kommunikation für Smart Meter

Es wurde eine zusätzliche Anbindung für Smart Meter mittels USB hinzugefügt. Dies wurde speziell für das vorhandene **Luna Smart Meter LUN4** entwickelt, genaueres zu dieser Anbindung kann Abschnitt 7.3 entnommen werden.

## 5.2. XBee Anpassungen

#### Korrektur des ReceiveDataHandlers

Das Einlesen der Daten, Escaping und Zerlegen in einzelne API-Pakete im *ReceiveDataHandler*, wie es vorgesehen war, funktioniert nicht in der gewünschten Form. Das Event wird zwar als eigener Thread geführt, allerdings kam es zu Problemen, da der Event-Handler nur sehr kurz sein darf und die, eigentlich einfache, Aufteilung in die API-Pakete zu lang war. Es wurden Events nicht mehr abgearbeitet oder der Eventhandler aufgerufen, obwohl keine Daten vorhanden waren. Dies führt zwar zu keinem Fehlverhalten beim Empfang, allerdings zu unnötigen Overhead am Prozessor, welcher vermieden werden sollte.

Da diese Probleme auch durch Optimierung des Codes im Handler nicht vermieden werden konnten, wurde die Strategie für den Empfang der Daten verändert. Nach einigen Nachfragen bei GHI Electronics wurde empfohlen, alle verfügbaren Daten vom seriellen Port **auf einmal** zu lesen. Durch die Pufferung des Ports kann ein byteweises Lesen zu Problemen führen. Alle verfügbaren Daten werden nun in der Klasse **XBee** zu Beginn des *ReceiveDataHandler* in einem Block eingelesen und in eine Warteschlange (*receivedSerialData_*) eingefügt. Wenn Daten gelesen wurden, wird noch das *_receiveSerialDataEvent* gesetzt, damit der *ProcessSerialDataThread* weiter ausgeführt wird und die Daten aus der Warteschlange abarbeitet. Das Escaping bzw. Verarbeiten der Daten wird nun von diesem Thread übernommen.

Die Funktionalität dieses Threads gleicht der Funktion des vorherigen *ReceiveDataHandlers*, mit der Erweiterung, dass der Thread pausiert wird, wenn keine Daten mehr vorhanden sind und es musste eine Synchronisation für die zusätzlich vorhandene Warteschlange angelegt werden. Dafür wurde der *_receive_serial_data_lock* eingeführt, welcher bei jedem Zugriff auf die Queue geholt und anschließend wieder freigegeben wird. Anfängliche Bedenken, dass dies Probleme mit dem Event geben könnte bestätigten sich nicht, da Events im System eine wesentlich höhere Priorität besitzen als Threads und der Lock beim Pausieren eines Threads freigegeben wird. Um den Ablauf zu verdeutlichen, wurde die Abbildung 4.12 aus [Mad11, Seite 83] abgeändert, auf den neuen Ablauf (siehe Abbildung 5.2). Alle neuen Elemente wurden orange eingefärbt, die bläulich gefärbten Abläufe wurden übernommen. Es wurde hier nur der erste Teil abgeändert, daher wird auch nur dieser dargestellt.

Im Prinzip wurde ein zusätzlicher Puffer eingefügt, damit die Daten nicht direkt vom seriellen Port byteweise gelesen werden und der Receive-Handler des Ports entlastet wird. Die Abarbeitung der Events und die korrekten Aufrufe des Event-Handlers konnten so sichergestellt werden.


Abbildung 5.2.: geänderter Datenempfang XBee

### Powerlevel

Die XBee-Digimesh-Module wurden bisher im höchsten Powermode betrieben. Dies führte ohne grafische Oberfläche zu keinen Problemen. Sobald eine grafische Oberfläche verwendet wird, gibt es bei reiner USB-Versorgung des ChipworkX-Development-Systems Probleme

bei der Versorgung. Das System wird am PC zum Teil nicht mehr richtig erkannt, es ist kein Debugging mehr möglich bzw. das Display flackert.

Als Lösung kann eine externe, besser stabilisierte Stromversorgung verwendet werden, allerdings hat die derzeitige Firmware des ChipworkX-Development-Boards Probleme mit USB-Debugging, wenn dieses extern versorgt wird. Als Lösung für die Entwicklung wurde der Powerlevel des am Development-System auf 1 reduziert, die *PWM-RSSI-Indicator LED* und die *Status Indicator LED* deaktiviert. Dadurch war ein problemloses Debugging möglich.

## 5.3. Anpassungen Ethernet

Die Klasse **Ethernet** ist nicht mehr nur für die Kommunikation mit den Sensoren zuständig, sondern zusätzlich für das Senden und Empfangen von Werten zum und vom Client. Folgende Threads sind dafür verantwortlich:

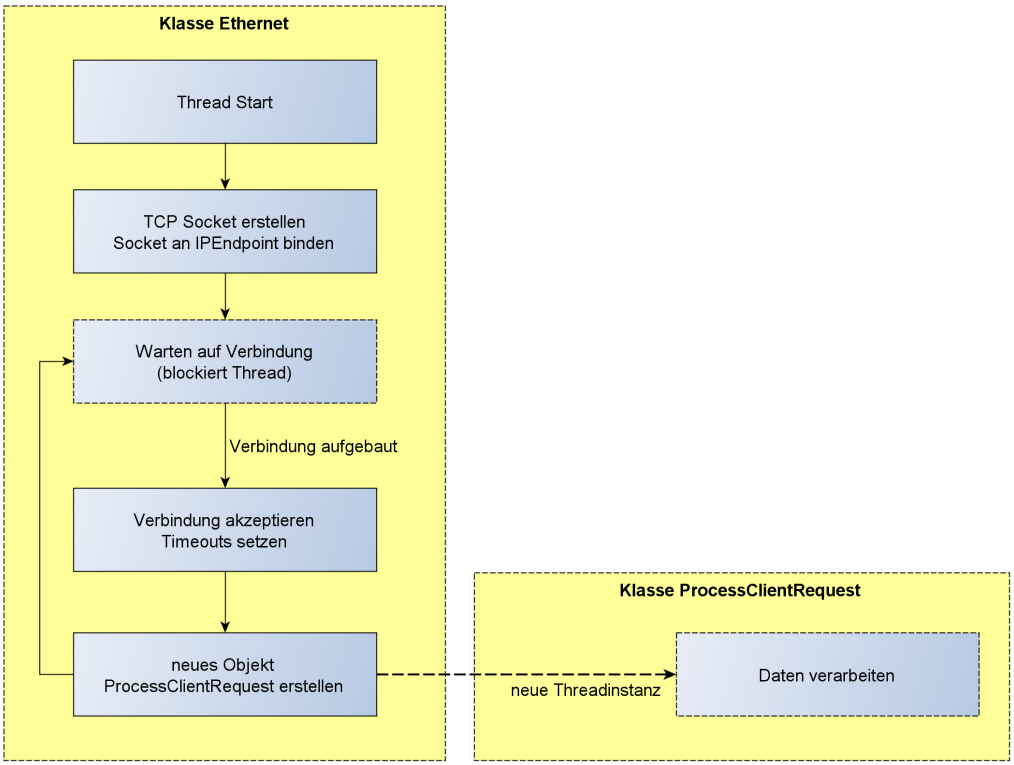



Abbildung 5.3.: Ablauf Datenempfang $Eco_2$-Client

- **EthernetTcpListeningSensorsThread:** Dieser Thread hört auf den in *IOConstants.TCP_LISTENING_PORT_SENSOR* vorgegebenen Port und empfängt auf diesem Daten von den Sensoren, wie bereits in [Mad11] beschrieben.
- **EthernetTcpListeningClientThread:** Um die Daten an den $Eco_2$-Client zu senden wurde dieser Thread neu hinzugefügt. Dieser hört am Port, der in der Konstante

*IOConstants.TCP_LISTENING_PORT_CLIENT* definiert ist und legt für jede neue Verbindung ein **ProcessClientRequest** Objekt an, welches die Daten verarbeitet (asynchron). Die Klasse **ProcessClientRequest** wird in Abschnitt 5.4 genauer beschrieben, veranschaulicht wird dies in Abbildung 5.3.

- **EthernetUdpBroadcastThread:** Wie bereits in [Mad11] beschrieben, sendet dieser Thread UDP-Broadcasts aus, mit einer Nachricht, mit der man diese dem $Eco_2$-Manager zuordnen kann und somit dessen IP-Adresse ermitteln kann.

## 5.4. ProcessClientRequest

Zum Verarbeiten der Daten des $Eco_2$-Client mussten einige neue Klassen eingeführt werden. Die wichtigste Klasse dabei ist sicherlich **ProcessClientRequest**, welche die Anfragen abarbeitet, diese wird in Folge beschrieben. Aber auch die Klassen **Serializeable, SerialProcessingInstruction, SerialSensor, SerialSensorData, SerialActor** und **SerializationHelper** spielen hier eine wichtige Rolle, diese werden in Unterabschnitt 5.4.3 genauer erklärt.

### 5.4.1. Grundprinzip

Der Thread *EthernetTcpListeningClientThread* aus der Klasse **Ethernet** erstellt, sobald am gewählten Port Daten empfangen wurden, ein neues Objekt der Klasse **ProcessClientRequest**. Im Konstruktor werden die Membervariablen der Klasse gesetzt (Verweis auf **Socket** und **Eco2IO**) und es wird ein neuer Thread *ProcessRequestThread* erzeugt, um den aufrufenden Thread wieder verfügbar zu machen. Der aufrufende Thread kann fortgesetzt werden, dieser wird hier wieder für einen neuen Datenempfang konfiguriert.

Im **ProcessRequestThread** werden die Daten analysiert (mit Hilfe der Klasse **SerialProcessingInstruction** - siehe Unterabschnitt 5.4.3) und dementsprechend weitere Funktionen aufgerufen. Anfangs wurde über die Methoden *send* und *receive* der Sockets versucht den Datentransfer abzuwickeln, dies wird allerdings zu aufwändig und es sind zu viele Sonderfälle zu beachten. Durch verwenden eines *NetworkStreams* wurde dieses Problem behoben. Dieser Stream ermöglicht es, über einen Socket zu senden und zu empfangen, ohne jedes Mal alle Sonderfälle zu überprüfen und abzufangen. Mittels einfacher Funktionen kann darauf geschrieben werden, um den Versand kümmert sich die Klasse **NetworkStream** selbst, allerdings treten auch hierbei einige Probleme auf, wie in Unterabschnitt 5.4.4 beschrieben. Empfangen von Daten funktioniert ebenso einfach, wobei alle empfangenen Daten im Puffer warten, bis sie dort abgeholt werden.

### 5.4.2. Senden und Empfangen von Daten

Um die Daten richtig mit dem $Eco_2$-Client austauschen zu können wurde unter folgenden Möglichkeiten das Protokoll ausgesucht, welches sich für diese Anwendung am besten eignet.

### Übertragung mit XML

- **Vorteile:**
  - für Personen gut lesbar
  - Parser in fast jedem Framework verfügbar (.NETMF, Java...)
  - plattformunabhängig
  - große Verbreitung
  - flexibel
- **Nachteile:**
  - großer Overhead (Tags, Header...)
  - hoher Aufwand zum Parsen, eventuell langsam am Embedded System
  - nicht für Binärdaten geeignet
  - variable Datenlänge (Integer zum Beispiel zwischen 1 und 11 Byte, Double eventuell mehr)

Die Übertragung mit XML war anfangs geplant. Während der Umsetzung wurde klar, dass die Datenübertragung zu langsam ist, da XML zu viel Overhead erzeugt. Zudem ist der XML-Writer und XML-Parser in .NETMF im Vergleich zu anderen Technologien zu langsam, wodurch die Übertragung sehr unperfomant wurde. Die letztendlich gewählte Technologie (binäre Codierung) war bis zu vier mal schneller als die Übertragung mittels XML. Weiterhin verwendet wird diese Technologie für das Übertragen der Sensorwerte, da dort nur sehr geringe Datenmengen verschickt werden und hier das Parsen von XML die einfachste und für Menschen leserlichste Methode darstellt.

### Übertragung mit JavaScript Object Notation (JSON)

- **Vorteile:**
  - für Personen gut lesbar
  - weniger Overhead als XML
  - ressourcenschonender als XML
  - plattformunabhängig
- **Nachteile:**
  - geringere Verbreitung
  - oft keine Parser vorhanden
  - nicht für Binärdaten geeignet
  - keine Validierung möglich
  - variable Datenlänge

Da das Parsen und Erstellen von JSON vom .NET Micro Framework standardmäßig nicht unterstützt wird und der Overhead immer noch relativ groß ist, ist diese Variante nicht in die nähere Auswahl gekommen.

#### Übertragung mit SOAP:

- **Vorteile:**
  - Parser in den meisten Systemen vorhanden (auch .NETMF, Java...)
  - vielseitig
  - sehr gut validierbar
  - schnellere Übertragung durch Optimierungen
  - basiert auf XML
  - plattformunabhängig
  - robust
- **Nachteile:**
  - sehr viel Overhead
  - großer Verarbeitungsaufwand
  - langsam
  - variable Datenlänge

SOAP wird zwar von .NETMF voll unterstützt, erzeugt aber, wie auch bereits XML zu viel Overhead. Hier wurden noch schlechtere Werte als bei XML erreicht und die Versuche der Übertragung mittels SOAP wurden eingestellt.

#### Übertragung mit binärer Codierung

- **Vorteile:**
  - fast kein Overhead
  - sehr schnell durch geringen Rechenaufwand
- **Nachteile:**
  - jeder Datentyp muss eigens codiert werden
  - keine einheitliche Codierung
  - plattformabhängige Implementierung nötig (little/big-endian)
  - vorhandene Bibliotheken bei Datenaustausch zwischen verschiedenen Plattformen oft nicht Nutzbar, da kein Standard vorhanden

Diese Art der Übertragung wurde letztendlich für das Projekt ausgewählt. Es konnte die beste Perfomance erzielt werden, allerdings bedeutet dies auch einigen Aufwand. Es mussten Klassen erstellt werden, welche die Daten serialisieren, also in eine definierte Form bringen, die sowohl per Stream versendet werden kann, als auch am Client wieder decodiert werden kann. Unterabschnitt 5.4.3 beschreibt dies im Detail.

#### Gegenüberstellung der Geschwindigkeit der Übertragung

Zu Testzwecken umgesetzt wurden drei Arten der Übertragung - XML, SOAP und binär. Um die Geschwindigkeit beurteilen zu können, wurde das Senden von 1000 Sensorwerten über Sockets zu einer Beispielanwendung am PC über einen **NetworkStream** realisiert. Am PC wurden die Daten nur eingelesen, aber nicht weiter verarbeitet, gesendet wurden immer die selben Daten. Gemessen wurde die Zeit von Beginn eines neu erstellten Threads bis zum Ende der Übertragung (Ende des Threads). Die benötigte Zeit wird in Abbildung 5.4 gegenübergestellt.

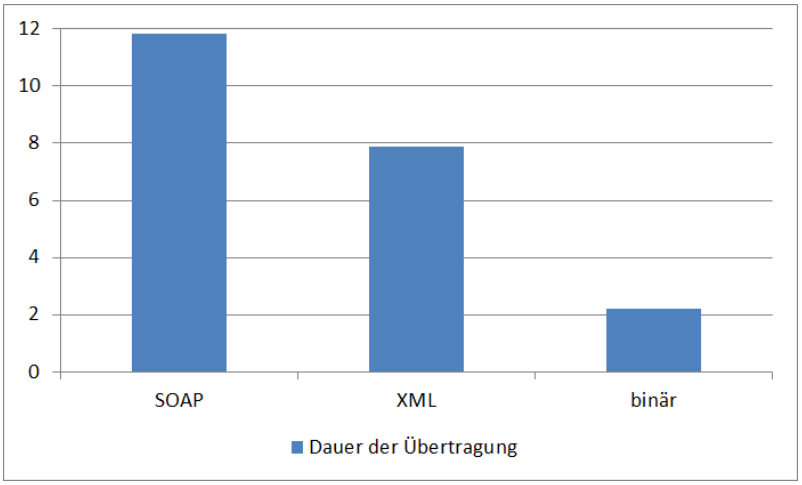



Abbildung 5.4.: Gegenüberstellung der Übertragungsmethoden

In der Grafik ist deutlich ersichtlich, warum auf binäre Codierung umgestellt wurde. Die Geschwindigkeit ist gegenüber XML in etwa **vier mal** und gegenüber SOAP sogar circa **sechs mal schneller**.

### 5.4.3. binäre Codierung

Um die Sensoren, Aktoren, Sensorwerte usw. binär codieren zu können, wurden verschiedene Klassen erstellt, die dies vereinfachen. Die gleichen Klassen müssen auch in Java zum Decodieren bzw. Codieren der zurück gesendeten Werte vorhanden sein.

#### Serializeable

Diese Klasse stellt die wichtigsten Funktionen zum Codieren der Daten dar. Für jeden verwendeten Datentyp steht eine Funktion zum Codieren/Serialisieren und zum Decodieren/Deserialisieren zur Verfügung, die Funktionsnamen geben Auskunft über den Datentyp der damit codiert werden kann. Als Übergabewerte bekommen die Funktionen einen Stream, in den die Daten geschrieben werden, mit dem jeweiligen Wert dazu, bzw. einen Stream von dem gelesen wird und die gelesenen Elemente werden per Rückgabewert ausgegeben. Eine Überladung der Funktionen wurde absichtlich vermieden, um Fehler zu minimieren, alle Funktionen wurden für **Little Endian** Plattformen geschrieben, da bisher keine .NETMF Plattformen mit Big Endian bekannt sind. Übertragen werden die

Werte in **Big Endian**, da dies der Standard von Java ist. Auf das Senden der Datentypen wurde verzichtet, da der PC immer genau das anfordert, was er benötigt und es mit dieser Information decodieren kann.

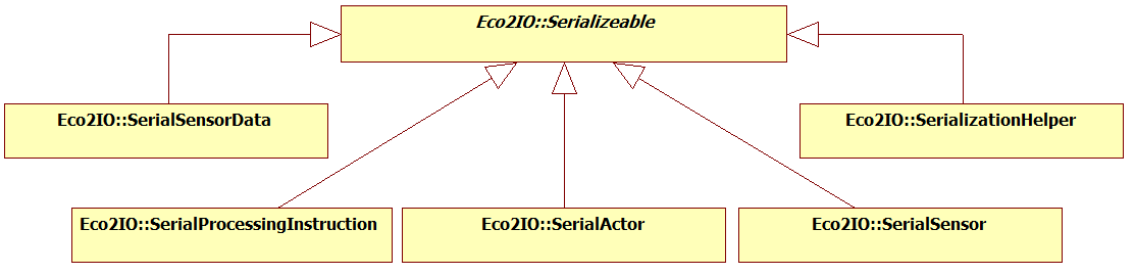



Abbildung 5.5.: Übersicht über Klassen zum Codieren der Daten

Alle weiteren Klassen die mit *Serial* beginnen leiten von dieser Klasse ab und verwenden die Funktionen, die hier definiert wurden. Eine Übersicht über die Klassen kann Abbildung 5.5 entnommen werden.

