BigData mit RaspberryPi und F#

Seminararbeit

Version 0.90

Funktionale Programmierung
Zürcher Hochschule für Angewandte Wissenschaften

Simon Lang, Daniel Brun

20. Juni 2016

Abstract

Ausgangslage und Ziel

Sensordaten sammeln

Sensordaten aufbereiten und auswerten

Abstract. Abstract Template: ZHAW_FUP_SEM_IOT_Abstract.tex

Eigenständigkeitserklärung

Hiermit bestätigt ich, dass vorliegende Semesterarbeit zum Thema „BigData mit RaspberryPi und F#“ gemäss freigegebener Aufgabenstellung ohne jede fremde Hilfe und unter Benutzung der angegebenen Quellen im Rahmen der gültigen Reglemente selbständig verfasst wurde.

Zürich, 20.06.2016 Simon Lang, Daniel Brun

Inhaltsverzeichnis

KAPITEL 1

Einleitung

Diese Arbeit wurde als Seminararbeit zur Vorlesung von funktionalen Programmiersprachen verfasst. In diesem Kapitel wird die Aufgabenstellungen und Rahmenbedingungen der Arbeit erläutert.

1.1 Hintergrund

Einer immer grösseren Beliebtheit erfreuen sich kleine Alltagsgegenstände welche mit dem Internet verbunden sind. Dieser Bereich wird IoT genannt. Diese Gegenstände sind in der Lage Daten zu erheben und weiterzuleiten. Da es zukünftig voraussichtlich immer mehr IoT Gegenstände geben wird fallen immer mehr Daten an. Diese Daten werden wegen ihrer Masse auch BigData genannt.

In dieser Arbeit wird evaluiert wie sich funktionale Programmiersprachen im Bezug auf IoT eignen, um BigData auszuwerten.

1.2 Ziel

Mit einem IoT Gerät sollen Daten aufgezeichnet werden. Diese werden als BigData gesammelt und sollen mit einer funktionalen Programmiersprache ausgewertet und ansprechend ausgegeben werden.

Das Hauptziel der Arbeit besteht darin zu überprüfen wie geeignet funktionale Programmiersprachen für die Auswertung von BigData sind. Als Nebenziel soll evaluiert werden, ob eine funktionale Programmiersprache zum erfassen von Daten auf einem IoT Gerät verwendet werden kann.

1.3 Aufgabenstellung

Die freigegebene Aufgabenstellung lautet wie folgt:

- Projektname: Seminar BigData mit RaspberryPi und F#
- Ausgangslage: Durch die rasante Entwicklung im Bereich IoT ergeben sich viele neue Anwendungsmöglichkeiten. Da der RaspberryPI immer leistungsfähiger geworden ist, soll evaluiert werden ob er sich für den Einsatz von funktionalen Sprachen im Bereich BigData eignet.
- Ziel der Arbeit: Es soll gezeigt werden wie F# Sharp auf einem RaspberryPi im Bereich BigData und IoT eingesetzt werden kann.
- Aufgabenstellung: Es soll gezeigt werden, wie eine funktionale Programmiersprache (F#) im Kontext von BigData und IoT eingesetzt und verwendet werden kann. Es soll eine Anwendung zur Sammlung von Sensordaten auf einem Raspberry PI und eine Anwendung zur Analyse / Auswertung der gesammelten Daten implementiert werden.

1.4 Erwartete Resultate

Gemäss freigegebener Aufgabenstellung werden folgende Resultate erwartet:

- Dokumentation
- Implementation / Prototyp

1.5 Abgrenzung

Aufgrund des Umfanges der Arbeit und der begrenzten Zeitdauer werden folgende Punkte von der Arbeit abgegrenzt:

Hemmer Abgrenzige?

- **...**
  ...

KAPITEL 2

Recherche

In diesem Kapitel werden die Grundlagen recherchiert wie die beiden Teilprojekte „Sensordaten sammeln“ und „Sensordaten auswerten (BigData)“ angegangen werden könnten.

2.1 Ausgangslage

Die Vorlesung zu diesem Semniar befasst sich mit den Konzepten der Funktionalen Programmierung. Zur Veranschaulichung dieser Konzepte wurde die Programmiersprache F# des .NET-Frameworks verwendet. Aufgrund dessen haben wir uns entschieden auch dieses Seminar mit der uns nun bekannten Sprache F# umzusetzen. Als IoT Gerät wird ein Raspberry Pi[1] verwendet. Der Raspberry Pi ist ein Einplatinencomputer welcher von der britischen Raspberry Pi Foundation entwickelt wurde. Der Raspberry Pi bietet den Vorteil, dass er sehr weit verbreitet ist[2], er kostengünstig ist und es inzwischen eine sehr grosse Anzahl an Sensoren auf dem Markt gibt mit welchem man Daten sammeln kann[3].