Das Format, in welches die Datentypen gebracht werden bzw. aus dem sie decodiert werden ist wie folgt festgelegt und in den Funktionen umgesetzt:

- **String:** Für einen String werden zuerst **zwei Bytes für die Länge des Strings** (Big Endian) und anschließend der String übertragen (siehe Tabelle 5.1).

| Länge MSB | Länge LSB | Zeichen des Strings |
|---|---|---|
| **Summe: (2 + Stringlänge) Bytes** | | |

Tabelle 5.1.: Übertragungsschema String

- **Integer:** Ein Integer wird mit vier Bytes in Big Endian übertragen (siehe Tabelle 5.2).

| Byte 3 (MSB) | Byte 2 | Byte 1 | Byte 0 (LSB) |
|---|---|---|---|
| **Summe: 4 Bytes** | | | |

Tabelle 5.2.: Übertragungsschema Integer

- **Double:** Für Double-Werte existiert der Standard IEEE 754 - Standard for Floating-Point Arithmetic [IEE08], wie Floating-Point Werte im Speicher abgebildet werden. Dieser wird sowohl in .NETMF, als auch in Java für die Speicherung von Doubles verwendet, der Standard selbst legt auch zusätzlich Vorschriften für Rundung, Wurzelberechnung, arithmetische Operationen etc. fest, erklärt wird dies sehr gut in [Gol91]. Da im .NET Micro Framework die Klasse **BitConverter** eingespart wurde, muss hier, um die einzelnen Bytes zu bekommen, mit **Pointern** direkt auf den Speicher zugegriffen werden, wozu das Schlüsselwort **unsafe** bei der Definition der Methode verwendet werden muss. Dieses soll darauf hinweisen, dass in der Funktion

mit Pointern gearbeitet wird und deren Zugriff nicht kontrolliert wird. Tabelle 5.3 zeigt den Aufbau des Standards IEEE 754.

| **Vorzeichen** | **Exponent** | **Mantisse** |
|---|---|---|
| 1 Bit | 11 Bits | 52 Bits |
| **Summe: 64 Bits = 8 Bytes** | | |

Tabelle 5.3.: Übertragungsschema Double

- **Bool:** Der einfachste Wert zum Codieren ist ein boolscher Wert, allerdings ist es nicht möglich, nur ein Bit zu verschicken, daher wird ein gesamtes Byte verwendet. Für **true** wird **0xFF** verschickt, für **false 0x00**.

#### SerialProcessingInstruction

Mit dieser Klasse wird zu Beginn immer der Befehl vom $Eco_2$-Client gesendet, welche Daten dieser anfordert bzw. sendet. Anhand dieser Information kann dann in die einzelnen Funktionen aufgeteilt werden, die diese Aufgabe erledigen. Zusätzlich muss bei den Befehlen **get_data** und **delete_value** die ID des gewünschten Sensors angegeben werden, für den die Daten gesendet werden sollen und zusätzlich zwei Zeitstempel für den ersten Tag und für den letzten Tag der angezeigt werden soll. Den Aufbau einer Nachricht dieser Klasse zeigt Tabelle 5.4, ein Serialisieren wird hier nicht angeboten, da die Anforderungen immer durch den Client erfolgen und in die umgekehrte Richtung keine Anforderung geschieht.

| | | |
|---|---|---|
| **Auszuführender Befehl** (String) | 10-19 Byte | |
| **ID eines Sensors** (Integer) | 4 Byte | (nur bei get_data und delete_value) |
| **Timestamp Beginn** (Integer) | 4 Byte | |
| **Timestamp End** (Integer) | 4 Byte | |

Tabelle 5.4.: Aufbau einer Nachricht "SerialProcessingInstruction"

#### SerialActor

**SerialActor** wird zum Senden der in der Datenbank gespeicherten Aktoren an den Client verwendet. Dazu stehen die Funktionen *serialize*, *deserialize* und *serializeOnTheFly* zur Verfügung. Die letzte Funktion wurde hinzugefügt, damit nicht für jeden Datensatz ein neues Objekt instanziert werden muss, da dies beim Senden von sehr vielen Datensätzen wesentlich die Geschwindigkeit beeinträchtigt. Ein Aufruf einer statischen Methode ist performanter.

| | |
|---|---|
| **SensorID** (String) | variable Länge |
| **ActorID** (String) | variable Länge |

Tabelle 5.5.: Aufbau der Nachricht "SerialActor"

#### SerialSensor

Die Datenbanktabelle *Sensors* kann mit dieser Klasse übertragen werden. Dazu werden alle Spalten aus der Datenbank an diese Klasse übergeben bzw. werden von dieser Klasse zurück gegeben.

| | |
|---|---|
| **ID** (Integer) | 4 Byte |
| **SensorID** (String) | variable Länge |
| **ValueID** (String) | variable Länge |
| **co2factor** (Double) | 8 Byte |
| **co2offset** (Double) | 8 Byte |
| **absoluteValue** (Integer) | 4 Byte |

Tabelle 5.6.: Aufbau der Nachricht "SerialSensor"

#### SerialSensorData

Die aufgezeichneten Daten der Sensoren können mittels *get_data* Kommando, ID und Zeitstempel mit Beginn und Ende angefordert werden. Diese werden nacheinander geschickt - mit Hilfe der Klasse **SerialSensorData**. Da hier meist sehr viele Datensätze hintereinander geschickt werden, wurde die Nachricht möglichst kurz gehalten und es wird nur der Wert und der entsprechende Zeitstempel gesendet.

| | |
|---|---|
| **value** (Double) | 8 Byte |
| **timestamp** (Integer) | 4 Byte |

Tabelle 5.7.: Aufbau der Nachricht "SerialSensorData"

#### SerializationHelper

Diese Klasse beinhaltet zusätzliche Funktionen, die für den Datenaustausch benötigt werden. *sendSerializedNumberOfElements* sendet nur ein Integer, welches angibt, wie viele Datensätze in Folge gelesen werden können. Es wird darauf vertraut, dass die Anzahl der Elemente korrekt ist, sollte die Übertragung abgebrochen werden, wird am Socket eine *TimeoutException* ausgelöst und der Transfer wird abgebrochen. Für das Empfangen der Anzahl, wie viele Datensätze der Client retour sendet, ist die Funktion *receiveSerializedNumberOfElements* zuständig.

### 5.4.4. gepufferte Socketverbindung

Um die codierten Daten auch im Netzwerk übertragen zu können wird ein **NetworkStream** verwendet, wie bereits weiter oben erwähnt. Allerdings führte die direkte Verwendung dieses Stream zu keinen brauchbaren Ergebnissen - die Geschwindigkeit war nicht akzeptabel. Bei genauerer Betrachtung der Übertragung mittels *Wireshark*, fällt auf, dass der **NetworkStream** die Daten nicht puffert, sondern immer in sehr kleinen Paketen sendet und anschließend auf ein Acknowledged (ACK) oder Not Acknowledged (NACK)

wartet (siehe Abbildung 5.6), obwohl die Puffergröße des Streams einstellbar ist, dies hat aber fast keine Auswirkung auf die gesendeten Pakete oder die Geschwindigkeit der Übertragung.

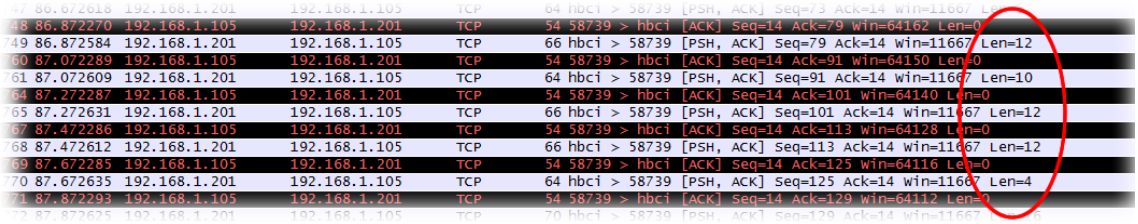



Abbildung 5.6.: Untersuchung der Übertragung bei Verwendung des NetworkStream

Der **BufferedStream**, wie er im .NET Framework enthalten ist wurde im .NET Micro Framework leider eingespart. Daher wurde dieser Stream im Projekt hinzugefügt, es handelt sich dabei um eine abgewandelte Version des **BufferedStream** vom Projekt *redfly* von *Novell*, welcher noch etwas optimiert wurde, einige Funktionsaufrufe und Instanzierungen konnten zusätzlich eingespart werden. Der **BufferedStream** benötigt im Konstruktor einen weiteren Stream, an welchen die Daten übergeben werden (in diesem Fall der **NetworkStream**) und eine Angabe der Puffergröße, welche auf 2000 Byte festgelegt wird, damit ein Datenpaket mit 1500 Byte (maximale Nutzdatenlänge für TCP) locker Platz findet und während des Sendens bereits ein weiteres aufgebaut werden kann.

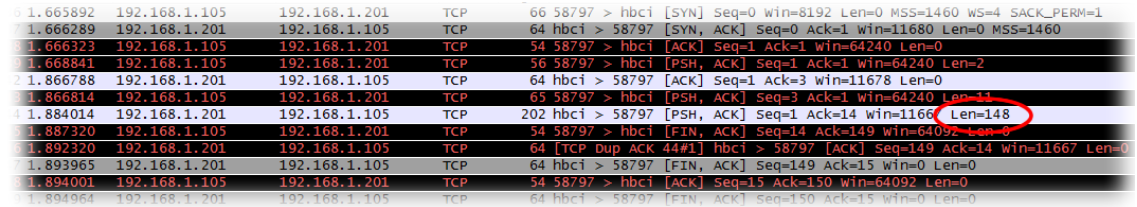



Abbildung 5.7.: Untersuchung der Übertragung bei Verwendung des BufferedStream

Der Geschwindigkeitsunterschied ist das Resultat größerer Pakete, da nicht so oft auf eine Antwort der Gegenseite gewartet werden muss. Die gleiche Beispielübertragung, die in Abbildung 5.6 dargestellt wird, wurde mit dem zusätzlichen Puffer erneut ausgeführt (siehe Abbildung 5.7), wobei hier die 148 Byte Nutzdaten in einem einzigen Paket übertragen werden. Der Traffic konnte von ungefähr 25 Paketen für diese Übertragung auf circa 10 Pakete reduziert werden. Bei größeren Übertragungen wirkt sich dies noch stärker aus.

Um eine optimale Übertragung zu gewährleisten, wurde auch die Anzahl der Daten, die zugleich aus der Datenbank selektiert und gesendet werden, beschränkt. Es werden mehrere Durchläufe mit einer je einer bestimmten Anzahl an Datensätzen ausgeführt, wobei jedes Mal ein Query an die Datenbank geschickt wird und die Daten anschließend codiert in den Puffer geschrieben werden. Die optimale Anzahl der gleichzeitig gelesenen Daten und der Puffergröße wurde experimentell ermittelt (siehe Diagramm in Abbildung 5.8). Mit Hilfe von Time-Profiling wurden diverse Werte erfasst und in das Diagramm eingetragen. Gewählt wurde für die Anzahl der gleichzeitig selektierten Daten 1000 (IOCon-

stants.VALUES_SENT_AT_ONCE_COUNT) und für die Puffergröße 2000 Bytes, da bei höheren Werten keine wesentliche Steigerung des Durchsatzes mehr erreicht werden konnte.

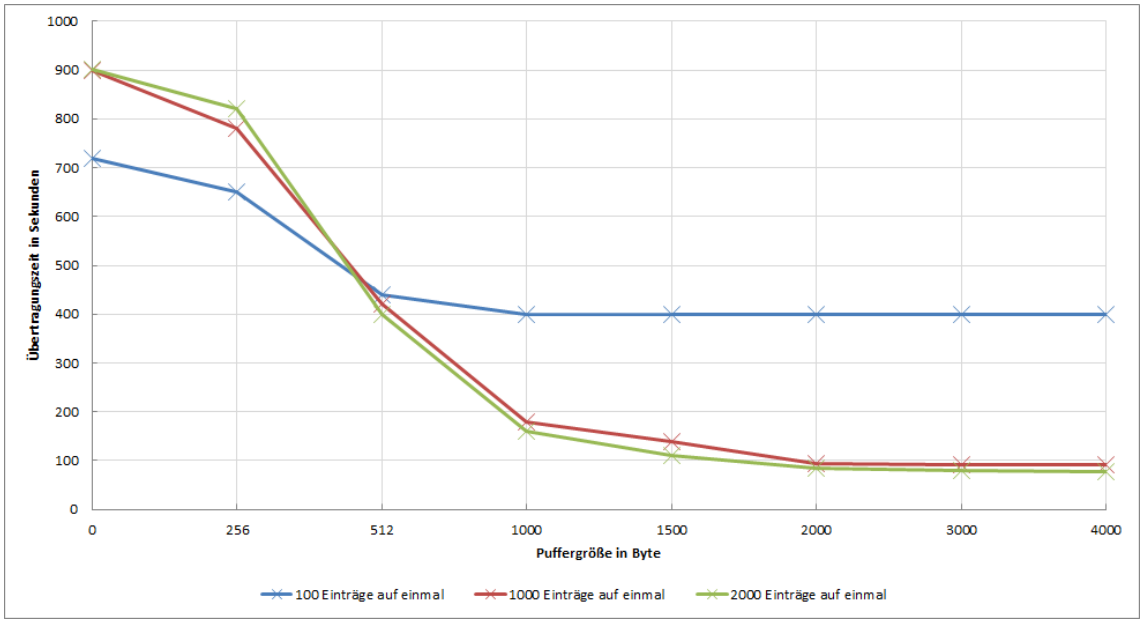



Abbildung 5.8.: Diagramm der Übertragungsdauer für 44500 Werte bei verschiedenen Puffergrößen und verschiedener Anzahl gleichzeitig gesendeter Werte

# Kapitel 6.

# ECO2-Client

Der **$ECO_2$-Client** soll eine mögliche Form der Darstellung der Daten am PC zeigen. Es wurde dazu ein Java-Programm entwickelt, welches die Daten beim $ECO_2$-Manager über Ethernet anfordert, empfängt und dementsprechend darstellt. Zudem soll die Parametrisierung (Einstellung der Sensoren und Aktoren) mit Hilfe dieses Programms durchgeführt werden können.

## 6.1. Programmstart und grafische Struktur

Die Klasse **$ECO_2$-Client** wird bei diesem Projekt als Main-Klasse verwendet und erledigt einige Dinge zu Beginn des Programmstarts:

- **SQLite JDBC Treiber laden:** Als erstes nach dem Programmstart wird in der Methode *main* überprüft, ob das Paket **org.sqlite.JDBC** erreichbar ist und der Datenbanktreiber wird dynamisch geladen und initialisiert (mit `Class.forName()`). Der Treiber wird später für den Zugriff auf SQLite-Datenbankdateien verwendet. Tritt ein Fehler beim Laden der Klasse auf, wird dies mitgeloggt und das Programm mit Rückgabewert -1 beendet.
- **Starten der grafischen Oberfläche:** Durch Instanzieren und sichtbar setzen des **MainFrames** wird die grafische Oberfläche gestartet.

### 6.1.1. MainFrame

Um ein Diagramm darstellen zu können muss zuerst eine Quelle ausgewählt werden, dazu stehen Ethernet oder eine Datei zur Auswahl, wobei Ethernet nur anwählbar ist, wenn der $ECO_2$-Manager im Netzwerk gefunden wurde. Anschließend sollte der gewünschte Zeitraum gewählt werden, wurde als **Verbindungsart Ethernet** gewählt, darf die **Zeitspanne maximal 31 Tage** betragen, da sonst die zu übertragende Datenmenge zu groß ist, mit **Ausnahme des Kuchendiagramms (Pie Chart)**, dieses kann auch für eine längere Zeit dargestellt werden. Liniendiagramme für längere Zeiträume wie ein Monat machen auch wenig Sinn, da nichts mehr zu erkennen ist. Das Diagramm selbst wird nach einem Klick auf den entsprechenden Button gezeichnet. Welche Diagramme derzeit zur Auswahl stehen wird in Abschnitt 6.4 beschrieben.

Die Klasse **MainFrame** ist der Hauptbestandteil des Programms. Hier werden alle Aktionen der grafischen Oberfläche weitergeleitet, andere Klassen instanziert und der Ablauf des Programms gesteuert. Im Konstruktor wird der **BroadcastReceiver** (siehe Unterabschnitt 6.2.1) instanziert, der Thread zum Empfangen des UDP-Broadcasts gestartet und der **CommunicationManager** (siehe Unterabschnitt 6.2.2) instanziert. Diese Instanzen werden in Membervariablen für den späteren Zugriff bereitgehalten.

Die **grafische Oberfläche** wurde in *NetBeans* erstellt und wird über die Methode *initComponents* aufgerufen. Diese erstellt die Grundstruktur der grafischen Oberfläche wie sie in Abbildung 6.1 dargestellt wird. Diese Grundstruktur wird durch diverse Methoden verändert und den Reaktionen des Users bzw. den verfügbaren Verbindungen angepasst.


Abbildung 6.1.: Screenshot $ECO_2$-Client beim Start

Weiters wird im Konstruktor die Zeit des **Start-Buttons** und des **End-Buttons** auf das aktuelle Datum gesetzt und **checkDateAndUpdateText** aufgerufen. Diese Methode setzt die Zeit des Starts auf 00:00:00 Uhr und die Zeit vom Ende auf 23:59:59. Zusätzlich wird überprüft, ob das *von-Datum* nach dem *bis-Datum* liegt und gegebenenfalls der gleiche Tag dafür verwendet. Um die Zeit zu visualisieren, werden die beiden Tage in das *fromLabel_* und das *toLabel_* eingetragen, diese werden jeweils neben den beiden Buttons zur Datumsauswahl positioniert.

Damit wurde die grundlegende Oberfläche initialisiert und wird dem Benutzer angezeigt.

### Funktionen des MainFrames

- **setConnectionStatus** aktiviert oder deaktiviert die Ethernetverbindung zum $ECO_2$-Manager. Wird der Methode *true* übergeben, werden entsprechenden Funktionen

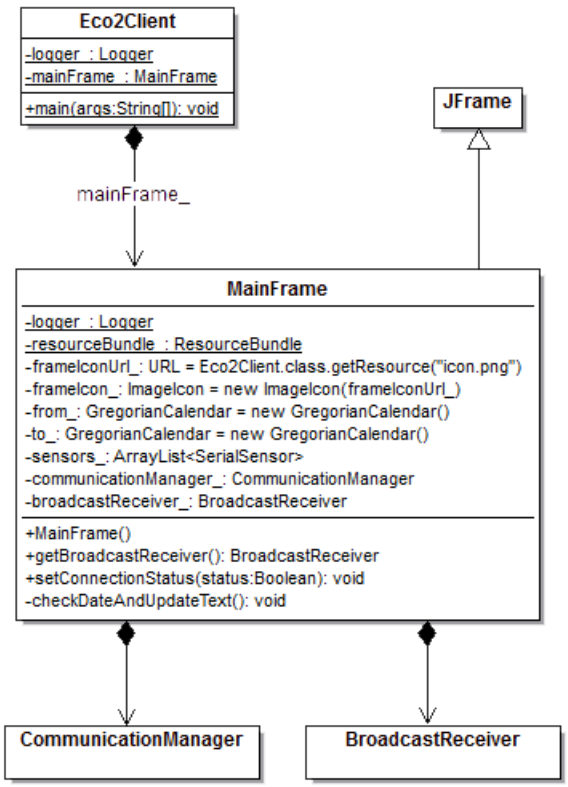



Abbildung 6.2.: Klassendiagramm $ECO_2$-Client-Start und MainFrame

und Buttons für Ethernet aktiviert. Damit signalisiert der **BroadcastReceiver**, dass ein Broadcast vom $ECO_2$-Manager empfangen wurde und dessen IP nun bekannt ist.

- **fillInSensors** holt die Sensoren von der aktuellen Datenquelle (SQLite oder Ethernet) über den **CommunicationManager**.
- **showFileChooserAndOpenDatabase** zeigt einen Dialog zum Auswählen einer Datei an. Dabei wird der **sqlFileChooser**, dem bereits zu Beginn im Konstruktor mit der Klasse **MyFileFilter** mitgeteilt wurde, welche Dateien akzeptiert werden, aktiviert. Im Anschluss an die Auswahl einer SQLite-Datei (Dateiendung .dbs) wird dem **CommunicationManager** der Pfad zur Datenbankdatei mitgeteilt und bei Erfolg *true* zurück gegeben. Bricht der Benutzer ab oder wählt eine falsche Datei wird *false* zurückgegeben.
- **enableChartButtons/disableChartButtons** aktiviert/deaktiviert die Buttons zum Zeichnen der Diagramme. Diese sind deaktiviert, solange keine gültige Datenquelle existiert.
- **...ChartButtonActionPerformed** wird aufgerufen, wenn ein entsprechendes Diagramm per Button ausgewählt wurde. Wie die Diagramme gezeichnet werden wird in Abschnitt 6.4 erklärt.

Es wurden in der Liste die wichtigsten Methoden beschrieben. Die Interaktionen und Elemente der grafischen Oberfläche wurden nicht alle erwähnt, diese wurden zum größten Teil automatisch generiert und können mit *NetBeans* bearbeitet werden.

### 6.1.2. SensorSettingsFrame und Hilfsklassen

Dieser Frame wird für die **Einstellungen der Sensoren** verwendet, diese können allerdings nur bei vorhandener Ethernetverbindung getroffen werden. Mit Hilfe von *Netbeans* wurde eine Tabelle erstellt, die die Spalten der Einträge in der Datenbank für die Sensoren enthält (Nummer, Sensor ID, Value ID, $CO_2$-Offset, $CO_2$-Faktor und Absolutwert). Diese Eingaben können hier korrigiert werden. Ein Löschen eines Eintrags ist nicht möglich, da für die Speicherung der Sensordaten die eindeutige Nummer aus der ersten Spalte verwendet wird und ein Löschen zu Inkonsistenz der Daten führen würde. Mit dem Button **Add Row** kann eine neue Zeile hinzugefügt werden, wobei die Nummer in der ersten Spalte automatisch erhöht wird. Wurden alle gewünschten Einstellungen getätigt, können die Werte mit dem Button **Save** zurück an den $ECO_2$-Manager gesendet werden. Wurde das Speichern der Daten vergessen, wird man beim Schließen des Fensters mit einem entsprechenden Dialogfenster darauf hingewiesen. Ein Screenshot des Einstellungsfensters ist in Abbildung 6.3 dargestellt.

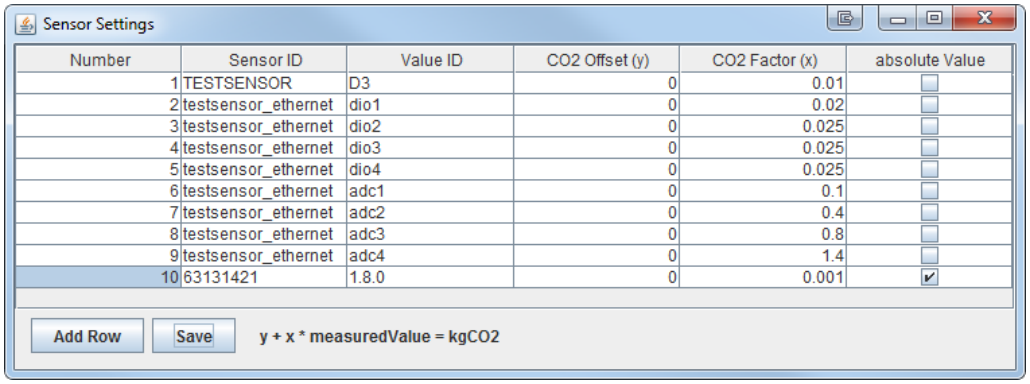



Abbildung 6.3.: Screenshot Fenster Sensor Settings

#### Erklärung der Funktionsweise

Im Konstruktor der Klasse **SensorSettingsFrame** werden als erstes die Komponenten mit der, von *NetBeans* generierten Methode *initComponents* intialisiert und ein *WindowListener* hinzugefügt, welcher das Schließen des Fensters auf die Methode *close* umleitet, damit beim Schließen des Fensters ohne Speichern der Werte ein entsprechender Dialog angezeigt wird. Zudem wird im Konstruktor das Tabellenmodell als Membervariable abgespeichert und ein **SensorTableListener** hinzugefügt.

Die am $ECO_2$-Manager vorhandenen Sensoreinstellungen werden vom **CommuncationManager** angefordert, wobei hier zusätzlich angegeben wird, dass die Daten zwingend über das Netzwerk geholt werden sollen. Diese Daten werden mit Hilfe des *TableModel* in die Tabelle eingefügt. Alle weiteren Methoden werden in Folge kurz erklärt, das Klassen-

diagramm ist in Abbildung 6.4 dargestellt. Hier wurden die autogenerierten Methoden und Membervariablen für die grafische Oberfläche zwecks Übersichtlichkeit weggelassen.

- **displayErrorBox** zeigt eine Fehlermeldung in einem Popup an.
- **sendTableContent** schreibt die in der Tabelle vorhandenen Sensoren in eine ArrayList von **SerialSensor**-Elementen und gibt diese an den **CommunicationManager** zum Senden weiter.
- **close** wurde als *WindowListener* für das Schließen des Fensters hinzugefügt. Wird das Fenster geschlossen, wird im **SensorTableListener** nachgesehen, ob alle Änderungen gespeichert wurden, wenn nicht, wird eine dementsprechende Abfrage über die Methode **askForSaving** angezeigt und die Daten werden bei Bedarf gespeichert.
- **addRowButtonActionPerformed** wird bei Klicken auf "Add Row" aufgerufen. Es wird die höchste Nummer der aktuellen Tabelle herausgesucht und in das *TableModel* wird eine neue Zeile mit der um eins inkrementierten Nummer eingefügt.
- **saveButtonActionPerformed** ist für die Abarbeitung des Speichervorgangs bei einem Klick auf "Save" zuständig. Dabei wird das Editieren der Tabelle beendet und die Methode *sendTableContent* wird aufgerufen um die Daten abzusenden.

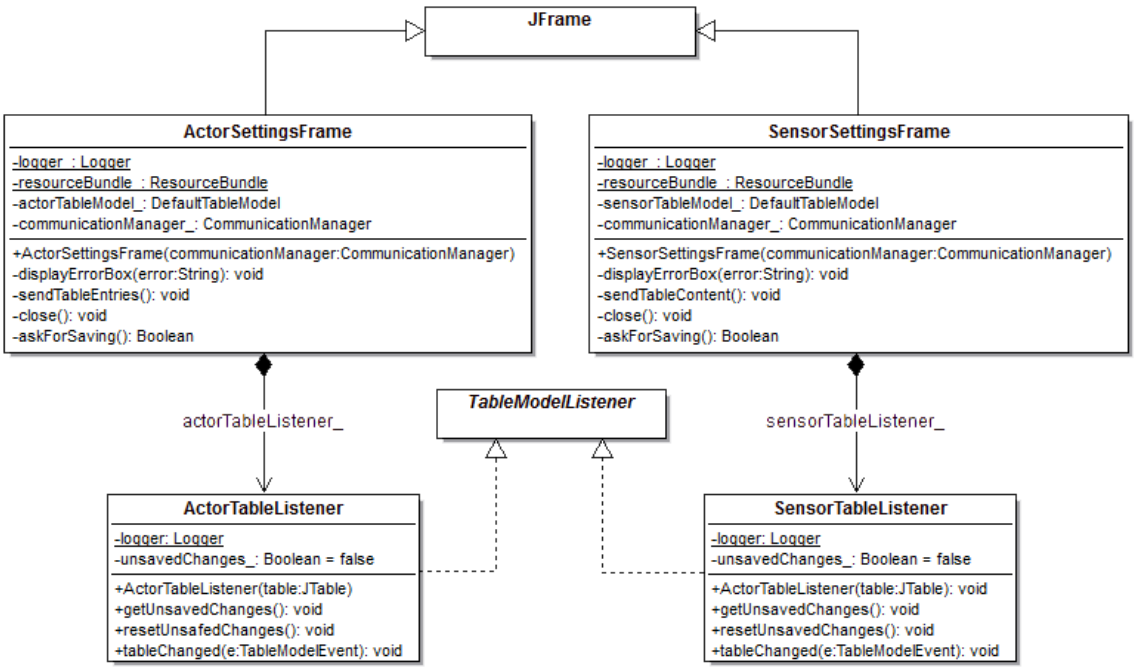



Abbildung 6.4.: Klassendiagramm SensorSettingsFrame und ActorSettingsFrame mit Hilfsklassen und Ableitungen

#### SensorTableListener

Diese Klasse implementiert das Interface eines **TableModelListeners**. Vor allem die Methode *tableChanged* ist von Interesse, da diese bei jeder Änderung der Tabelle aufgerufen wird. Mit Hilfe dieser Methode wird kontrolliert, ob doppelte Einträge vorhanden sind, sollte dies der Fall sein wird der doppelte Eintrag automatisch entfernt, um die Datenbank konsistent zu halten. Die beiden Methoden *resetUnsafedChanges* und *getUnsafedChanges* werden für die Sicherheitsabfrage beim Schließen des Fensters benötigt.

### 6.1.3. ActorSettingsFrame

Diese Klasse ist vom Prinzip gleich wie der **SensorSettingsFrame**, die Tabelle mit den Einstellungen enthält dieses Mal allerdings nur zwei Spalten: Sensor ID und Actor ID (siehe Abbildung 6.5). Die Funktionalität ist ansonsten die gleiche wie im Kapitel zuvor erklärt, abgeändert auf die Aktoren, mit der Ausnahme, dass hier auch ein Löschen einer Zeile mit der Funktion *deleteRowButtonActionPerformed* ermöglicht wird, es können beliebig viele Aktoren hinzugefügt und entfernt werden. Den Aufbau der Klasse mit den entsprechenden Hilfsklassen zeigt ebenfalls Abbildung 6.4, die Klasse wurde in der Abbildung zusammengefasst mit dem **SensorSettingsFrame**.

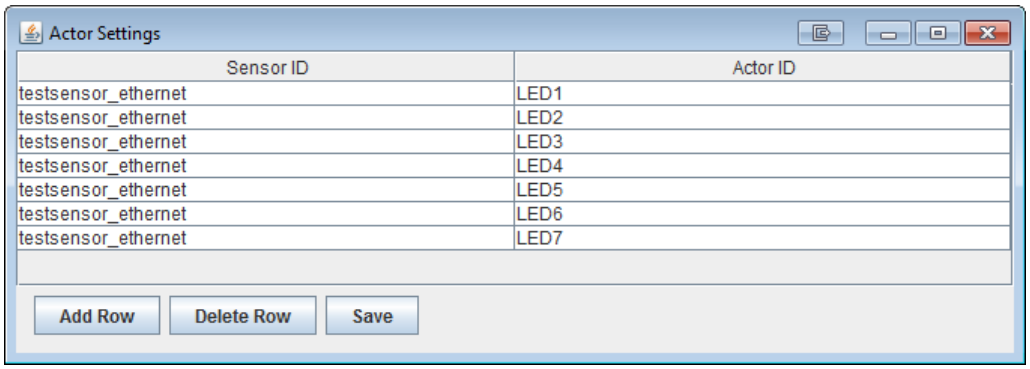



Abbildung 6.5.: Screenshot ActorSettingsFrame

### 6.1.4. AboutDialog

Als Info über das Projekt wurde ein About-Dialog eingebaut, welcher Auskunft über die aktuelle Version, den Autor und des Release-Datums des $ECO_2$-Clients gibt. Dieser Dialog wurde eingeführt, um eine Übersicht über die verschiedenen Varianten zu behalten, die ausprobiert wurden.

## 6.2. Package Communication

Dieses Java-Paket beinhaltet alle Klassen, die für die Kommunikation verantwortlich sind. Es werden Daten per Ethernet empfangen und verschickt bzw. Daten per Java Database Connectivity (JDBC) von einer SQLite-Datenbank eingelesen. Die einzelnen Komponenten werden in den folgenden Kapiteln beschrieben.

### 6.2.1. BroadcastReceiver

Um dynamische IP-Adressvergabe zu ermöglichen, sendet der $ECO_2$-Manager per UDP Broadcasts mit einer "Hello Message" aus. Diese können vom $ECO_2$-Client empfangen werden, was durch den **BroadcastReceiver** umgesetzt wurde. Gestartet wird dieser im Konstruktor des **MainFrame**, wobei dort die Methode *Start* aufgerufen wird, welche den Thread für den Empfang startet. Bereits zuvor wird im Konstruktor der Klasse ein neues **DatagramSocket** mit dem gewünschten UDP-Socket angelegt. Das Timeout für den Empfang wird auf die doppelte Zeit des Broadcast-Intervalls gestellt, damit eine unterbrochene Verbindung anhand der fehlenden Pakete erkannt werden kann.

Wird der Thread gestartet, legt die Methode *run* ein neues **DatagramPacket** an. In einer Schleife wird daraufhin immer wieder Receive am UDP-Socket aufgerufen. Wird innerhalb des Timeouts ein Paket empfangen, kann der Inhalt aus diesem ausgewertet und mit der definierten Broadcast-Nachricht des $ECO_2$-Managers verglichen werden (*CommunicationConstants.UDP_BROADCAST_MESSAGE*). Stimmt die Nachricht überein, wird die Adresse, von der das Paket gekommen ist ausgelesen und in der Membervariable *eco2managerAddress_* notiert. Der **MainFrame** wird über den neuen Status informiert.

Wird allerdings innerhalb des Timeouts keine entsprechende Nachricht empfangen, wird die Adresse auf `null` gesetzt und der **MainFrame** über das Fehlen der Verbindung informiert.

Über die Methode *getEco2MasterAddress* können andere Klassen die IP-Adresse des Managers beziehen.

### 6.2.2. CommunicationManager

Die Verwaltung der Kommunikationsschnittstellen übernimmt der **CommunicationManager**. Derzeit ist die Kommunikation über Ethernet und mittels JDBC für SQLite implementiert, sollten weitere Schnittstellen gewünscht sein, können diese implementiert und zum Manager hinzugefügt werden. Welche Kommunikation verwendet werden soll wird in der Membervariable *communicationType_* angegeben. Die Methoden, die Daten lesen verwenden immer diesen angegebenen Kommunikationstyp, geschrieben werden kann allerdings nur über das Ethernet, da nur dort die Daten für die Sensoren und Aktoren gespeichert sind und verändert werden können. In die SQLite-Datenbank wird in keinem Fall geschrieben, die Datenänderung darauf hätten keine Auswirkung auf die Erfassung der Sensordaten.

Im Konstruktor werden die beiden derzeit bereits vorhandenen Kommunikationsklassen **Eco2EthernetConnection** und **SQLiteCommunication** für die weitere Verwendung instanziert, die Methoden zum Senden und Empfangen im **CommunicationManager**

leiten weiter an diese Klassen und übernehmen keine Funktionalitäten für den Datenaustausch.

Folgende Methoden stehen zur Verfügung:

- **getSensors** liefert eine Liste von **SerialSensor**-Objekten zurück. Sie enthält alle Sensoren, die im System bekannt sind mit deren Parametern. Alternativ kann für diese Methode auch der Parameter *forceEthernet* angegeben werden, wird dieser auf *true* gesetzt, findet die Kommunikation, sofern möglich, ausschließlich über Ethernet statt. Dies muss bei den Einstellungen der Sensoren erfolgen, da sonst eventuell die falschen Sensoren aus der Datenbankdatei geladen werden.
- **getSensorData** liefert eine **ArrayList** von **SerialSensorData**-Objekten zurück. Als Parameter muss die gewünschte ID des Sensors, ein Zeitstempel für den Beginn der Leseperiode und ein Zeitstempel für das Ende der Leseperiode angegeben werden.
- **getSensorDataSum** fordert eine Datensumme an. Es werden die gleichen Paramameter wie bei **getSensorData** angegeben, dieses mal wird aber die Summe für den angegebenen Zeitraum angefordert. Zurück geliefert wird ein einziges **SerialSensorData**-Objekt, als Wert wird aber die Summe eingesetzt.
- **sendSensors/sendActors** kann nur mit aktiver Ethernetverbindung verwendet werden und sendet die mittels ArrayList übergebenen Sensoren/Aktoren zurück an den $Eco_2$-Manager. Ist die Ethernetverbindung unterbrochen führt dies zu einer Exception.
- **receiveActors** funktioniert ebenfalls nur mit Ethernet und es werden alle vorhandenen Aktoren als ArrayList zurückgegeben.

Das Zusammenspiel zwischen **CommunicationManager**, **Eco2EthernetConnection**, **Eco2ManagerSocket** und **SQLiteCommunikation** ist in der Abbildung 6.6 dargestellt.

### 6.2.3. Eco2EthernetConnection

Die Kommunikation über Ethernet mit dem $ECO_2$-Manager wird über diese Klasse abgewickelt. Dazu werden diverse Methoden angeboten, die Daten senden oder empfangen, zurückgeliefert wird jeweils ein spezifisches Objekt aus dem **Package Serialization** (siehe Abschnitt 6.3) oder eine ArrayList mit diesen Objekten, sollten mehrere Datensätze erwartet werden. Die einzelnen Methoden sind gleich benannt wie im **CommunicationManager**, implementieren hier aber die eigentliche Funktionalität. Der Ablauf der Methoden ist ähnlich - für eine Datenanforderung wird ein **Eco2ManagerSocket** erstellt, eine **SerialProcessingInstruction** mit der Anforderung für die gewünschten Daten gesendet, eventuell mit zusätzlichen Infos wie Sensor-ID und Zeitstempel, anschließend werden die Daten, die seriell empfangen werden deserialisiert und in das entsprechende Objekt gespeichert.

Die Methoden für den Versand von Daten erstellen ebenfalls ein Socket-Objekt, die entsprechende Instruktion, die Anzahl der zu sendenden Elemente und alle Objekte werden im Anschluss serialisiert gesendet.