2.2 Der Raspberry Pi

Abbildung 2.1: Raspberry Pi 2 Model B
Quelle: https://www.raspberrypi.org/wp-content/uploads/2015/01/Pi2ModB1GB_-comp.jpeg

1 Rasb.
2 Rasd.
3 Rasa.

Wie bereits im vorangehenden Kapitel beschrieben, handelt es sich beim Raspberry Pi um einen Einplatinencomputer welcher von der Raspberry Pi Foundation entwickelt und vertrieben wird. Er hat ungefähr die Grösse einer Kreditkarte und bietet zahlreiche On-Board Schnittstellen wie USB-, HDMI und Audio Anschlüsse (Abhängig vom konkreten Modell). Zusätzlich stehen eine bestimmte Anzahl an GPIO-Pins (General Purpose Input / Output) zur Verfügung. Mit Hilfe dieser Pins lassen sich zum einen Erweiterungs-Boards anschliessen und zum anderen können auch über ein spezielles Erweiterungsboard eigene Schaltungen, etc. gebaut und verlötet werden. Die Anzahl und genaue Funktion der einzelnen GPIO-Pins ist vom konkreten Raspberry PI Modell abhängig.

Am 29 Februar 2016 ist die neuste Version, der Raspberry Pi 3 (Model B), auf dem Markt erschienen[1]. Einige Zahlen zu dem Gerät:

- 1.2GHz 64-bit quad-core ARM Cortex-A53 CPU (10x die Leistung eines Raspberry Pi 1 und 50-60% die Leistung eines Raspberry Pi 2)
- Integriertes 802.11n wireless LAN und Bluetooth 4.1
- Komplette Kompatibilität zu Raspberry Pi 1 und 2 (Model B)

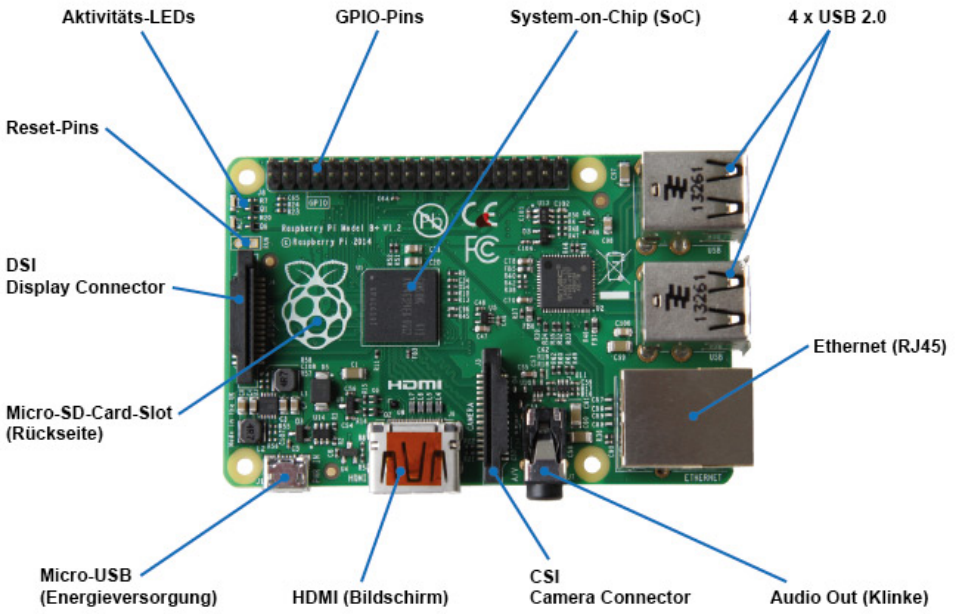


Abbildung 2.2: Raspberry Pi 2 Model B Überblick
Quelle: https://www.elektronik-kompendium.de/sites/raspberry-pi/bilder/19052512.jpg

1 Rasc.

2.2.1 Raspberry Pi Modelle im Überblick

	Raspberry Pi Model A	**Raspberry Pi Model A+**	**Raspberry Pi Model B**	**Raspberry Pi Model B+**	**Raspberry Pi 2 Model B**	**Raspberry Pi 3 Model B**	**Raspberry Pi Compute**	**Raspberry Pi Zero**
Gewicht in Gramm	31	23	40	45	40	40	7	9
System-on-a-Chip (SoC):	BCM2835				BCM2836	BCM2837	BCM2835	
CPU Kerne	1	1	1	1	1	4	1	1
CPU Takt in MHz	700				900	1200	700	1000
CPU Architektur	ARMv6 (32-bit)				ARMv7 (32-bit)	ARMv7 (64-bit)	ARMv6 (32-bit)	
GPU Takt in MHz	250					300/400	250	
Arbeitsspeicher in MB	256		256 / 512		512	1024	512	
Pins	26	40	26		40		60	40
GPIO-Pins	17	26	17		26		48	26

2.3 F# mit dem Raspberry Pi

Um F# auf dem dem Raspberry PI auszuführen, gibt es grundsätzlich zwei Möglichkeiten, welche nachfolgend erläutert werden. Da es sich bei F# um eine Sprache des Microsoft .NET-Frameworks handelt, wird für die Ausführung zwingend eine Implementierung des .NET-Frameworks benötigt.