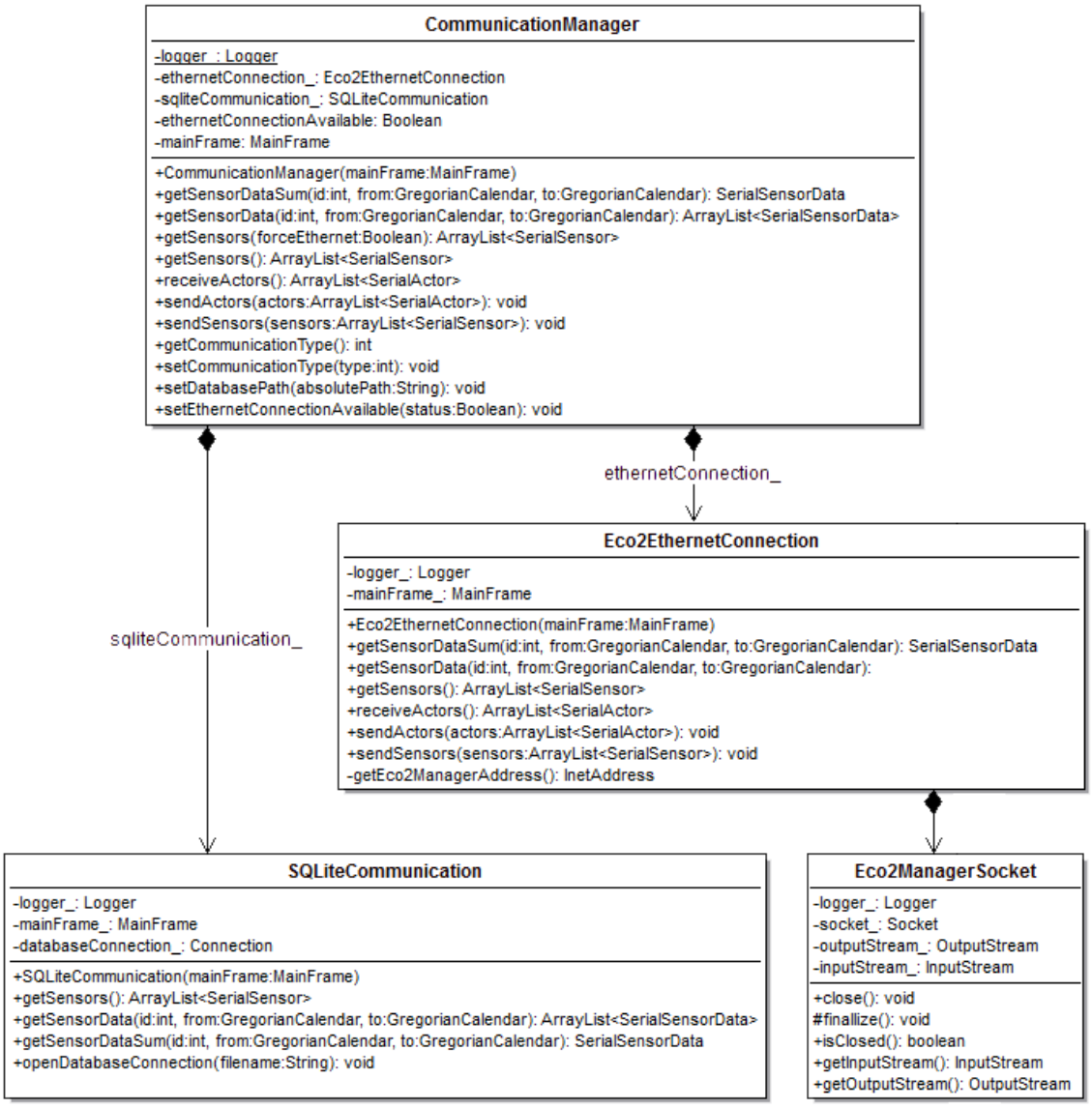



Abbildung 6.6.: Klassendiagramm CommunicationManager und Kommunikationsklassen

### Eco2ManagerSocket

Senden und Empfangen wird beim $ECO_2$-Client mit TCP-Streaming-Sockets erledigt. Um nicht jedes Mal eigens die Initialisierung durchführen zu müssen und sicher zu gehen, dass der Port auch wieder geschlossen wird, wurde die Klasse **Eco2ManagerSocket** implementiert. Diese setzt im Konstruktor bereits die gewünschten Einstellungen wie die IP-Adresse, die Portnummer, Timeouts, verbindet mit dem gewünschten Socket und holt den In- und Outputstream um beide als Klassenmember zu speichern. Mit Hilfe diverser Funktionen kann auf den Socket geschrieben (Outputstream), von ihm gelesen (Inputstream) und der Socket wieder geschlossen werden (close). Sollte das Schließen vergessen worden sein, wird

dies auch automatisch durch den Destruktor durchgeführt.

### 6.2.4. SQLiteCommunication

Ähnlich wie zuvor für die Ethernetkommunikation beschrieben, werden hier die Methoden des **CommunicationManagers** implementiert, allerdings einige weniger, da die Methoden zum Senden der Sensoren und Aktoren wegfallen und die Aktoren in der Datenbank nicht zu finden sind. Ansonsten werden wiederum die gleichen Methodennamen verwendet, wie bereits im Manager, hier werden die Daten allerdings aus einer Datenbankdatei gelesen.

Werden Daten angefordert, werden diese in der Datenbank selektiert, in das entsprechende Objekt verpackt und zurückgegeben. Der Zugriff auf die Datenbank erfolgt mittels SQLite JDBC Library, diese kann wie andere JDBC Treiber verwendet werden. Die Verbindung zur Datenbank wird aufgebaut, wenn eine Datei ausgewählt wurde, diese Verbindung wird dann in der Klasse **SQLiteCommunication** als Member gehalten.

Um Daten selektieren zu können, wird ein **Statement** erstellt und mit diesem Statement das entsprechende Query ausgeführt. Das vom Query zurückgelieferte **ResultSet** kann iteriert werden, die Daten werden während der Iteration in ein neues Objekt der entsprechenden Klasse geschrieben und der ArrayList hinzugefügt. Wird keine ArrayList benötigt entfällt die Iteration und der Wert wird direkt zurückgegeben. Die Queries ähneln denen am $ECO_2$-Manager in der Klasse **ProcessClientRequest**, da hier als Datenbank die gleiche Struktur zur Verfügung steht wie am Embedded System, da die Byte-Reihenfolge bei Integer-Werten vertauscht wird.

## 6.3. Package Serialization

In diesem Paket finden sich die Klassen für die Serialisierung von Objekten, die im Projekt verwendet werden. Die Klassen werden äquivalent zu denen am Embedded System implementiert und daher wird die Funktionsweise hier nicht mehr genauer erklärt (die Implementierung in C# wird in Unterabschnitt 5.4.3 erklärt). Einige Unterschiede bestehen jedoch - in Java wurde das Interface **ISerialization** eingeführt, welches alle Klassen für Datenserialisierung implementieren müssen und Java Virtual Machines (VMs) sind immer Big-Endian, wodurch die Codierung etwas anders aussieht als am Embedded System.

## 6.4. Package Chartframes

Die eigentlichen Diagramme, die dargestellt werden sollen, werden mit den Klassen in diesem Paket erzeugt. Es wurden zur Auswertung einige Beispieldiagramme generiert, weitere Varianten können mit der **Library JFreeChart** jederzeit hinzugefügt werden. Alle Diagramme werden im unteren Bereich des **MainFrame** angezeigt. Wird der entsprechende Button im **MainFrame** gedrückt, werden die benötigten Daten über den **CommunicationManager** geholt und für die Generierung des Diagramms bereitgestellt. Dies kann, wenn eine längere Zeitspanne und Ethernetkommunikation gewählt wurde durchaus einige Zeit in Anspruch nehmen. Anschließend wird der entsprechende **ChartFrame** (Ableitung

von **JInternalFrame**) instanziert und als *ContentPane* für den **chartViewerFrame** verwendet und das Diagramm sichtbar gesetzt. Damit wird das Diagramm im unteren Bereich des **MainFrames** eingeblendet.

### PieChartFrame

Der erste Diagrammtyp zeigt ein Kuchendiagramm, wobei die Summen für alle Sensoren ungleich Null angezeigt werden. Die Verteilung des Verbrauchs wird als Kuchendiagramm dargestellt, fährt man über die einzelnen Teile, erhält man genaue Infos über den Prozentsatz des entsprechenden Sensors und der Gesamtsumme des $CO_2$-Ausstoßes in $kgCO_2$. Ein Beispiel mit drei Testsensoren ist in Abbildung 6.7 dargestellt, wobei auch der Tooltip für den *testsensor_ethernet/dio1* dargestellt ist.

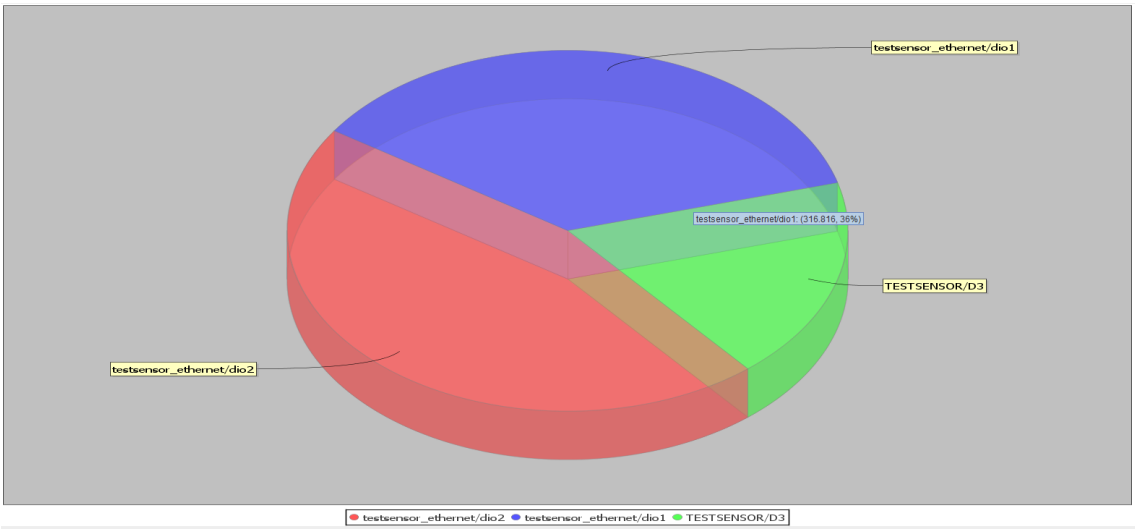



Abbildung 6.7.: Beispiel für ein Kuchendiagramm (PieChart)

Zum Erstellen dieses Diagramms muss dem Konstruktor eine **Hashmap** mit **String-Double-Paaren** übergeben werden, in denen der Name des Sensors und der Wert der Summe steht. Aus diesem wird mittels *createDataset* ein **PieDataset** erstellt, indem die Hashmap durchiteriert wird. Aus dem Dataset wird anschließend ein **JFreeChart** erstellt, wobei hier ein Titel angegeben werden kann, der über dem Diagramm angezeigt wird. Zudem werden in der Methode *createChart* Einstellungen getroffen, die das Aussehen und Verhalten des Diagramms beeinflussen. Was hier alles genau eingestellt werden kann, kann der Dokumentation der **Library JFreeChart** [Gil12] entnommen werden. Zum Schluss wird noch ein **ChartPanel** erstellt und dieses als **ContentPane** für den **JInternalFrame**, von dem die Klasse abgeleitet wurde verwendet.

### TimeSeriesChartFrame

Das gleiche Vorgehen wie für den vorherigen Diagrammtyp wird auch hier angewandt. Es wird diesmal allerdings ein **TimeSeriesChart** erstellt, welches die Datenwerte für alle

Sensoren enthält, wie in Abbildung 6.8 zu sehen, wobei diese Werte absichtlich sehr variabel angelegt wurden, um die Interpolation mittels Splines zu zeigen. Die Interpolation wurde durchgeführt um ein lesbares Diagramm zu bekommen, ohne diese entstehen sehr viele Zacken. Um die Interpolation zu erreichen, wird der Graph mittels **XYSplineRenderer** gezeichnet, dieser erledigt die Interpolation von selbst. Die Datenpunkte wurden nicht eingezeichnet, da dies bei sehr vielen Messpunkten zu einer Unlesbarkeit des Diagramms führt.

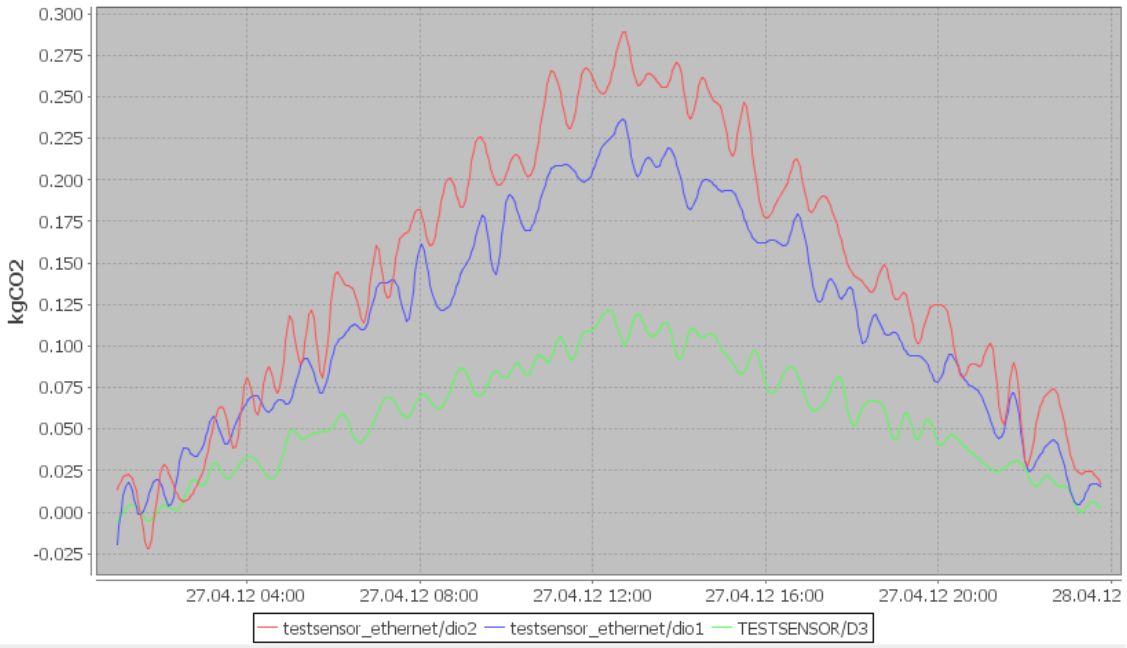



Abbildung 6.8.: Beispiel für ein Liniendiagramm (TimeSeriesChart)

### TimeSeriesSumChartFrame

Eine Erweiterung des obigen Diagramms stellt dieser Frame dar. Hier werden zwei Plots kombiniert, ein Plot zeigt, wie bereits vorher, die Werte aller Sensoren einzeln als Linie dargestellt, zusätzlich wird in einem eigenen Plot die Summe aller Sensoren aufgezeichnet. Diese wurde in einen eigenen Plot ausgelagert, da ansonsten die Skalierung der Achsen nicht sinnvoll möglich wäre. Das Diagramm wird wie in Abbildung 6.9 ersichtlich mit zwei getrennten Diagrammen geplottet.

### Diagrammfunktionen

Die Diagramme des $ECO_2$-Client bieten auch einige Funktionen für eine Auswertung. In den einzelnen Diagrammen kann gezoomt werden, indem ein Rechteck für den gewünschten Bereich aufgezogen wird oder, alternativ, über einen Rechtsklick in das Diagramm. Dort kann der Zoom auch wieder zurückgesetzt werden (Auto Range).

Weiters stehen in den Liniendiagrammen Fadenkreuze zur Verfügung, die **auf den Messpunkten** (nicht auf der Interpolierung) positioniert werden können und die helfen, die

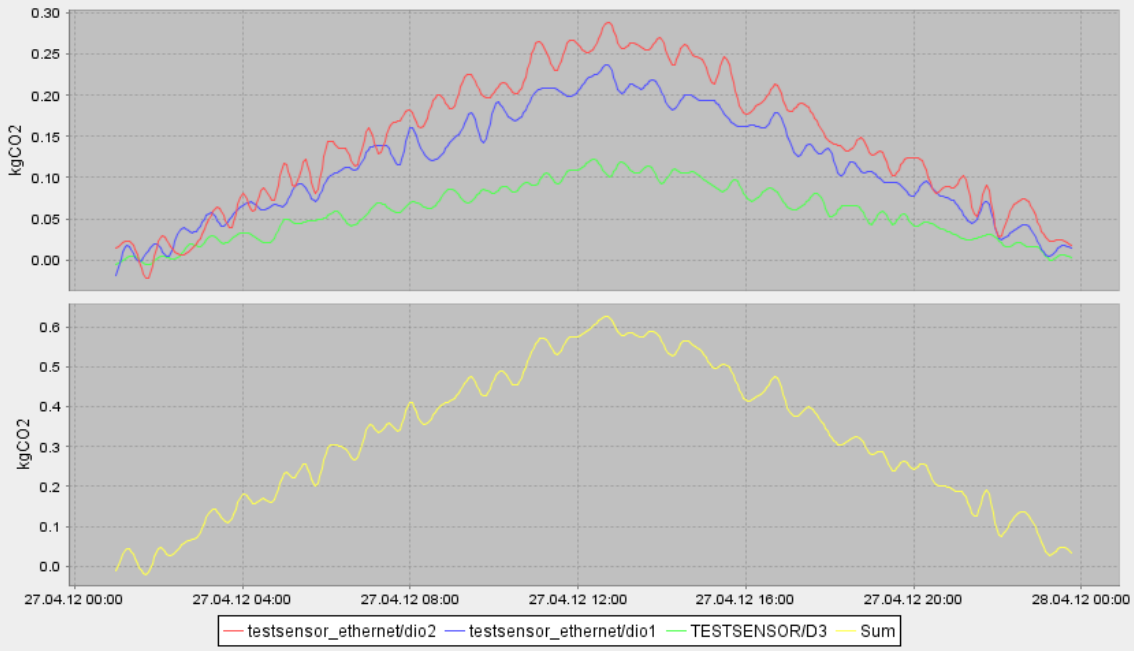



Abbildung 6.9.: Beispiel für ein Liniendiagramm mit Summenanzeige (zwei TimeSeriesCharts)

Werte abzulesen. Die Kreuze können mit einem einfachen Klick in das Diagramm angezeigt werden.

Funktionen wie Ausdrucken, als Grafik kopieren oder eine Speicherfunktion für die Grafik stehen ebenfalls zur Verfügung. Sollten weitere Diagramme gewünscht sein, können diese mit der JFreeCharts-Library sehr gut umgesetzt und hinzugefügt werden. In [Gil12] finden sich viele gute Beispiele für Diagramme, mit deren Hilfe neue Diagramme entwickelt werden können. In der Library stehen unzählige Funktionen zur Verfügung, die eine Gestaltung der Diagramme nach den eigenen Vorstellungen ermöglichen.

## 6.5. Internationalisierung

Damit das Programm in verschiedenen Sprachen verwendet werden kann wurden Texte, Datumsangaben und dergleichen mittels **I18N** internationalisiert. Dazu wurden Ressource-Dateien im **Paket properties** eingefügt, für jede betroffene Klasse und Sprache eine eigene Datei. Das *NetBeans IDE* unterstützt automatische Internationalisierung seit Version 6.9, dazu muss, wenn ein neues Design erstellt wird, der oberste Knoten des Designs gewählt werden (im Inspector des Designers) und in den Properties das Häkchen **Automatic Internationalization** gesetzt werden. NetBeans legt dann automatisch ein Paket in der aktuellen Sprache an. Eine zusätzliche Sprache kann mittels Rechtsklick auf die angelegte *Lokale* und dann **Add/Locale...** zum Projekt hinzugefügt werden. Es kann dann eine Sprache und eine Region ausgewählt werden, für die diese Sprache verwendet werden soll. Wurde diese angelegt, kann mittels Rechtsklick auf die ursprüngliche Datei und **Open** eine Tabelle für alle Lokalisationen angezeigt werden. Hier lassen sich alle Texte bequem

bearbeiten. [Ora12]

Um auch Fehlermeldungen und andere Strings zu internationalisieren, kann das **ResourceBundle** auch manuell eingebunden werden. Dazu kann mittels `ResourceBundle .getBundle("properties/BundleName")` ein Verweis erstellt werden. Wird hier zusätzlich eine spezifische Sprache angegeben, wird diese geladen, ansonsten der Systemstandard. Nach dem Bereitstellen der Ressource, zum Beispiel in einer Membervariable der Klasse, kann mit `resourceBundle.getString("Stringname")` auf den gewünschten String in der entsprechenden Sprache zugegriffen werden. Es werden auch noch weitere Funktionen, wie verschiedene Datumsformate etc. unterstützt, dies wurde hier allerdings nicht verwendet. Beim Start des Java-Programms wird die Systemeinstellung für die Sprache und Örtlichkeit verwendet, man kann allerdings mittels zusätzlich angegebener Option beim Start die gewünschte Sprache angeben. Wie man zum Beispiel englische Sprache und eine englische Örtlichkeit erzwingt, zeigt das folgende Beispiel: [Shi02]

```
java Eco2Client -Duser.language=en -Duser.country=en
```

# Kapitel 7.

# Smart Meter

## 7.1. Allgemeines

Als einer der wichtigsten Sensoren zum Berechnen des $CO_2$-Haushalts zählt sicher die elektrische Energie, die in einem Haushalt umgesetzt wird. Bisher sind in vielen Haushalten noch **mechanische Energiezähler** vorhanden, die nach dem **Ferraris-Prinzip** arbeiten und die Wirkleistung aufzeichnen. Diese mussten immer wieder vom Besitzer oder einem Mitarbeiter von einem lokalen Energieversorgungsunternehmen (EVU) abgelesen werden. Die Daten wurden dann meist am PC erfasst, mussten aber händisch abgelesen werden.