2.3.1 Linux (Raspbian)

Ein weit verbreitetes Betriebsystem für den Raspberry Pi ist das Raspbian[1] OS. Bei dem Namen handelt es sich um eine Zusammenfassung von Raspberry und Debian. Demnach handelt es sich auch um eine Debian Distribution, welche spezifisch für den Raspberry Pi entwickelt wurde.

Eine Möglichkeit um unter Linux, beziehungsweise Raspbian, F# auszuführen ist das Mono-Framework[2]. Dabei handelt es sich um eine Open Source Implementierung von Microsoft's .NET Framework.

2.3.2 Windows 10 IoT

Microsoft hat mit Windows 10 IoT eine Version ihres Betriebssystem herausgebracht, welches speziell für leistungsschwächere Geräte entwickelt wurde[3]. Bei der IoT Version von Windows 10 ist das .NET Framework bereits standardmässig an Bord. Demnach sollte es keine Probleme geben um F# auf dieser Plattform zu betreiben.

2.4 Verwendete Hardware für die Umsetzung

Wir haben uns entschieden für die Umsetzung die nachfolgend aufgelisteten Hardware-Komponenten zu verwenden:

- Raspberry Pi 2 Model B
- Raspberry Pi 3 Model B
- GrovePi Sensoren
  - Temperature & Humidity Sensor
  - Sound Sensor
  - Light Sensor
  - Blue LED

1 Fro.
2 Mon.
3 Win.

GrovePi Sensoren

Für den Raspberry Pi gibt es viele Sensoren auf dem Markt. Herauskristallisiert hat sich jedoch das Starter Kid GrovePi+[1]. Dieses beinhaltet unter anderem Ton-, Temperatur-, Feuchtigkeit- und Lichtsensoren. Der Vorteil an den GrovePi Sensoren besteht an den geringen Kosten, dem einfachen Anschluss an den Raspberry Pi und die vielen verfügbaren Beispiele in unterschiedlichsten Programmiersprachen.

Die Recherchen haben ebenfalls gezeigt, dass es eine Library für .NET gibt mit welchem die Sensoren angesprochen werden können[2].

2.5 Datenauswertung

Die von den Sensoren gespeicherten Daten werden in einem noch zu definierenden Format abgespeichert und danach für die Datenauswertung ausgelesen. Dafür wurde vom Dozenten die Library F# Data empfohlen[3]. Diese kann Daten in den Formaten CSV, HTML, JSON und XML entgegennehmen und verarbeiten.

1 Gro.
2 Nug.
3 Fsh.

KAPITEL 3

Sensordaten sammeln

Dieses Kapitel beschäftigt sich mit dem Teilprojekt der Sammlung von Sensordaten. Die Aufbereitung der gesammelten Daten ist Teil eines weiteren Kapitels.

3.1 F# auf dem Raspberry Pi

Im Abschnitt 2.3 F# mit dem Raspberry Pi wurden zwei Möglichkeiten aufgezeigt, welche es ermöglichen F# auf einem Raspberry Pi laufen zu lassen.

Es ist zu erwarten, dass der Weg über Window 10 IoT der einfachere sein wird, da dort das benötigte .NET Framework Teil des Systems ist. In dieser Seminararbeit wurden bewusst beide Varianten (Linux (Raspbian) und Windows 10 IoT) ausprobiert und getestet. Die folgenden Abschnitte erläutern den Setup und die Erkenntnisse aus dem Test der beiden Varianten.

3.1.1 Variante 1: Raspbian Jessie

1: Installation & Konfiguration von Raspbian Jessie

Zuerst wurde Raspbian Jessie auf dem Raspberry Pi installiert. Für die Installation des Betriebssystems wurde NOOBS (New Out Of Box Software)[1] verwendet. NOOBS ist ein Installationsmanager für Betriebssysteme, der es ermöglicht per Klick ein gewünschtes Betriebssystem zu installieren. Es wurde dabei die Anleitung verwendet.

Je nach Netzwerk und Setup sind einige kleinere weitere Konfigurationen notwendig (z.B. Konfiguration des WLAN, Deaktivierung von IPv6 in einem IPv4 Netzwerk).

1 Noo.

2: Installation des Mono-Frameworks & F#

Im Anschluss wurde die neuste Version des Mono-Frameworks (4.2.3) und der F#-Interactive Shell (F# Version 4.0) über die in den Software-Repositories vorhandenen Pakete installiert. Mono ist eine Open Source implementation von Microsofts .NET Framework und wird benötigt, um F# Code auszuführen.

```
1 sudo apt-get install mono-complete fsharp
```

3: Installation & Konfiguration GrovePi

Die Installation und Konfiguration wurde anhand der vom Hersteller zur Verfügung gestellten Anleitung durchgeführt. Der Setup wurde anschliessend mit einem bereitgestellten Python-Script getestet.

Schwierigkeiten
Bei der Installation musste ein Git-Repository des Herstellers geklont werde. Dieses beinhaltet die Firmeware und verschiedenste Code-Beispiele und Beispiel Scripts. Das Repository scheint jedoch von gewissen Netzwerkknoten nicht erreichbar zu sein. Das Repository konnte nur im Netzwerk der Zürcher Hochschule für Angewandte Wissenschaften (ZHAW) geklont werden.

Testsetup
Die Installation und Konfiguration des GrovePi wurde über ein Testsetup mit Hilfe einer einfachen LED und eines mitgelieferten Python-Scripts getestet.

4: Anbindung des GrovePi via GrovePi-NuGet-Library

Nun galt es den GrovePi über F# anzusteuern und entsprechende Sensordaten auszulesen. Während der Recherchen wurden wir auf eine NuGet-Library[1] aufmerksam, welche die entsprechende Funktionalität bereitstellt. In der Entwicklungsumgebung (Monodevelop) wurde eine neues F# Projekt angelegt und das entsprechende NuGet-Packet installiert. Der erste testweise Build mit dem NuGet-Paket schlug fehl. Gemäss der Fehlermeldung wird für dieses Paket das Microsoft .NET Framework 5 benötigt. Beim Microsoft .NET Framework handelt es sich um eine Neuauflage des klassischen .NET Frameworks. Im nachfolgenden Abschnitt werden die wichtigsten Aspekte des .NET Frameworks 5 (beziehungsweise .NET Core) erläutert.

.NET Core
Ursprünglich war es geplant .NET Core als „.NET Framework Version 5“ auszurollen. Inzwischen wurde von Microsoft entschieden, .NET Core unter einer eigenständigen Versionsnummer zu führen (Ausgehend davon war die im vorangehenden Abschnitt beschriebene Fehlermeldung nicht mehr korrekt.).

Nachfolgend eine Auflistung der wichtigsten Aspekte und Ziele von .NET Core:

1 Nug.

- Hoher Grad an Portabilität
  - Plattformunabhängigkeit
  - Architekturunabhängigkeit (32-Bit / 64-Bit)
- Open Source Implementation
- Kleine und optimierte Runtime
  - Modularer Aufbau
  - Runtime wird als NuGet Package ausgeliefert
- Default Compiler für x64: „Roslyn“ just-in-time Compiler
- Verfügbare Runtime Varianten: .NET Native (Windows Only), Core CLR (Common Language Runtime, Multiplattform), Weitere ...

Die nachfolgende Grafik zeigt eine grobe Übersicht über das .NET Framework und .NET Core:

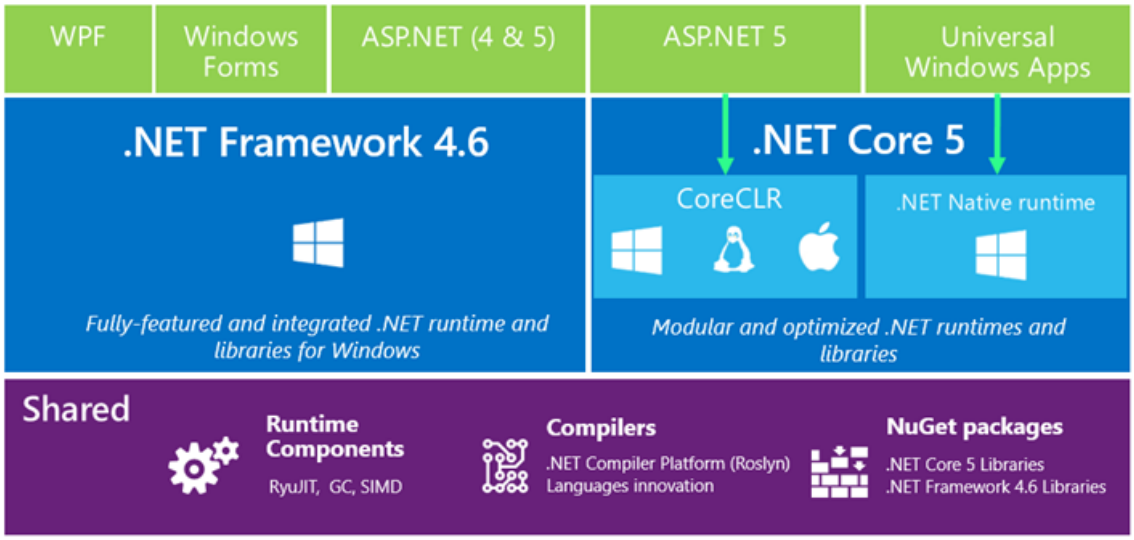


Abbildung 3.1: Übersicht .NET Framework & .NET Core
Quelle: https://blogs.msdn.microsoft.com/bethmassi/2015/02/25/understanding-net-2015/