In Zukunft werden diese Zähler immer mehr durch einen elektronischen Zähler abgelöst, dem sogenannten **Smart Meter**. Die Erfassung des Energieverbrauchs erfolgt dort mittels verschiedener Methoden, wie zum Beispiel mit einem Hall-Sensor oder Stromwandlern. Ein Smart Meter besteht meist aus verschiedenen Kontrolleinheiten, diversen Sensoren und einer Schnittstelle für eine Datenübertragung, wobei derzeit fast alle am Markt erhältlichen Smart Meter zusätzlich eine Anzeige (z.b. Flüssigkristallanzeige) besitzen, um auch eine manuelle Ablesung durchführen zu können. In Zukunft wird die Verwendung von Smart Metern noch wesentlich mehr an Bedeutung gewinnen, um Lastprofile des Netzes besser erstellen zu können und zukünftige Entwicklungen, wie auch zum Beispiel die Auswertung des $CO_2$-Ausstoßes zu ermöglichen. Spezielle Smart Meter berücksichtigen auch eine Einspeisung einer Photovoltaikanlage oder können verschiedene Tarifzeiten definieren. [DWDG11]

In vielen Ländern sind auch Smart Meter für den Gasverbrauch oder für das Erfassen von verbrauchter Energie von Fernwärmeheizwerken im Einsatz, wobei die Versorgungsunternehmen vom **Smart Metering** sprechen. Zum Smart Metering zählen die digitale Erfassung von Messgrößen und die Datenübermittlung vom Verbraucher an den Lieferanten. Durch die Einführung dieser elektronischen Messinstrumente soll der Energieverbrauch (Strom, Gas, Fernwärme...) transparenter, verständlicher und für den Kunden leichter kontrollierbar werden. Dazu soll dem Kunden ermöglicht werden, via Webseite des lokalen Anbieters, seinen eigenen Verbrauch abzufragen. Dadurch hat der Kunde erstmals die Möglichkeit seinen eigenen Verbrauch genauer zu analysieren, zu kontrollieren und eventuell anzupassen. Bei einer Kosten-Nutzen-Analyse zum österreichweiten Tausch von Strom- und Gaszählern, wobei für bestehende Kunden der Austausch kostenlos sein soll, wurde errechnet, dass bei einem Austausch im Durchschnitt 3,5% weniger elektrische Energie und 7% weniger Gas verbraucht wird und geringere Kosten durch effizientere Lieferung der Netzbetreiber entstehen. Durch die Reduktion des Verbrauchs würde auch der Ausstoß an $CO_2$ sinken, in der Studie ist, für den Modellzeitraum von 15 Jahren bei Strom und

12 Jahren bei Gas, je nach Modell, eine Einsparung von **4,6 bis 6,1 Millionen Tonnen $CO_2$** allein in Österreich möglich. [Pri10]

Die hier genannte Studie verdeutlicht, wie wichtig es ist, dem Kunden seinen Verbrauch sichtbar zu machen und eventuell Anreize zu bieten, damit dieser Verbrauch gesenkt wird. Hier soll wiederum der $Eco_2$-Manager ansetzen, indem nicht nur Strom- und Gasverbrauch, sondern wesentlich mehr Faktoren erfasst und dargestellt werden sollen. In [Pri10, Seite 41, Abbildung 1] wird ein ähnliches Modell zum Zusammenfassen von Strom- und Gas-Smart-Meter bereits gezeigt, die Datenanbindung erfolgt dann weiters mittels Power Line Communication (PLC) oder Mobilfunk, wie es auch für weitere Ausbaustufen des $Eco_2$-Managers geplant ist.

## 7.2. Anbindung von diversen Smart Metern

Von sehr großer Bedeutung beim Betrieb von **Smart Metern** ist die Kommunikationsschnittstelle, die die Verbindung zwischen dem Smart Meter und anderen Geräten herstellt. Die meisten Smart Meter am europäischen Markt (für Strom, Gas, Wasser,...) verwenden den gemeinsamen Standard **IEC-1107**, welcher eine optische Schnittstelle beschreibt (siehe Abbildung 7.1). Der Standard definiert die physikalischen Eigenschaften des optischen Interfaces und beschreibt das zu verwendende Software-Protokoll.

Die verwendete Hardware wurde sehr simpel gehalten, es gibt fertige Umsetzer auf RS232 oder USB zu kaufen, allerdings kann man diese mit ein paar einfachen Bauteilen auch selbst bauen, wie in [Mes02, Seite 52] beschrieben. Das Softwareprotokoll, mit dem die Daten übertragen werden ist ebenfalls sehr einfach gehalten (ASCII-Steuerzeichen und ASCII codierte Nutzdaten). Die Datenraten für die Übertragung, das Format der einzelnen Zeichen usw. werden im Standard ebenfalls festgelegt. Leider halten sich viele Hersteller nicht exakt an die Vorgaben des Standards und weichen von diesem ab. Inwiefern der Standard eingehalten wurde muss auch meist beim Hersteller erfragt oder selbst durch Versuche herausgefunden werden.

Um eine Anbindung an den $Eco_2$-Manager aufbauen zu können, wurden von **Luna Elektronik Elektrik Sayaclari** in der Türkei zwei Smart Meter (siehe Abbildung 7.1) zur Verfügung gestellt. In den folgenden Kapiteln wird die Kommunikation mit diesen Smart Metern beschrieben.

## 7.3. Anbindung Luna Smart Meter

Die beiden Smart Meter von Luna, die zur Verfügung gestellt wurden, halten sich leider nicht genau an den Standard IEC-1107 zum Auslesen der Daten. Die optische Schnittstelle hält sich von der Hardware daran und kann nach den Vorgaben laut IEC umgesetzt werden, das Softwareprotokoll wurde allerdings angepasst.


Abbildung 7.1.: Foto Luna Smart Meter

### 7.3.1. Softwareprotokoll

#### Schnittstellenparameter

Das von Luna verwendete Protokoll ist in Form einer Grafik mit dem Kommunikationsprotokoll (siehe Anhang B) definiert. Allerdings werden dort keine genauen Angaben zu Einstellungen der seriellen Schnittstelle wie Anzahl Datenbits, Parität, Anzahl Stopbits etc. gegeben, diese mussten eigens erfragt werden. Folgende Einstellungen sind nötig um mit dem Smart Meter kommunizieren zu können:

- **Baudrate:** zu Beginn 300, später wählbar 300/2400/9600 (siehe Anhang B)
- **Anzahl Datenbits:** 7
- **Parität:** Even
- **Anzahl Stopbits:** 1
- **Handshake:** Keiner

### Steuerzeichen

Mit den genannten Einstellungen kann man eine Verbindung zum Smart Meter herstellen. Alle im Protokoll gelisteten Zeichen (auch Zahlen) werden als ASCII-Zeichen gesendet und die Steuerzeichen (ACK, SOH, STX,...) können ebenfalls einer vollständigen ASCII-Tabelle entnommen werden, diese müssen allerdings als Hexadezimal-Zahl angegeben werden beim Senden. Hier eine Liste der verwendeten Steuerzeichen und zugehörigem Hexadezimalwert:

- **0x01:** Start of Heading (SOH)
- **0x02:** Start of Text (STX)
- **0x03:** End of Text (ETX)
- **0x04:** End of Transmission (EOT)
- **0x06:** Acknowledged (ACK)
- **0x0A:** Line Feed (LF)
- **0x0D:** Carriage Return (CR)
- **0x15:** Negative Acknowledged (NAK)

In Folge werden die Steuerzeichen zur Verdeutlichung immer als Tags (`<SOH>, <STX>,...`) geschrieben, damit sich diese von den restlichen Daten abheben.

### Prüfsummenberechnung

Der **Block Check Character (BCC)** stellt einen Sonderfall dar. BCC ist kein Steuerzeichen sondern eine Prüfsumme, die am Ende der Pakete hinzugefügt wird, um Fehler im Datenstrom zu erkennen. Berechnet wird diese Prüfsumme durch bitweise **exklusiv oder (XOR)-Aufaddierung ($\oplus$)** des gesamten Pakets, wobei das Steuerzeichen SOH nicht berücksichtigt wird. Hier ein Beispiel für die Berechnung der Prüfsumme für das Paket `<SOH>R2<STX>0.0.0()<ETX><BCC>` (Paket zum Lesen der Seriennummer):

$$0x52 \oplus 0x32 \oplus 0x02 \oplus 0x30 \oplus 0x2E \oplus 0x30 \oplus 0x2E \oplus 0x30 \oplus 0x28 \oplus 0x29 \oplus 0x03 = 0x50(P)$$

### Verbindungsaufbau

Der Verbindungsaufbau zu einem Luna Smart Meter kann, wie in der Bedienungsanleitung [Lun12] beschrieben, nur erfolgen, wenn **mindestens eine Phase mit Spannung versorgt** wird oder alternativ mit der optischen Schnittstelle, wenn der blaue Knopf gedrückt wird und innerhalb von 15 Sekunden eine Anforderung geschickt wird. Weiters gibt es zwei verschiedene Arten, wie der Aufbau der Verbindung erfolgen kann:

1. Auf die im Dokument angegebene Art und Weise. Hier wird lediglich `/?!<CR><LF>` an das Smart Meter gesendet, welches mit der Seriennummer antwortet.

2. Beim Aufbau der Verbindung kann eine Seriennummer mitgesandt werden. Stimmt die Seriennummer mit der des Gerätes überein, antwortet dieses in der gleichen Art und Weise wie im Protokoll beschrieben mit seiner eigenen Identifikation. Wird eine andere Seriennummer gesendet, die nicht der des Geräts entspricht, reagiert dieses nicht auf die Anforderung.

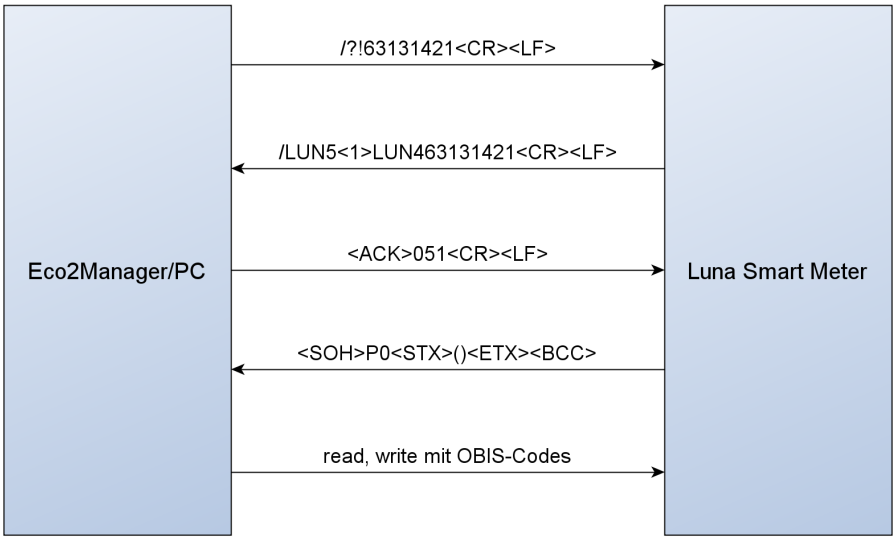



Abbildung 7.2.: Verbindungsaufbau Luna Smart Meter

Abbildung 7.2 zeigt ein Beispiel für einen Verbindungsaufbau zwischen dem $Eco_2$-Manager bzw. einen PC und einem zur Verfügung stehenden Smart Meter mit der Seriennummer **63131421**. Hier eine kurze Beschreibung zum Ablauf:

- Senden des *Requests* inklusive Seriennummer
- Smart Meter Antwortet mit dem Typ (LUN5) und der eigenen Seriennummer
- Der Empfang wird mit einem ACK bestätigt, gefolgt von einer 0 und zwei Optionen:
  - Das erste Byte (Z) steht für die Übertragungsgeschwindigkeit, die in Folge verwendet wird. Mögliche Optionen:

    0 = **300 Baud**

    3 = **2400 Baud**

    5 = **9600 Baud**

    Leider lässt sich die Geschwindigkeit des Verbindungsaufbaus nicht ändern.
  - Das zweite Byte (Y) steht für die Art der folgenden Kommunikation.

    0 = **Data Readout** (das Smart Meter liefert alle Parameter)

    1 = **Programming Mode** (Parameter können mittels OBIS-Codes geschrieben/gelesen werden)
- Das Smart Meter sendet je nach Modus folgende Werte zurück:

- **Data Readout Mode:** Es werden alle Parameter getrennt durch `<CR><LF>` zurück gesendet, wie in der Protokollbeschreibung unter *Data Readout* abgebildet. Allerdings werden hier immer alle Parameter gesendet, was im Fall des $Eco_2$-Managers nicht gewünscht ist.
- **Programming Mode:** Das Smart Meter sendet zur Bestätigung den Befehl *P0* mit leerem Datenwert (und eventuell bereits höherer Baudrate) zurück (siehe Beispiel und Protokollbeschreibung). Ein Nachteil der Baudratenänderung ist, dass der Befehl `P0` schwer zu empfangen ist, da die Baudratenumstellung auf der Seite des Smart Meters sehr schnell stattfindet und man oft nicht in der Lage ist, die eigene Baudrate in dieser Geschwindigkeit zu ändern. Allerdings muss dieser Befehl auch nicht ausgewertet werden und kann daher verworfen werden.
  Im Anschluss an diesen Verbindungsaufbau können Daten mittels OBIS-Codes gelesen und geschrieben werden. Dies wird im nächsten Kapitel genauer erläutert.

Wenn sich das Gerät im **Data-Readout- oder Programming-Mode** befindet, wird im **Display** `Read` angezeigt. In dieser Zeit ist keine manuelle Ablesung am Display möglich und die Übertragung kann auch nicht vom User abgebrochen werden.

### Daten lesen im Programming Mode

Daten können mittels Object Identification System (OBIS) Codes (Nachfolger von Energie Data Identification System (EDIS)) vom Smart Meter angefordert werden oder auch geschrieben werden. Das System für OBIS-Codes ist normalerweise wie in Tabelle 7.1 aufgebaut, wird hier aber nur in der gekürzten Version (zwischen den dicken Linien) verwendet. Dabei wird das Medium und der Kanal weggelassen und es werden nur **Messgröße**, **Messart**, **Tarifstufe** und **Vorwertzählerstand** in der Form `GG.AA.T*VV` angegeben. Die Tabelle mit den verfügbaren Parametern, die vom Luna Smart Meter gelesen werden können, befindet sich in Anhang B. Zudem werden die OBIS-Codes im Display des Smart Meters im unteren Rand eingeblendet und sie können der Bedienungsanleitung [Lun12] entnommen werden. Welche Parameter ausgewertet werden sollen kann individuell entschieden und dementsprechend im $Eco_2$-Manager programmiert werden.

| M | - | KK | : | GG | . | AA | . | T | * | VV |
|---|---|---|---|---|---|---|---|---|---|---|
| Medium | ASCII 0x2D | Kanal | ASCII 0x3A | Mess-größe | ASCII 0x2E | Mess-art | ASCII 0x2E | Tarif-stufe | ASCII 0x2A | Vorwert-zählerstand |

Tabelle 7.1.: Aufbau OBIS Codes

Die Befehle zum Schreiben, Lesen, Ausführen etc. sind in der Beschreibung des Kommunikationsprotokolls genau dargestellt. Der für die Anwendung mit dem $Eco_2$-Manager interessanteste Teil besteht sicher im **Auslesen der Parameter** vom Smart Meter, was auch ohne Passwort funktioniert, allerdings ist hier der Herstellerfirma bei der Dokumentation ein **Fehler unterlaufen**, welcher **rot korrigiert** wurde. Damit das Lesen funktioniert, muss der OBIS-Code gefolgt von `()` geschrieben werden, ansonsten wird das Paket als nicht gültig erkannt und es werden keine Daten zurück geschickt. Das in Tabelle 7.2

abgebildete Paket fordert die Daten für die aktuelle, gesamte Wirkleistung (OBIS-Code 1.8.0) an. Es wird der Aufbau des Pakets in ASCII Zeichen und darunter in Hexadezimaldarstellung gezeigt.

| ASCII | <SOH> | R | 2 | <STX> | 1 | . | 8 | . | 0 | <ETX> | <BCC> |
|---|---|---|---|---|---|---|---|---|---|---|---|
| HEX | 01 | 52 | 32 | 02 | 31 | 2E | 38 | 2E | 30 | 03 | 58 |

Tabelle 7.2.: Beispielpaket zum Anfordern der aktuellen Wirkleistung

Wurden die gewünschten Daten angefordert bzw. geschrieben, kann mittels **Break-Befehl** der Programming Mode verlassen werden. Wird ein Break gesendet, geht das Display wieder in den normalen Anzeigemodus über und es muss zuerst wieder ein Verbindungsaufbau zu diesem Smart Meter erfolgen.

### 7.3.2. Hardwareschnittstelle

Die zur Verfügung stehenden Smart Meter haben unterschiedliche Schnittstellen, über die man die Daten auslesen kann. Das Softwareprotokoll, wie oben beschrieben ist jedoch bei allen Schnittstellen gleich und kann daher für alle Varianten in gleicher Art und Weise eingesetzt werden.

#### optische Schnittstelle

Die Smart Meter von Luna sind alle mit einer optischen Schnittstelle laut IEC-1107 Hardware-Spezifikation ausgestattet. Diese Schnittstelle ermöglicht es, eine galvanisch getrennte Verbindung zum Smart Meter herzustellen. Dabei wird zum Senden jeweils eine Infrarot-LED und zum Empfangen ein Fototransistor verwendet. Die Übertragung findet durch Modulation des Infrarotsignals statt, wobei die LED bei einem High-Pegel eingeschalten und bei einem Low-Pegel ausgeschalten ist.


Abbildung 7.3.: Verwendung der optischen Schnittstelle und Lesegerät von unten

Eine solche Schnittstelle kann mit wenigen Bauteilen für eine RS232-Verbindung zum PC vorbereitet werden, meist sind die Lesegeräte heute aber mit einem USB-Anschluss

versehen. Abbildung 7.3 zeigt das Lesegerät von Luna auf einem Smart Meter. Die Ausrichtung ist in diesem Fall wichtig, da die optischen Elemente aufeinander ausgerichtet sein müssen. Dieses Lesegerät enthält einen IC von Silicon Laboratories (SiLabs) (CP210x USB to Serial Bridge) welcher das USB-Signal auf das Protokoll von RS232 umsetzt. Die weitere Hardware wurde in das Gehäuse integriert, wodurch das Lesegerät sehr handlich verwendet werden.

- **Vorteile optisches Interface:**
  - galvanische Trennung, Netzspannung weit entfernt
  - einfache Hardware
  - schnelle Installation (Magnetbefestigung)
  - kein Öffnen des Klemmenblocks erforderlich
  - steht bei sehr vielen Smart Metern zur Verfügung
- **Nachteile optisches Interface:**
  - nur ein Gerät anschließbar (kein Bus)
  - Lesegerät kann leicht unabsichtlich entfernt werden
  - genaue Ausrichtung am Gerät erforderlich

### RS232-Schnittstelle

Zusätzlich zur optischen Schnittstelle besitzen viele Luna Smart Meter eine RS232-Schnittstelle. Diese befindet sich beim Klemmenblock zwischen der Klemme für den Ausgang der Phase 3 und der Klemme für den Neutralleiter, wobei hier drei Anschlüsse zur Verfügung stehen:

- **R-A**: RX-Anschluss, über diesen Anschluss werden die Daten empfangen
- **T-B**: TX-Anschluss, über diesen Anschluss werden die Daten gesendet
- **GND**: gemeinsames Nullpotential für RX und TX

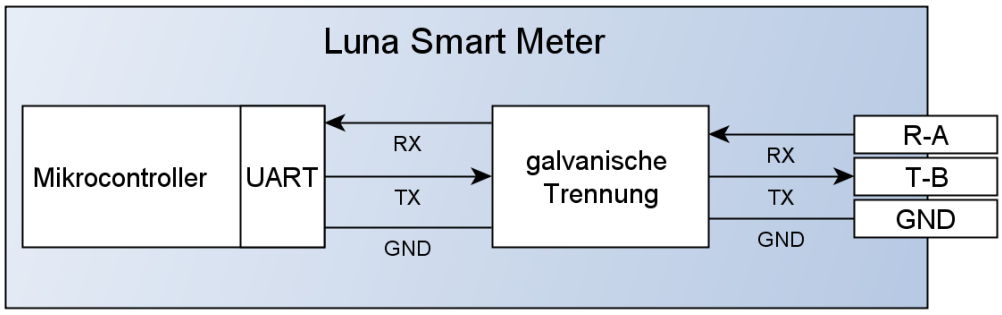



Abbildung 7.4.: Blockschaltbild interner Aufbau Luna Smart Meter mit RS232

Der Anschluss kann direkt an eine RS232-Schnittstelle eines PC/Embedded Systems erfolgen oder kann mittels RS232 zu USB Umsetzer an den USB Port angeschlossen werden. Im Zuge dieser Arbeit wurde die Variante mit dem USB-Umsetzer gewählt, wobei ein Future Technology Devices International Ltd. (FTDI) FT232R Umsetzer zum Einsatz kam und getestet wurde. Laut Hersteller ist diese Schnittstelle intern auch galvanisch getrennt, wie in Abbildung 7.4 dargestellt. Der in der Abbildung gezeigte innere Aufbau wurde den Infos eines Luna Technikers entnommen. Eine Übertragung über RS232 ist zudem nur möglich, wenn mindestens eine Phase an das Stromnetz angeschlossen wurde.

- **Vorteile RS232:**
  - weit verbreitete Schnittstelle
  - auf Embedded Systems/Mikrocontrollern vorhanden (als UART oder USART bezeichnet)
  - günstige Schnittstellenwandler erhältlich (TTL-Pegel/USB...)
- **Nachteile RS232:**
  - Anschlüsse nahe an der Netzspannung
  - kein Anschluss ohne Öffnen des Klemmenblocks möglich
  - drei Anschlussleitungen benötigt
  - keine Übertragung ohne Netzspannung
  - empfindlich gegenüber Gleichtaktstörungen wegen asymmetrischer Übertragung

#### RS485-Schnittstelle

Als Alternative zur RS232 werden die Smart Meter auch mit einer RS485-Schnittstelle ausgestattet. Die Anschlüsse dafür befinden sich ebenfalls beim Klemmenblock zwischen Ausgang Phase 3 und Neutralleiter, es werden aber diesmal nur zwei Anschlüsse benötigt, da hier eine differentielle Übertragung im Halbduplex-Modus stattfindet. Es wird also immer nur abwechseln gesendet oder empfangen.

- **R-A**: auch oft als + oder positiver Anschluss bezeichnet
- **T-B**: auch oft als - oder negativer Anschluss bezeichnet

Der große Vorteil bei RS485 liegt in der geringen Störanfälligkeit durch die differentielle Datenübertragung und dementsprechend können lange Datenleitungen verwendet werden. Außerdem kann RS485 als Bus mit bis zu 32 Teilnehmern ausgelegt werden, wobei die Busleitung bis zu 500m lang sein kann und zu den Teilnehmern wird jeweils eine maximal 5m lange Stichleitung geführt. Die Unterscheidung in High und Low geschieht durch die Differenz der beiden Adern. Ist $A-B < 0,3V$ wird dies als **Mark/logisch High** gewertet, bei $A - B > 0,3V$ wird ein **Space/logisch Low** erkannt.

Der Vorteil des Busses kann für den Zusammenschluss mehrerer Smart Meter genutzt werden, die auf einmal ausgelesen werden sollen, dies ist mit den anderen erwähnten Technologien nicht möglich. Wie ein solches Netzwerk mit mehreren Smart Metern aussehen kann wird in Abbildung 7.6 gezeigt. Die 120Ω Widerstände werden zur Vermeidung von

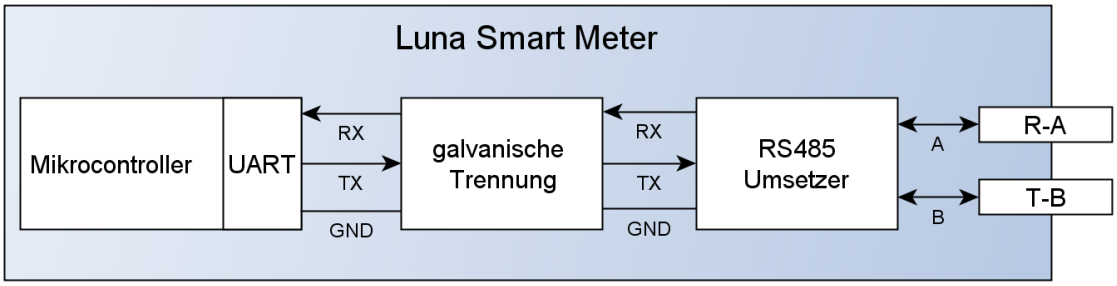



Abbildung 7.5.: Blockschaltbild interner Aufbau Luna Smart Meter mit RS485

Reflexionen zum Bus hinzugefügt, bei kurzen Leitungslängen können diese entfallen. Sollten Probleme im Ruhezustand des Busses auftreten, sollte zur Leitung A ein Pullup und zur Leitung B ein Pulldown Widerstand (560Ω bis 1$k\Omega$) hinzugefügt werden. [Axe99] RS485 stellt auch einen sehr häufig verwendeten Industriestandard dar, als Protokoll für die Übertragung wird ebenfalls die Spezifikation von Luna (siehe Unterabschnitt 7.3.1) verwendet.

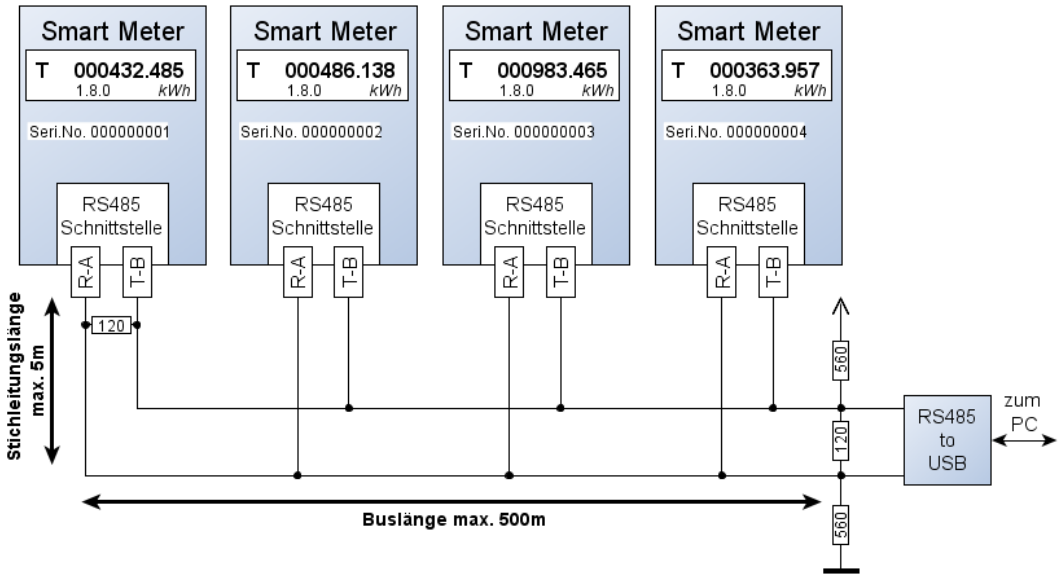



Abbildung 7.6.: Buskonfiguration RS485

Der RS485-Standard sieht zwar standardmäßig keine galvanische Trennung vor, in den Luna Metern wird aber eine galvanische Trennung zwischen der Auswertungselektronik und den Datenleitungen realisiert. Die ISO 9549 Norm schreibt diese auch vor, allerdings müsste für diese Norm die Trennung nach der Umsetzung auf RS485 geschehen, laut Luna geschieht dies bei den Smart Metern allerdings vorher (siehe Abbildung 7.5. Bei großen Leitungslängen können allerdings große Potentialverschiebungen auftreten, die den Empfangsbaustein beschädigen könnten.

Umgesetzt wurde die Verbindung mit einem von Luna entwickelten USB zu RS485 Umsetzer, wobei wieder ein IC von SiLabs (CP210x USB to Serial Bridge) zum Einsatz kommt, welcher auch beim optischen Umsetzer verwendet wurde. Da das gleiche Softwareprotokoll verwendet wird, kann hier die gleiche Implementation verwendet werden wie für das optische Interface. Alternativ zu diesem Baustein wurde ein RS232 zu RS485 Umsetzer von **ABUS** verwendet (ABUS TV8469). Dieser wurde am vorhandenen Smart Meter getestet, es konnte allerdings keine Verbindung hergestellt werden. Nach Betrachtung des Busses mit einem Oszilloskop konnte festgestellt werden, dass das Senden an das Smart Meter funktioniert, aber die Daten die zurück gesendet werden nicht umgesetzt werden, da der Umsetzer zu langsam zwischen Senden und Empfangen umschalten dürfte. Solche günstigen Umsetzer können also in Kombination mit den Luna Smart Metern nicht verwendet werden.

- **Vorteile RS485:**
  - kann als Bus ausgeführt werden (bis zu 32 Geräte) - siehe Abbildung 7.6
  - nur ein Adernpaar benötigt
  - lange Busleitungen möglich
  - unempfindlich gegenüber Gleichtaktstörungen
  - günstige Bustreiberbausteine erhältlich
- **Nachteile RS485:**
  - bei Mikrocontrollern leider wenig verbreitet
  - Anschlussklemmen nahe zur Netzspannung
  - günstige RS232 zu RS485 Umsetzer funktionieren nicht

### 7.3.3. Umsetzung der Anbindung mit ChipworkX

Alle drei oben genannten Schnittstellen wurden testweise an das Embedded System angebunden. Mittels Mikrocontroller können die Smart Meter über die vorhandenen Kommunikationswege angebunden werden (Ethernet und XBee), es wurde aber zusätzlich die Möglichkeit geschaffen, die Luna Smart Meter auch über USB an den $Eco_2$-Manager anschließen zu können. Damit wird auch ein RS485 Bus oder die Verwendung des optischen Interfaces ermöglicht, wofür die jeweiligen Umsetzer von Luna bereitgestellt wurden. Da die beiden Luna-Umsetzer (optisch und RS485) beide mit einem SiLabs-IC arbeiten, müssen die Treiber für zwei unterschiedliche Bausteine verwendet werden - für SiLabs und FTDI Serial Devices.

Das .NETMF bietet bereits eine Sammlung von Treiber für USB zu seriellen Umsetzern, unter anderem für die beiden verwendeten ICs. Zu finden sind diese Treiber in den GHI Libraries im Namespace **GHIElectronics.NETMF.USBHost**. Die Klasse **USBH_Device** stellt die Überklasse aller Geräte dar, die per USB verbunden werden können, die Klasse **USBH_SerialUSB** ist speziell auf Umsetzer von USB auf eine serielle Verbindung spezialisiert. Von dieser werden Geräte von **FTDI, SiLabs, Profilic und CDC** unterstützt, wobei nur die Funktionalität für die vorhandenen Geräte implementiert

wurden. Profilic und CDC Umsetzer waren leider keine vorhanden, um eine Realisierung dieser testen zu können.

#### DeviceConnectedEvent und DeviceDisconnectedEvent

Diese beiden Events werden automatisch vom Framework aufgerufen, wenn ein USB-Gerät angesteckt oder entfernt wird. Es können beliebig viele Eventhandler zu den Methoden hinzugefügt werden, es sollten allerdings einige Dinge beachtet werden:

- Da bereits im **UsbSdController** je ein Handler zu den Events zugewiesen wurde, muss aufgepasst werden, dass sich diese nicht gegenseitig beeinflussen. Da aber die Handler im **UsbSdController** nur Massenspeichermedien behandeln, können in der Klasse **UsbSmartMeter** ohne Probleme weitere Handler für serielle Geräte hinzugefügt werden. Dies geschieht im Konstruktor und es werden die Methoden *addSerialUsbDevice* und *removeSerialUsbDevice* aufgerufen.

- Wird das Embedded System bereits mit eingesteckten USB-Geräten gestartet, wird jeweils **nur der erste zugewiesene Eventhandler** aufgerufen. Da der **UsbSdController** vorher instanziert wird, wurde im Konstruktor von **UsbSmartMeter** zusätzlich eine Schleife eingebaut, die für alle bereits angeschlossenen Geräte den Handler aufruft. Damit wird dieses Problem umgangen.

Im **addSerialUsbDevice**-Handler wird überprüft, ob es sich beim neu gefundenen Gerät um einen Umsetzer von USB auf eine serielle Verbindung handelt, beachtet werden dabei derzeit Produkte der Hersteller FTDI und und SiLabs. Die Behandlung der Geräte der verschiedenen Hersteller unterscheidet sich leicht:

- Bei **FTDI-Geräten** kann über den **USBH_DeviceType** abgefragt werden, ob es sich um ein entsprechendes Produkt handelt und anschließend kann ein neues **USBH_SerialUSBparity** Objekt mit der entsprechenden Baudrate, Parity und der richtigen Anzahl an Datenbits und Stopbits erstellt werden. Das Gerät wird im Anschluss gleich noch geöffnet und der ArrayList *serialUSB_* hinzugefügt. Diese ArrayList enthält alle aktuell gefundenen und geöffneten Geräte.

- Wird ein **SiLabs-Gerät** angeschlossen kann dies oft nicht richtig erkannt werden. Viele dieser Geräte werden als DeviceType *Unknown* erkannt, daher wird eine zusätzliche Abfrage für die verwendeten ICs nach VendorID (4292 für SiLabs - identifiziert den Hersteller) und ProductID (60000 für den IC CP210x) eingeführt. Sollten andere Umsetzer von SiLabs verwendet werden, muss diese Abfrage eventuell erweitert werden.
  Wurde ein Gerät als SiLabs-Produkt identifiziert, kann dieses in ein solches umgewandelt werden. Dazu wird ein neues **USBH_Device** mit den Angaben aus dem an den Handler übergebenen Device erstellt, allerdings wird der **DeviceType** auf **UBSH_DeviceType.Serial_SiLabs** gesetzt. Mit diesem Gerät wird im Anschluss ein neues Objekt von **USBH_SerialUSBparity** erstellt, der Port geöffnet und wiederum der ArrayList *serialUSB_* hinzugefügt.

Zusätzlich wird bei jedem neu hinzugefügten Gerät ein Signaling für den **getSerialUsbDataThread** durchgeführt, da dieser auf *waiting* gesetzt wird, wenn keine Geräte vorhanden sind.

### Klasse USBH_SerialUSBparity

Normalerweise kann für USB-Seriell-Umsetzer die Klasse **USBH_SerialUSB** verwendet werden. Diese Klasse enthält alle notwendigen Methoden um mit den eingestellten Werten Daten senden und empfangen zu können, allerdings gibt es mit SiLabs-Devices einige Probleme, wobei bereits bei GHI Electronics dokumentiert ist, dass **kein Handshake unterstützt** wird und die **Einstellungen nach Erstellen des Objekts nicht mehr verändert** werden können.

Es wurde nicht dokumentiert, dass es zu Problemen beim **Hinzufügen des Paritätsbits** gibt. Bei den Einstellungen 7E1 (7 Datenbits, gerade Parität, 1 Stopbit) wird, wie nach Untersuchungen mit einem digitalen Speicheroszilloskop festgestellt wurde, das Paritätsbit falsch gesetzt. Anstatt je nach Anzahl der Einsen in den Daten ein entsprechendes Paritätsbit hinzuzufügen, wurde immer eine Null (Space) hinzugefügt, was nicht den Einstellungen entspricht. Tabelle 7.3 zeigt die Zeichen die für einen Verbindungsaufbau gesendet wurden und die entsprechenden, aus dem Bild des Oszilloskops herausgelesenen, Bits. Wie man erkennen kann, wird als Paritätsbit immer eine Null angefügt.

| ASCII | gesendete Bits | korrekte Bits |
|---|---|---|
| / | 0 1111010 0 1 | 0 1111010 1 1 |
| ? | 0 1111110 0 1 | 0 1111110 0 1 |
| ! | 0 1000010 0 1 | 0 1000010 0 1 |
| 6 | 0 0110110 0 1 | 0 0110110 0 1 |
| 3 | 0 1100110 0 1 | 0 1100110 0 1 |
| 1 | 0 1000110 0 1 | 0 1000110 1 1 |
| 3 | 0 1100110 0 1 | 0 1100110 0 1 |

Tabelle 7.3.: mit SiLabs-Treiber gesendete Daten mit falscher Parität

Um dieses Problem zu beheben wurde die Klasse **USBH_SerialUSBparity** geschrieben. Diese dient als Wrapper, leitet also alles an die Klasse **USBH_SerialUSB** weiter, aber wenn sieben Datenbits und ein Paritätsbit verwendet werden, wird die Parität in der Software berechnet und als acht Datenbits ohne Parität verschickt. Das **Paritätsbit** wird also als **achtes Datenbit** in den Datenstrom eingefügt, was den gleichen Effekt hat, wie es an sieben Datenbits anzuhängen. Bei den verwendeten Datenraten sollten auch keine Probleme mit der Geschwindigkeit der Berechnung auftreten.

Die Methode *addParityBit* in der geschriebenen Klasse fügt an der Stelle des achten Datenbits je nach gewünschter Parität das entsprechende Bit ein. Dabei kann die gerade Parität (Even Parity) berechnet werden, indem zu einer Null alle Bits der Reihe nach XOR verknüpft werden. Für eine ungerade Parität startet man mit einer Eins.

Beim Empfangen von Daten wird das Paritätsbit derzeit ignoriert, eine Alternative wäre das Verwerfen der Daten, wenn das Paritätsbit falsch gesetzt ist.

### Klasse UsbSmartMeter

Um die Daten von den Smart Metern zu lesen wird im Konstruktor dieser Klasse ein eigener Thread (**getSerialUsbDataThread**) gestartet. Da die elektronischen Zähler von sich aus keine Daten an den $Eco_2$-Manager schicken, müssen diese in einem vorgegebenen Intervall dort abgefragt werden. Da normalerweise ein Lastgang alle 15 Minuten abgefragt

wird, wurde dieses Intervall auch für die Datenabfrage übernommen, allerdings kann diese Zeitspanne in den **IOConstants** mit der Variable *REPEAT_TIME_GET_DATA* in Sekunden eingestellt werden.