5: Installation von .NET Core

Aufgrund der Fehlermeldung haben wir nun versucht .NET Core auf dem Raspberry Pi zu installieren. Zur Zeit zu dem dieser Setup durchgeführt wurden, existierten keine offiziellen Microsoft-Quellen zur Installation von .NET Core unter Linux.

Um .NET Core unter Raspbian zu installieren, wird das bereits installierte Mono Framework benötigt. Anschliessend kann das .NET / DotNet Execution Environement (DNX) installiert

werden. Diese Execution Environement beinhaltet alle notwendigen Libraries, um .NET Core Applikation auszuführen. Um DNX zu installieren ist es am einfachsten den .NET / DotNet VersionManager (DNVM) zu verwenden. Mit dem Version Manager können einfach verschiedene Version von .NET parallel installiert werden.

Installation DNVM

```
curl -sSL https://raw.githubusercontent.com/aspnet/Home/dev/dnvminstall.sh |
DNX_BRANCH=dev sh && source ~/.dnx/dnvm/dnvm.sh
```

Installation DNX .NET Core

```
sudo apt-get install libunwind8 gettext libssl-dev libcurl4-openssl-dev zlib1g
libicu-dev uuid-dev
mozroots --import --sync
dnvm upgrade -u
dnvm install latest -r coreclr -u
```

Die oben beschriebenen Schritte konnten soweit problemlos durchgeführt werden. Anschliessend galt es das Test-Projekt auf das neue Build-System von DNX umzustellen, welches mit .NET Core verwendet werden muss. Während dieser Umstellung hat sich gezeigt, dass der DNVM und die DNX bereits wieder verworfen wurde und direkt durch die .NET Core CLI (Command Line Interface) abgedekct werden[1]. Zum Zeitpunkt dieser Arbeit bestand jedoch neben dem DNVM keine andere Möglichkeit .NET Core unter Linux / Mono zu installieren. Glücklicherweise konnte das .NET Core CLI zusammen mit der .NET Core Runtime dennoch via DNVM installiert werden.

Die nächste Herausforderung bestand anschliessend mit dem neuen Build-System das korrekte Target-Framework zu finden. Aufgrund dessen, dass es noch keinen ersten finalen Release von .NET Core CLI gibt, kursierten im Web verschiedenste Definitionen für das Target-Framework. Darunter zum Beispiel „dotnet“ und „dnxcore50“.

Längere Recherchen und Analysen haben anschliessend zwei Fakten zu Tage gebracht, mit welchen wir nicht gerechnet hatten.

- **.NET Core in 32-Bit**
  Aktuell gibt es vom .NET Core keine 64-Bit Implementation. Sämtliche Raspberry Pi Modelle basieren auf einer ARM 32-Bit Architektur. Ausgenommen davon ist das neuste Model, der Raspberry Pi 3 Model B. Ebenfalls gibt es Raspbian aktuell nur in einer 32-Bit Version.
- **GrovePi NuGet Paket verwendet Windows-Only Funktionalitäten**
  Das GrovePi NuGet Paket hat Abhängigkeiten zur Universal Windows Platform (UAP). Die Universal Windows Platform gibt es nur für Windows, nicht für Linux

1 http://dotnet.github.io/docs/core-concepts/dnx-migration.html

oder Mac OS. Diese Abhängigkeit rührt wahrscheinlich daher, dass die Library ursprünglich spezifisch für Windows 10 IoT entworfen wurde.

6: Finaler Setup

Aufgrund dieser letzten Erkenntnisse haben wir uns entschieden die Umsetzung auf Raspbian Jessie (32-Bit) mit dem regulären Mono-Framework zu realisieren. Aufgrund dessen, dass das GrovePi NuGet Paket nicht verwendet werden kann, muss der Zugriff, beziehungsweise die Kommunikation zwischen Raspberry Pi und GrovePi auf eine andere Art gelöst werden.

3.1.2 Variante 2: Windows 10 IoT

Windows 10 IoT ist eine Version des Betriebssystems von Microsoft, welches speziell für kleinere Geräte mit weniger Rechenleistung konzipiert wurde.

Installation von Windows 10 IoT

Microsoft bietet zwei Varianten an, um Windows 10 IoT auf einem Raspberry Pi zu installieren. Bei ersteren wird das Betriebsystem direkt auf die SD-Karte installiert[2]. Dazu muss das Windows 10 IoT Core Dashboard auf einem Computer installiert werden. Anschliessend wird eine ISO Datei heruntergeladen, welches eine Installationsdatei beinhaltet welche über ein richtiges Windows 10 gestartet werden muss. Dies Datei kompiliert eine FFU Datei welche über das Core Dashboard installiert werden muss. Danach sollte das Raspberry Pi startbar sein.

Leider startete das Raspberry Pi nach dem genannten Prozedere nicht, sondern blieb bei dem Rainbow Screen[3] hängen.

2 Ins.
3 Rpi.

Abbildung 3.2: Raspberry Pi rainbow screen

Die zweite Variante der Installation von Windows 10 IoT arbeitet mit dem NOOBS Installer (siehe **?? ??**), welcher von der Raspberry Pi Foundation zur Verfügung gestellt wird. Dazu muss NOOBS auf einer SD-Karte installiert werden. Danach kann man die Karte in das Raspberry einfügen und starten. Um Windows 10 IoT zu installieren, muss in NOOBS das Wireless aktiviert werden, damit der Installer das Windows 10 IoT aus dem Internet herunterladen kann. Danach installiert es automatisch.

Die Installation über NOOBS war einiges einfach als diejenige über das Core Dashboard und war schlussendlich auch erfolgreich.

Installation des Mono-Frameworks & F#

Windows 10 IoT wird bereits mit dem .NET Micro Framework. Deshalb war keine Installation für .NET nötig.

Installation von .NET Core

Da für die Entwicklung für das GrovePi das .NET Core nötig ist, wurde nach einer Anleitung[4] versucht dieses zu installieren. Als erstes muss DNVM installiert werden, danach die CoreCLR[5]

Installation DNVM

4 **How_to_run_DotNET_Core_Application_on_Pi2**.
5 **CoreCLR_2016-06-17**.

```
@powershell -NoProfile -ExecutionPolicy unrestricted -Command "`&{$Branch='dev';
iex ((new-object net.webclient).DownloadString('https://raw.githubusercontent.com
/aspnet/Home/dev/dnvminstall.ps1'))}"'
```

Installation CoreCLR

```
c:\> dnvm install -r coreclr -arch x64 latest -u
```

Dies installiert die CoreCLR 32bit Version für einen ARM Prozessor, welcher auf dem Raspberry Pi verwendet wird. Leider funktionierte dies nicht.

```
Run Build Command:/usr/bin/make "cmTryCompileExec3109069071/fast"
/usr/bin/make -f CMakeFiles/cmTryCompileExec3109069071.dir/build.make CMakeFiles/
cmTryCompileExec3109069071.dir/build
make[1]: Entering directory `/home/\dfrac{win10iot}{den}/coreclr/bin/obj/Linux.
x64.Debug/CMakeFiles/CMakeTmp'
/usr/bin/cmake -E cmake_progress_report /home/win10iot/coreclr/bin/obj/Linux.x64.
Debug/CMakeFiles/CMakeTmp/CMakeFiles 1
Building C object CMakeFiles/cmTryCompileExec3109069071.dir/testCCompiler.c.o
/usr/bin/clang-3.5   -Wall -std=c11    -o CMakeFiles/cmTryCompileExec3109069071.
dir/testCCompiler.c.o   -c /home/win10iot/coreclr/bin/obj/Linux.x64.Debug/
CMakeFiles/CMakeTmp/testCCompiler.c
Linking C executable cmTryCompileExec3109069071
/usr/bin/cmake -E cmake_link_script CMakeFiles/cmTryCompileExec3109069071.dir/
link.txt --verbose=1
/usr/bin/clang-3.5   -Wall -std=c11     CMakeFiles/cmTryCompileExec3109069071.dir
/testCCompiler.c.o  -o cmTryCompileExec3109069071 -rdynamic
make[1]: Leaving directory `/home/win10iot/coreclr/bin/obj/Linux.x64.Debug/
CMakeFiles/CMakeTmp'
```

Deshalb wurde entschieden, das Projekt mit der Variante 1: Rasbian Jessie fortzusetzen.

3.2 Speicherung der Sensordaten

Zum Abspeichern der Sensordaten haben sich verschiedene Datenformate angeboten. Um die Komplexität möglichst gering zu halten und maximale Flexibilität zu erreichen, haben wir uns entschieden die Sensordaten File-basiert abzuspeichern.

Neben CSV (Comma Separated Values) standen auch JSON (JavaScript Object Notation) oder XML (Extensible Markup Language) zur Auswahl. Da die Daten in einem kontinuierlichen Stream geschrieben werden müssen haben wir uns schlussendlich für ein klassisches CSV-File entschieden. Dies bietet den Vorteil, dass neue Datensätze ohne Probleme laufend ans Ende der Datei angehängt werden können.