In diesem Thread wird die Datenbank mit den Sensoreinstellungen abgefragt und es werden alle Einträge, die eine 8-stellige, als Integer interpretierbare Zahl als *SensorID* und einen OBIS-Code als *ValueID* haben, in eine HashMap geschrieben. Anschließend werden alle derzeit vorhandenen SerialUsb-Devices durchlaufen und es wird jeweils versucht, über jede Schnittstelle die entsprechende Seriennummer aus der HashMap zu erreichen. Dies wird für alle Seriennummern wiederholt, reagiert ein Smart Meter auf die Anfrage mit seiner Seriennummer, werden die Daten zu den entsprechenden OBIS-Codes abgefragt. Dazu wird zuerst mit der Methode *enterProgrammingMode* in den Programming Mode gewechselt und anschließend die Baudrate gewechselt. Leider ist es nicht möglich, die Baudrate durch simples Setzen des Properties zu wechseln, dazu muss das **USBH_SerialUSBparity** Objekt gewechselt werden. Es wird ein neues Objekt mit neuer Baudrate in die ArrayList geschrieben. Jetzt werden alle OBIS-Codes der Reihe nach ausgelesen, nach dem Ende der Kommunikation wird ein Break-Befehl gesendet und wieder auf die alte Baudrate zurück gewechselt.

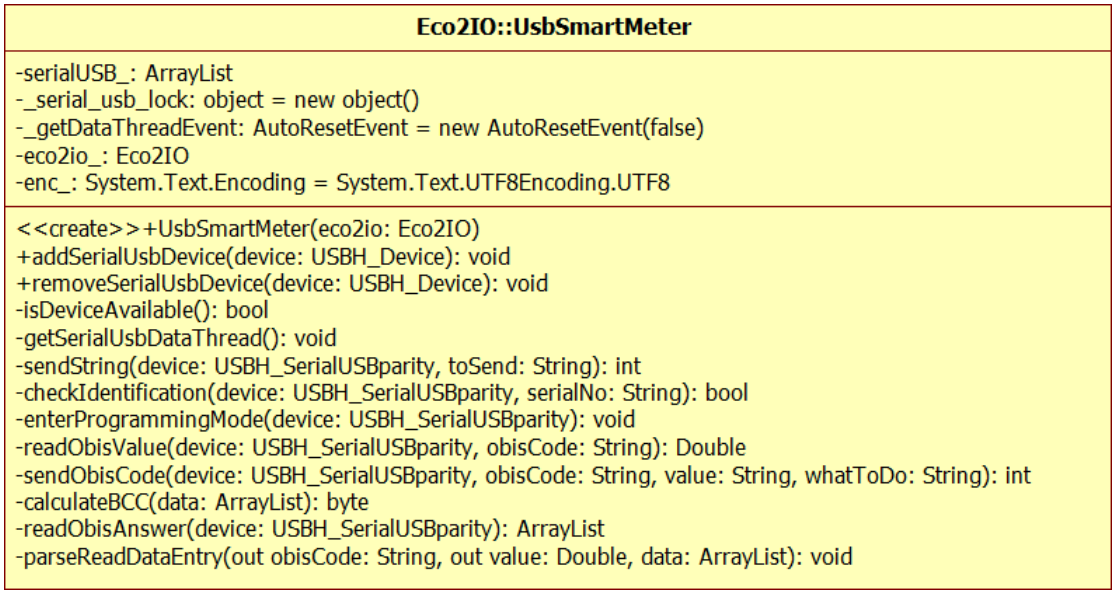



Abbildung 7.7.: Klassendiagramm UsbSmartMeter

Die einzelnen Funktionen für die Kommunikation (siehe Abbildung 7.7) werden in Folge beschrieben:

- **sendString:** Sendet einen String über das angegebene SerialUsb-Device.
- **checkIdentification** versucht eine Verbindung zu einem Smart Meter mit der angegebenen Seriennummer herzustellen. Dazu wird die Anforderung `/?!12345678<CR><LF>` am angegebenen Port gesendet, wobei als 8-stellige Zahl die Seriennummer eingefügt

wird. Da bei 300 Baud ein Zeichen 33ms zum Senden benötigt und 13 Zeichen gesendet werden, wird im Anschluss eine halbe Sekunde gewartet und dann die Antwort gelesen. Kommt eine Antwort mit der gleichen Seriennummer zurück, gibt die Funktion **true** zurück, kommt keine oder eine falsche Seriennummer zurück, wird **false** zurückgegeben. Damit kann überprüft werden, ob ein Smart Meter mit einer bestimmten Seriennummer angeschlossen ist oder nicht.

- **enterProgrammingMode** sendet `<ACK>051<CR><LF>` über den angegebenen Port. Dies ist die Bestätigung, dass eine Kommunikation mit diesem Gerät gewünscht ist und zwar mit **Baudrate 9600** und im **Programming Mode**, damit die Parameter individuell mit OBIS-Codes ausgelesen werden können. Die Antwort vom Smart Meter wird hier nicht überprüft, da eine Baudratenänderung stattfindet und diese zu langsam ist, um die Antwort komplett empfangen zu können.

- **sendObisCode** sendet einen OBIS-Code, wobei der Code selbst, der zu sendende Wert und der Befehl der auszuführen ist (R2, W2,...) angegeben werden kann. In der Methode wird eine ArrayList angelegt, in diese werden die zu sendenden Zeichen geschrieben. Der am häufigsten verwendete Befehl ist der **Read-Befehl**, welcher zum Beispiel für die Wirkleistung (1.8.0) in folgender Form zusammengestellt wird: `<SOH>R2<STX>1.8.0()<ETX><BCC>`. Die anderen Befehle weichen nur geringfügig davon ab und können alle mit dieser Methode versendet werden.

- **readObisValue** benutzt die Funktion **sendObisCode** um einen Parameter anzufordern. Es wird der Read-Befehl ausgeführt für den gewünschten Wert. Nachdem die Anforderung gesendet wurde wird die Methode **parseReadDataEntry** aufgerufen, welche die angeforderten Daten einliest und als Double zurück gibt. Zudem wird der gelesene OBIS-Code zur Kontrolle zurückgegeben.

- **readObisAnswer** liest eine komplette Zeile vom angegeben Port ein. Dabei wird gleich das `<SOH> oder <STX>` zu Beginn abgeschnitten und die Blockchecksumme mit der Methode **calculateBCC** ausgewertet. Zurückgegeben wird die gelesene Zeile in einer ArrayList.

- **parseReadDataEntry** liest aus einer übergebenen ArrayList (aus der Funktion **readObisAnswer**) die Daten die darin enthalten sind. Dazu wird die Position der Klammern gesucht, zwischen denen der Wert steht. Anschließend wird der OBIS-Code von Beginn der Daten bis zur Klammer eingelesen. Zwischen den Klammern wird noch nach dem `*` gesucht, der die Einheit vom Wert abgrenzt und nur der Wert eingelesen. Ein Beispiel, wie ein OBIS-Wert an das Embedded System gesendet wird sieht so aus: `<STX>1.8.0(000123.340*kWh)<ETX><BCC>`, zurückgeliefert wird der OBIS-Code (1.8.0) als String und der Wert (123,34) als Double, da derzeit nur Double-Werte verarbeitet werden und relevant für den $CO_2$-Ausstoß sind.

- **calculateBCC** berechnet die Checksumme der übergebenen Daten. Dazu werden alle Zeichen (mit Ausnahme von `<SOH>`) XOR verknüpft. Der resultierende Wert wird als Byte zurückgegeben.

Leider kann der Verbindungsaufbau zu den Smart Metern von Luna nur mit 300 Baud erfolgen, durch dies wird die Kommunikation beim Aufbau langsam. Sobald die Verbin-

dung aufgebaut wurde, können mit 9600 Baud sehr rasch alle gewünschten Daten gelesen werden. Da allerdings das Lesen in einem eigenen Thread ausgeführt wird, ist die Dauer nicht von besonderer Bedeutung, da alles im Hintergrund abläuft. Mehrere Werte von einem Smart Meter zu lesen funktioniert auch wesentlich schneller, als je einen Wert von mehreren Smart Metern zu lesen, da hier der lange Verbindungsaufbau (bedingt durch die niedrige Datenrate) jedes Mal durchgeführt werden muss.

# Kapitel 8.

# Codingstandard und Debugging-Hilfen

## 8.1. Codingstandard

Der im Projekt verwendete Codingstandard soll nicht die genaue genaue Formatierung vorgeben, sondern hauptsächlich bei den unten aufgelisteten Dingen behilflich sein. Er sollte sowohl für *C#* als auch für *Java* verwendet werden.

- Fehler vermeiden, wie zum Beispiel die Verwechslung von lokalen Variablen und Membervariablen
- Weiterentwicklung und Wartung vereinfachen durch besseren Überblick
- Einheitlichkeit schaffen in allen Klassen
- Lesbarkeit und Wartbarkeit erhöhen

### 8.1.1. Namenskonventionen

Durch die Namenskonventionen soll erreicht werden, dass verschiedene Konstrukte durch die Schreibweise unterschieden werden können.

- **Klassen und Namespaces**: Alle Wörter mit großem Anfangsbuchstaben. (`IchBinEineKlasse`)
- **Instanzmember-Variablen**: Erstes Wort klein, der Rest mit großen Anfangsbuchstaben und einem Unterstrich am Ende. (`IchBinEineInstanzmemberVariable_`)
- **Funktionen**: Das erste Wort klein und der Rest mit großen Anfangsbuchstaben. (`ichBinEineFunktion()`)
- **lokale Variablen**: Das erste Wort klein und der Rest mit großen Anfangsbuchstaben - sie können von Funktionen durch fehlende Klammerung am Ende unterschieden werden. (`ichBinEineLokaleVariable`)
- **Konstanten**: Alle Zeichen groß, die Wörter werden durch Unterstriche getrennt. (`ICH_BIN_EINE_KONSTANTE`)
- **Exceptions**: Gleich wie Klassen, aber zum Schluss das Wort *Exception*. (`IchBinEineException`)
- **Interfaces**: Gleich wie Klassen, aber ein *I* vorangestellt. (`IEinInterface`)

- **abstrakte Klassen**: Gleich wie normale Klassen, aber das Wort *Abstract* vorangestellt. (`AbstractClassName`)

### 8.1.2. Kommentare

Kommentare sollen entweder in der gleichen Zeile im Anschluss angegeben werden oder in der Zeile davor. Wird eine Bemerkung zu einem Block gemacht, erfolgt dies nach der schließenden Klammer. Jede Datei sollte außerdem einen Header aus Kommentaren enthalten, wobei mindestens der Autor angegeben werden sollte.

Die Dokumentationsmöglichkeiten von C# und Java sollten genutzt werden. Die unten stehenden Blöcke sind jeweils vor einer Methode einzufügen. Dies erleichtert die Entwicklung, da beim Verwenden der Funktion in einem Integrated Development Environment (IDE) die Beschreibung automatisch angezeigt wird. Die unten stehenden Beispiele stellen die Minimaldokumentation dar.

```
{JAVADOC}
/**
* kurze Beschreibung der Funktion
* @param parameter1 Beschreibung des Parameters
* @return Beschreibung des Returnwertes
*/
```

```
{C# XML Dokumentation}
/// <summary>
/// kurze Beschreibung der Funktion
/// </summary>
/// <param name="parameter1">Beschreibung des Parameters</param>
/// <returns>Beschreibung des Returnwertes</returns>
```

### 8.1.3. Sonstige Richtlinien

- Alle Blöcke, auch wenn sie nur eine Zeile Inhalt haben, sollen geklammert werden.
- Für Konstanten sollte eine eigene Klasse zur Verfügung stehen (z.B. **Constants**), um alle Einstellungen etc. in einer Klasse vornehmen zu können.
- Properties in C# werden in einen eigenen Block geschrieben (`#region PROPERTIES .... #endregion`)
- `using`-Statements in C# bzw. `import`-Statements in Java sollen immer an oberster Stelle in der Klassendatei stehen (nach dem Kommentar mit dem Autor)
- Für jede Klasse soll eine eigene Datei angelegt werden.

## 8.2. Debugging-Hilfen

### 8.2.1. Debugging in C#

#### ChipworkX Debugging Interface

Das ChipworkX-Development-System bietet verschiedene Interfaces für Debugging, welche in der Firmware implementiert sind. Es kann zwischen Universal Serial Bus (USB), seriellem Port und Ethernet gewählt werden, wobei USB wegen der Fähigkeit das Board gleichzeitig mit Strom zu versorgen bevorzugt wird und auch als Standard beim System eingestellt ist. Für die Verwendung von anderen Debug-Interfaces ist eine eventuelle Umkonfiguration der Firmware nötig, welches mittels verschiedener Klassen von *GHI Electronics* erledigt werden kann. Sollte das Debugging nicht mehr funktionieren, kann mit halten des *Center-* und *Down-Buttons* beim Starten, das Debugging-Interface auf USB umgestellt werden. Sollte das Debugging dann auch noch nicht funktionieren, kann mit Hilfe der Software *ChipworkXUpdater* der Bootloader (*TinyBooter*) und die Firmware erneuert werden. [GHI10b]

Das Debugging selbst wird im *Microsoft Visual C#* erledigt. Dort können, wie sonst von anderen IDEs bekannt, Breakpoints gesetzt werden oder auch der Code im Single-Stepping durchgegangen werden. Es stehen also die standardmäßigen Debugging-Funktionen zur Verfügung, wobei dieses zum Teil ziemlich langsam ist, da viele Code-Teile mit einer Ausgabe übersprungen werden. Dies bremst den Debugger manchmal ziemlich ein.

#### Debugging Klassen

Im .NETMF stehen für ein Debugging nicht viele Methoden zur Verfügung, einzig die Klasse **Debug** wird vom Framework zur Verfügung gestellt. Mit Hilfe dieser Klasse kann die Garbage Collection (GC) gestartet werden und eine Debugging-Ausgabe erstellt werden. Da die Ausgaben aus mehreren Klassen nicht unterscheidbar sind und dies ziemlich unübersichtlich wird, wurde die Klasse **DebugHelper** geschrieben. Diese enthält drei Methoden (*info, warn und error*), welche jeweils zwei Argumente annehmen. Einmal die Nachricht, die ausgegeben werden soll und einmal den Namen der aufrufenden Klasse, welcher über *this.GetType().Name* in jeder Klasse bestimmt werden kann. Die Ausgabe die erzeugt wird, hat die Form: `ERROR/WARN/INFO (ClassName): message` Mit Hilfe dieser Ausgabe, kann man die Klassen unterscheiden, von denen der Output kommt und die Priorität bzw. den Typ der Ausgabe erkennen.

### 8.2.2. Debugging in Java

Für Java wurde *NetBeans* als IDE verwendet. Dieses bietet auch alle Standardfunktionen für Debugging. Für die Debugging-Ausgaben wird der Logger **log4j** verwendet, da dieser eine gute Möglichkeit zur Konfiguration bietet. Der Debugger kann über die Datei **log4j.xml** konfiguriert werden, wobei für dieses Projekt eine Ausgabe auf die Konsole und eine Ausgabe in eine Datei gewählt wurde.

Es stehen für ein Logging diverse Funktionen wie *error, warn, info, trace,...* zur Verfügung, welche sich in der Ausgabe unterscheiden und zusätzlich nach Debugging-Level gefiltert werden können. Je nach Einstellung werden alle Infos, Warnings, Fehler ausgeben oder nur zum Beispiel Fehler. Für die Ausgabe auf der Konsole wurde als Level *DEBUG*

gewählt und für die externe Datei der noch niedrigere Level *TRACE* um alle Messages dort zu erhalten. Zum Betrachten dieser Log-Datei kann das Programm **BareTail** (`http://www.baremetalsoft.com/`) verwendet werden und auf die Schlüsselwörter ERROR, DEBUG... kann ein Farbfilter gelegt werden. Damit ist ein übersichtliches Logging möglich.

Format einer Logging-Zeile (kann in XML-Datei definiert werden):
`HH:MM:SS INFO/WARN/ERROR/DEBUG/TRACE KlassenName:Zeilennummer - Nachricht`

### 8.2.3. sonstige Debugginginghilfen

#### SQLite

Um die SQL-Statements, die am Embedded System verwendet werden, im Vorhinein überprüfen zu können, Tabellen mit einer grafischen Oberfläche zu erstellen und zu bearbeiten und Daten in einer SQLite-Datenbank anzusehen, wird das Programm **SQLite Database Browser Version 2.0b1** (`http://sqlitebrowser.sourceforge.net/`) verwendet. Dieses bietet eine einfache Oberfläche um die vorhin genannten Aktionen auszuführen und auch bereits im Vorhinein auf der SQLite-Datenbank am USB-Stick oder der SD-Karte zu testen. Damit können die Statements zwar getestet werden, ob der komplette SQLite-Standard in .NETMF implemtiert ist wird allerdings nirgends garantiert.

#### Ethernet-Ports

Um Verbindungen vom PC zum ChipworkX-Development-System zu debuggen, speziell auf PC-Seite, kann das Programm **CurrPorts** (`http://www.nirsoft.net/utils/cports.html`) verwendet werden. Dieses zeigt die derzeit aktiven Verbindungen mit allen nötigen Informationen wie Protokoll, Portnummer, IP-Adressen, Status etc an. Speziell der Status des Ports kann für das Debugging von Interesse sein. Die Ports können zusätzlich mit Hilfe der Software wieder geschlossen werden, einzige Voraussetzung für diese Funktion sind Administratorrechte für die Anwendung.

Der genaue Ablauf eines Verbindungsaufbaus, die möglich Port-Stati etc. können [Bal07] entnommen werden.

Um TCP- oder UDP-Pakete genau zu analysieren kann **WireShark** verwendet werden. Dieses Programm zeichnet sämtlichen Traffic an der Ethernet-Schnittstelle des PCs auf, wobei für das Debugging der Pakete zwischen anderen Geräten ein Hub (kein Switch) verwendet werden muss, da die Pakete sonst nicht an den PC gesendet werden. Der gesamte Traffic kann nach verschiedenen Dingen gefiltert werden und die Daten können inklusive Header genau analysiert werden. Diese Art des Debuggings wurde vor allem für die Übertragung der Sensorwerte auf den PC verwendet, um die Geschwindigkeit zu optimieren (siehe Unterabschnitt 5.4.4).

### 8.2.4. Versionen der verwendeten Software

Da die für diese Masterarbeit geschriebene Software voraussichtlich von anderen Personen weiterentwickelt wird, werden in Tabelle 8.1 die genauen Programmversionen angegeben, um Kompatibilitätsprobleme nach Möglichkeit zu vermeiden.

| **Name der verwendeten Software** | **Versionsnummer** |
|---|---|
| Microsoft Visual C# 2010 Express | 10.0.30319.1 |
| Microsoft .Net Micro Framework | 4.1.2821.0 |
| GHI Electronics NETMF SDK for NETMF v4.1 | 1.0.11 |
| Firmware ChipworkX-Module | 4.1.3.0 |
| Bootloader ChipworkX-Module (TinyBooter) | 4.1.3.0 |
| NetBeans IDE | 7.0.1 |
| Java Development Kit Standard Edition | 1.7.0 |
| JCalendarbutton Library (Apache Maven Bundle) | 1.4.5 |
| SQLiteJDBC Library | 0.5.6 |
| JFreeChart Library | 1.0.13 |
| SQLite Database Browser | 2.0b1 |
| CurrPorts | 1.97 |
| BareTail | 3.50a |
| StarUML (OpenSource UML Platform) | 5.0.2.1570 |
| Betriebssystem | Windows 7 Professional SP1 64 Bit |

Tabelle 8.1.: verwendete Softwareversionen

# Kapitel 9.

# Schlußbemerkung und Ausblick

Das hier vorliegende Projekt soll als Basis für die weitere Entwicklung eines $Eco_2$-Managers dienen. Leider konnten noch nicht alle Ideen umgesetzt werden und es gibt sicher einige Optimierungen, die noch durchzuführen sind. Das Thema Hardware wurde nur sehr kurz gestreift, allerdings finden sich in den letzten Jahren immer mehr Plattformen und Techniken, wie Pakete für grafische Oberflächen (zum Beispiel *.NET Clix* von Skewworks - `http://www.skewworks.com`) für das .NET Micro Framework, am Markt. Auch eine Eigenentwicklung der Hardware, um diese auf die gewünschte Größe zu bringen und die Peripherie anzupassen, ist durchaus denkbar. Um das Produkt attraktiver zu gestalten könnte man auch an den Sensorknoten ansetzen, Sensorknoten mit integrierter Power-Line-Kommunikation oder Smart Meter mit integrierter XBee-Anbindung sind durchaus denkbar und realisierbar. Bisher wurde bei der Hardware auf vorhandene Produkte zurückgegriffen um einen Prototypen zur Verfügung stellen zu können, damit das Prinzip des $Eco_2$-Managers gezeigt werden kann.