Bei JSON und XML ist dies nicht ohne weiteres möglich, da die Daten dort in Hierarchischer Form abgespeichert werden.

3.3 Umsetzung

In diesem Kapitel wird die konkrete Umsetzung der Problemstellung „Sammeln von Sensordaten“ beschrieben.

3.3.1 Verwendete Hardware

Für die Realisierung wurden nachfolgend aufgelisteten Hardware-Komponenten verwendet:

- Raspberry Pi 2 Model B
- Raspberry Pi 3 Model B
- GrovePi Sensoren
  - Temperature & Humidity Sensor (GrovePi Port D3)
  - Sound Sensor (GrovePi Port A0)
  - Light Sensor (GrovePi Port A2)
  - Blue LED (GrovePi Port D4)

3.3.2 Verwendete Software

Für die Realisierung werden folgende Softwarekomponenten verwendet:

- Raspbian Jessie
- GrovePi Firmware + Beispiele
- Mono 4.2

3.3.3 Dokumentation der Implementation

Die konkrete Implementation der Problemstellung „Sensordaten sammeln“ wird im Abschnitt 3.4 Dokumentation der Implementation dokumentiert.

3.3.4 Datentransfer zum Raspberry Pi

Die kompilierten Sourcen wurden via Linux-Befehl „scp“ auf den Raspberry Pi übertragen.

```
1 scp -rp ./Raspberry.FGrove pi@raspberrypi.local:/home/pi/Development/FUP/
```

3.3.5 Datentransfer vom Raspberry Pi

Der Datentransfer vom Raspberry Pi wurde durch einen einfachen HTTP-Server auf Python-Basis bewerkstelligt. Python wurde bei der Installation des GrovePi automatisch mitinstalliert. Dadurch konnten wir den „SimpleHTTPServer“ nutzen, welcher standardmässig mit Python ausgeliefert wird.

```
python -m SimpleHTTPServer 8000
```

3.3.6 Betrieb & Starten der Applikation

Damit der Raspberry Pi zum Sammeln von Sensordaten theoretisch auch unabhängig von einem Netzwerk betrieben werden könnte, wurde der Raspberry Pi als WiFi-Access-Point konfiguriert. Beim Start wird geprüft, ob ein bekanntes WiFi-Netzwerk verfügbar ist. Ist dies nicht der Fall, werden entsprechend die Konfigurationen für den WiFi-Access-Point geladen. Die Konfiguration dieses Access-Points, beziehungsweise Ad-Hoc-Netzwerkes, wurde anhand von Anleitungen von Lasse Christiansen und cpetty33 durchgeführt.

Mit diesem Setup kann nun jederzeit auf den Raspberry Pi zugegriffen und entweder die Anwendung gestartet / gestoppt oder die Daten heruntergeladen werden.

Für den Start der Anwendung musste ein zusätzliches Tool namens „screen“ installiert werden. Mit diesem Tool ist es einfach möglich mehrere Hintergrundverarbeitungen zu starten, welche nicht automatisch beendet werden, wenn die aktuelle Session (SSH-Verbindung) beendet wird.

Start der Anwendung

```
ssh pi@raspberrypi.local (SSH into Raspberry Pi))
screen -S rpfs (create new screen session "`rpfs"')
python -m SimpleHTTPServer 8000 (start Python HTTP-Server)
Ctrl + a + c (create new window in current screen session)
sudo fsharpi --lib:/home/pi/Development/FUP/Raspberry.FGrove/libs "/home/pi/
Development/FUP/Raspberry.FGrove/Program.fsx" (start F\# application)
ctrl + a + d (detach from current screen session)
```

Beenden der Anwendung

```
ssh pi@raspberrypi.local (SSH into Raspberry Pi))
screen -r rpfs (attach to the existing screen session "`rpfs"')
ctrl + c (terminate F\# application)
exit (close current window in screen session)
ctrl + c (terminate Python HTTP-Server)
exit (closes current window and screen session)
```

3.4 Dokumentation der Implementation

Nachfolgend werden die wichtigsten Punkte der einzelnen Komponenten dokumentiert.

3.4.1 Komponenten

Die Anwendung besteht aus folgenden Teilkomponenten:

- Raspberry.IO.InterIntegratedCircuit (C# Library)
- Raspberry.GrovePi (C# Library)
- Raspberry.FGrove (F# Applikation)

3.4.2 Source Code

Der Source Code wird in folgendem Git-Repository verwaltet: FUP_Sem_IOT

3.4.3 Raspberry.IO.InterIntegratedCircuit (C# Library)

Bei dieser Komponente handelt es sich um einen Klon / Fork der Teilkomponente „Raspberry.IO.InterIntegratedCircuit“ des Git-Repositorys raspberry-sharp-io von Raspberry#.