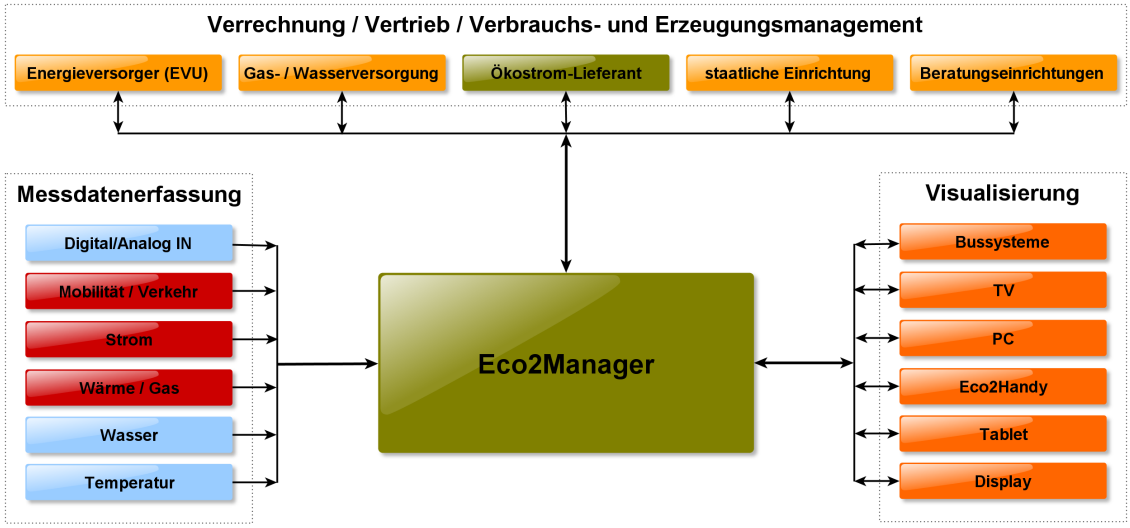



Abbildung 9.1.: möglicher Ausbau $Eco_2$-Manager

Das Zubehör rund um den $Eco_2$-Manager kann auch sehr stark erweitert werden. So ist eine Auswertung mittels Handy-App ($Eco_2$-Handy) oder etwa eine Anbindung an das Fernsehgerät realisierbar und für den Endkunden einfach anzuwenden. Dem Benutzer soll eine möglichst einfache und klare Darstellung über verschiedene Medien geboten werden,

damit dieser auch einen Nutzen, im Sinne von eingesparten Kosten, daraus ziehen kann. Den möglichen Umfang betreffend Messdatenerfassung, Visualisierungen, Datenauswertung usw. ist in Abbildung 9.1 dargestellt. Der Ausbau kann modular gestaltet werden und je nach Bedarf des Kunden erweitert werden. Schnittstellen zu externen Organisationen und Firmen sollen eine Beratung, Anpassung der Leistung an den Kunden oder auch eine Kontrolle durch staatliche Einrichtungen, unter Voraussetzung der nötigen Sicherheit, möglich machen.

In Zukunft mit einbezogen sollte auch der Verkehr werden. Entweder durch eine Erfassung mittels Fragebögen oder durch automatisierte Aufzeichnung im Fahrzeug sollte der $CO_2$ Ausstoß durch Mobilität mit berechnet werden. Denkbar wäre auch eine *persönliche Smart-Card* für die Aufzeichnung. Um das Auto nutzen zu können, muss die Smart-Card eingeschoben werden, Bus-, Zug- oder Taxifahrten können ebenso aufgezeichnet werden wie Flugreisen. Ein großer Anteil des persönlichen $CO_2$-Fingerabdrucks würde durch diese Technologie zusätzlich erfasst werden können.

Insofern ist zu hoffen, dass dieses, sicher zukunftsweisende, Projekt weitergeführt wird und auch erfolgreich in möglichst viele Haushalte integriert werden kann. Eine gute Zusammenarbeit mit den Energieversorgungsunternehmen ist hoffentlich gegeben, es sollte meiner Meinung nach auch in deren Interesse sein, zukunftsträchtige und umweltschonende Energie liefern zu können.

# Anhang A.

# Abkürzungen

**.NETMF** .NET Micro Framework
**ACK** Acknowledged
**ADW** Analog Digital Wandler
**API** Application Programming Interface
**BCC** Block Check Character
**CR** Carriage Return
**CRC** Cyclic Redundancy Checks
**CLR** Common Language Runtime
**EDIS** Energie Data Identification System
**EOT** End of Transmission
**ETS** Emission Trading System
**ETX** End of Text
**EVU** Energieversorgungsunternehmen
**EWR** Extended Weak Reference
**FTDI** Future Technology Devices International Ltd.
**GC** Garbage Collection
**GUI** Graphical User Interface
**HAL** Hardware Abstraction Layer
**IDE** Integrated Development Environment
**IGL** Immissionsschutzgesetz-Luft
**JDBC** Java Database Connectivity
**JSON** JavaScript Object Notation
**JTAG** Joint Test Action Group
**LF** Line Feed
**NACK** Not Acknowledged
**NAK** Negative Acknowledged
**OBIS** Object Identification System
**PAL** Platform Adaption Layer
**PLC** Power Line Communication
**PWM** Pulsweitenmodulation
**RDBMS** Relational Database Management System
**RLP** Runtime Loadable Procedure
**RTC** Real Time Clock
**SiLabs** Silicon Laboratories
**SOH** Start of Heading
**SPI** Serial Peripheral Interface

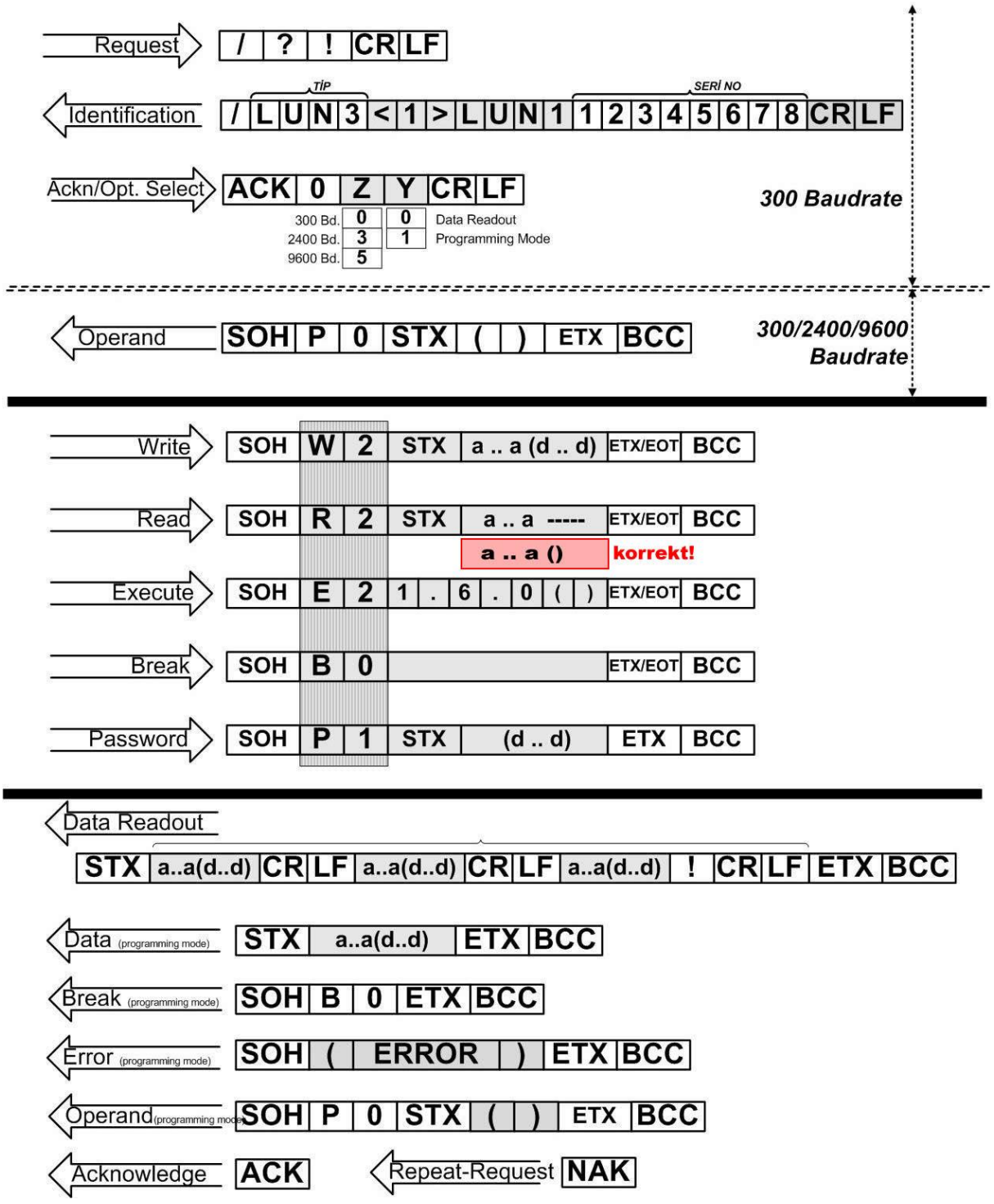
**SPOT** Smart Personal Objects Technology
**SQL** Structured Query Language
**STX** Start of Text
**TI** Texas Instruments
**USB** Universal Serial Bus
**VM** Virtual Machine
**WPF** Windows Presentation Foundation
**XAML** Extensible Application Markup Language
**XML** Extensible Markup Language
**XOR** exklusiv oder

# Anhang B.

# Technische Informationen

- Luna® Smart Meter Kommunikationsprotokoll
- Luna® Smart Meter Parametertabelle (OBIS-Codes)

Luna® Smart Meter Kommunikationsprotokoll
(korrigiert)



**Parametertabelle LUNA Stromzähler**

| | | | |
|---|---|---|---|
| **1** | **Seriennummer** | **0.0.0** | |
| **2** | **Uhrzeit** | **0.9.1** | |
| **3** | **Datum** | **0.9.2** | |
| **4** | **Tag** | **0.9.5** | |
| **5** | **Pmax aktuell** | **1.6.0** | |
| **6** | **T aktuell** | **1.8.0** | |
| **7** | **$T_1$ aktuell** | **1.8.1** | |
| **8** | **$T_2$ aktuell** | **1.8.2** | |
| **9** | **$T_3$ aktuell** | **1.8.3** | |
| **10** | **$T_4$ aktuell** | **1.8.4** | |
| **11** | **$E_R$ aktuell** | **5.8.0** | Induktive Blindleistung |
| **12** | **$K_R$ aktuell** | **8.8.0** | Kapazitive Blindleistung |
| **13** | **Batteriestatus** | **96.6.1** | |
| **14** | **Erstes Öffnungsdatum Gehäuse** | **96.70** | Datum wann das Gehäuse geöffnet wurde |
| **15** | **KLEM aktuell** | **96.71** | Datum wann der Klemmenblock geöffnet wurde |
| **16** | **PMAX verg. 1…12** | **1.6.0*1…12** | Maximalverbrauch Aufzeichnung |
| **17** | **$T_1$ verg. 1…12** | **1.8.1*1…12** | T1 Aufzeichnung |
| **18** | **$T_2$ verg. 1…12** | **1.8.2*1…12** | T2 Aufzeichnung |
| **19** | **$T_3$ verg. 1…12** | **1.8.3*1…12** | T3 Aufzeichnung |
| **20** | **$T_4$ verg. 1…12** | **1.8.4*1…12** | T4 Aufzeichnung |
| **21** | **$E_R$ verg. 1…12** | **5.8.0*1…12** | Ind. Blindleistung Aufzeichnung |
| **22** | **$K_R$ verg. 1…12** | **8.8.0*1…12** | Kap. Blindleistung Aufzeichnung |
| **23** | **KLEM verg. 1…12** | **96.71*1…12** | Datum Öffnung Klemmenblock Aufzeichnung |
| **24** | **Zähler zurücksetzen** | **0.1.0** | Verbrauchszähler zurücksetzen? |
| **25** | **DST verg. 1…12** | **0.1.2*1…12** | Geschichte Resetinfos zurücksetzen |
| **26** | **GU Zähler** | **96.77.4** | letzte Warnung (Datum) Spannung |
| **27** | **GU verg. 1…10** | **96.77.4*1…10** | Warnungsaufzeichnung (Datum) Spannung |
| **28** | **AU Zähler** | **96.77.5** | Letzte Warnung (Datum) Strom |
| **29** | **AU verg. 1..10** | **96.77.5*1…10** | Warnungsaufzeichnung (Datum) Strom |
| **30** | **UFK aktuell** | **96.77.0** | Drehstromzähler Unterbrechungsalarm |
| **31** | **UFK verg. 1…10** | **96.77.0*1…10** | Drehstromzähler Unterbrechungsalarm Aufzeichnung |
| **32** | **R-FK** | **96.7.1** | Unterbrechung Zähler Phase 1 |
| **33** | **S-FK** | **96.7.2** | Unterbrechung Zähler Phase 2 |
| **34** | **T-FK** | **96.7.3** | Unterbrechung Zähler Phase 3 |
| **35** | **R-FK verg. 1…10** | **96.7.1*1…10** | Unterbrechung Zähler Phase 1 Aufzeichnung |
| **36** | **S-FK verg. 1…10** | **96.7.2*1…10** | Unterbrechung Zähler Phase 2 Aufzeichnung |
| **37** | **T-FK verg. 1…10** | **96.7.3*1…10** | Unterbrechung Zähler Phase 3 Aufzeichnung |
| **38** | **Maximalverbrauch Zeit** | **0.8.0** | |
| **39** | **Herstellungsdatum** | **96.1.3** | |
| **40** | **Datum des Tarifwechsels** | **96.2.2** | |
| **41** | **Datum der Kalibrierung** | **96.2.5** | |
| **42** | **Tarifstunden (werktags)** | **96.50** | |
| **43** | **Tarifstunden (Samstags)** | **96.51** | |
| **44** | **Tarifstunden (Sonntags)** | **96.52** | |
| **45** | **Tarifimpulse? (werktags)** | **96.60** | |
| **46** | **Tarifimpulse? (Samstags)** | **96.61** | |
| **47** | **Tarifimpulse? (Sonntags)** | **96.62** | |

| | | |
|---|---|---|
| **48** | **Neues Passwort** | **96.96** |
| **49** | **Erste Verwendung Zählertarif?** | **96.99** |
| **59** | **Dauer der Belastung?** | **96.8.0** |
| **60** | **V1rms** | **32.7.0** |
| **61** | **I1rms** | **31.7.0** |
| **62** | **cosPhi1** | **33.7.0** |
| **63** | **V2rms** | **52.7.0** |
| **64** | **I2rms** | **51.7.0** |
| **65** | **cosPhi2** | **53.7.0** |
| **66** | **V3rms** | **72.7.0** |
| **67** | **I3rms** | **71.7.0** |
| **68** | **cosPhi3** | **73.7.0** |
| **69** | **F1** | **34.7.0** |
| **70** | **F2** | **54.7.0** |
| **71** | **F3** | **74.7.0** |

**Deutsche Übersetzung aus der türkischen Originalversion.**

# Literaturverzeichnis


[Alb11] Joseph Albahari. Threading in C#, April 2011.

[Axe99] Jan Axelson. Designing RS-485 Circuits. *Circuit Cellar - The Computer Applications Journal*, pages 20–24, Juni 1999.

[Bal07] Peter Balog. TCP/IP Client Server Architektur. Studienbrief, Fachhochschule Technikum Wien, August 2007.

[DT07] Rob S.Miles Donald Thompson. *Embedded Programming with the Microsoft .NET Micro Framework*. Microsoft, Juni 2007.

[DWDG11] S.S.S.R. Depuru, Lingfeng Wang, V. Devabhaktuni, and N. Gudi. Smart meters for power grid - Challenges, issues, advantages and status. In *Power Systems Conference and Exposition (PSCE), 2011 IEEE/PES*, pages 1 –7, März 2011.

[Eur09] Europäisches Parlament. Festsetzung von Emissionsnormen für neue Personenkraftwagen im Rahmen des Gesamtkonzepts der Gemeinschaft zur Verringerung der CO2-Emissionen von Personenkraftwagen und leichten Nutzfahrzeugen. Verordnung Nr 443/2009, Juni 2009.

[GHI09] GHI Electronics. *ChipworkX Development System Pinout*. GHI Electronics, 2009.

[GHI10a] GHI Electronics. *ChipworkX Development System Rev. 1.10: Getting Started*. GHI Electronics, Juli 2010.

[GHI10b] GHI Electronics. *ChipworkX User Manual Rev. 4.2*. GHI Electronics, Oktober 2010.

[Gil12] David Gilbert. *JFreeChart Developer Guide*. Object Refinery Limited, Jannuar 2012.

[Goe11] Jan Goeltenboth. *Embedded Software Engineering*. HSR Rapperswil, Jannuar 2011.

[Gol91] David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic, März 1991.

[IEE08] IEEE 754-2008 - IEEE Standard for Floating-Point Arithmetic, August 2008.

[Jon07] Bradley Jones. The .NET Micro Framework: A First Look. Webbook, September 2007.

[Kre10] Jay A. Kreibich. *Using SQLite*. O'Reilly Media, first edition, 2010. ISBN: 978-0-596-52118-9.

[Kue08] Jens Kuehner. *Expert .NET Micro Framework*. Apress, 2008. ISBN: 978-1-59059-973-0.

[Lun12] Luna Elektronik Elektrik Sayaclari. *Luna aktiver/reaktiver elektronischer Stromzähler - Bedienungsanleitung*, März 2012.

[Mad11] Mader Andreas, BSc. Sensoranbindung für CO2-Messung. Masterprojekt, Technische Universität Graz, November 2011.

[Mes02] Christian Mester. IEC-1107-Stromzähler-Interface zum Auslesen elektronischer Verbrauchszähler. *Elektor 1/2002*, 373:52–54, 2002.

[Mil10a] Colin Miller. Bicycle Computer Nr 8 - Changing Display Size - the UI Thread, EWR, and Touch Calibration. MSDN Blog, Juli 2010.

[Mil10b] Colin Miller. Connected Devices Using the .NET Micro Framework. MSDN Blog, Oktober 2010.

[Net11] Netzwissen.de. Alles über Klimazertifikate. Online-Artikel, 2011.

[Ora12] Oracle Corporation. Internationalizing a GUI Form in NetBeans. Online-Artikel, März 2012.

[Pri10] PricewaterhouseCoopers Österreich im Auftrag der E-Control. Studie zur Analyse der Kosten-Nutzen einer österreichweiten Einführung von Smart Metering, Juni 2010.

[Shi02] Joe Sam Shirah. Java internationalization basics. IBM developerWorks, April 2002.

[sql12] sqlite.org. SQLite Documentation. Online-Dokumentation, Jannuar 2012.

[Ull07] Christian Ullenboom. *Java ist auch eine Insel*, volume 7. Galileo Computing, 2007. ISBN 978-3-8362-1146-8.

[Zah02] Martin Zahn. Delegates and Events in C# / .NET. Online-Artikel, 2002.