Diese Library ist aktuell nicht als NuGet-Packet verfügbar und wurde spezifisch für .NET unter Mono optimiert. Die Library verwendet die NuGet-Packete „Raspberry.IO.GeneralPurpose“ und „Raspberry.System“, welche von Raspberry# veröffentlicht und im genannten Git-Repository gepflegt werden.

Das Ziel dieser Library ist es, den Zugriff auf den Inter-Integrated Circuit (I2c)-Bus des Raspberry Pi zu abstrahieren / vereinfachen.

3.4.4 Raspberry.GrovePi (C# Library)

Diese Komponente stellt unter Verwendung der Library „Raspberry.IO.InterIntegratedCircuit“ spezifische Funktionalitäten für den Zugriff auf den GrovePi via I2c zur Verfügung.

Die Library besteht aus einer einzigen Wrapper-Klasse, welche die I2c und GrovePi spezifischen Elemente für die eingesetzten Sensoren abstrahiert. Über den I2c-Bus müssen spezifische Befehle gesendet werden, um die gewünschten Sensordaten auszulesen. Diese Befehle werden von einem Arduino auf dem GrovePi Board interpretiert und verarbeitet. Die zulässigen Befehlssequenzen wurden den Hersteller-Dokumentationen entnommen.

- Raspberry Pi GPIO Connector
- GrovePi Port Description
- GrovePi Protocol and Adding Custom Sensors
- GrovePi Firmware Documentation
- GrovePi Sound Sensor
- GrovePi Temperature & Humidity Sensor
- GrovePi Light Sensor

Nachfolgend werden die verwendeten Befehlsfolgen aufgezeigt:

I2C auf dem Raspberry

```
SDA-Pin: P1Pin03 (Nummer des SDA GPIO-Pins auf dem Raspberry Pi Board, notwendig für I2C, Data-Line)
SCL-Pin: P1Pin05 (Nummer des SCL GPIO-Pins auf dem Raspberry Pi Board, notwendig für I2C, Clock-Line)
I2C Device Address: 0x04
```

Allgemeines Befehlsschema

```
Byte 0: 1 (Dummy, statischer Wert)
Byte 1: Command (ID der auszuführenden Aktion)
Byte 2: Pin (Nummer des anzusprechenden Pin auf dem GrovePi)
Byte 3: Data (Optional, Input-Daten)
Byte 4: Data (Optional, Input-Daten)
```

Sound / Noise Sensor

```
Byte 0: 1 (Dummy)
Byte 1: 3 (Command: Analog-Read)
Byte 2: 0 (Pin: 0)
Byte 3: 0
Byte 4: 0
Result: 3 bytes (Sensordaten: Umgebungslautstärke)
```

Temperature & Humidity Sensor

```
Byte 0: 1 (Dummy)
Byte 1: 40 (Command: Read DHT-Sensor)
Byte 2: 3 (Pin: 3)
Byte 3: 0
Byte 4: 0
Result: 9 bytes (Sensordaten: Temperatur, Luftfeuchtigkeit)
```

Light Sensor

```
Byte 0: 1 (Dummy)
Byte 1: 3 (Command: Analog-Read)
Byte 2: 2 (Pin: 2)
Byte 3: 0
Byte 4: 0
Result: 3 bytes (Sensordaten: Helligkeit / Lichtintensität)
```

LED (On)

```
Byte 0: 1 (Dummy)
Byte 1: 2 (Command: Digital-Write)
Byte 2: 4 (Pin: 4)
Byte 3: 1 (Data: On)
Byte 4: 0
Result: -
```

LED (Off)

```
Byte 0: 1 (Dummy)
Byte 1: 2 (Command: Digital-Write)
Byte 2: 4 (Pin: 4)
Byte 3: 0 (Data: Off)
Byte 4: 0
Result: -
```

Der GrovePi verarbeitet die eingehenden Befehle sequentiell. Damit sichergestellt ist, dass zum abgesetzten Befehl auch die korrekten Daten zurückgeliefert werden, wurde der Lock-Mechanismus von C# verwendet. Dadurch wird der Zugriff auf den GrovePi so eingeschränkt, dass kein simultaner Zugriff mehr möglich ist.

```
// Byte-Array for results
byte[] ret;

// Create lock, if locked: wait till lock is released
lock (_locker) {
  // Send command to GrovePi
  connection.Write (new[] { (byte)1, (byte)3, (byte)2, (byte)0, (byte)0 });

  // Wait a few ms (allows the arduino to collect the data)
  Thread.Sleep (10);

  // Read the response
  ret = connection.Read (3);

// release the lock
}
```

3.4.5 Raspberry.FGrove (F# Applikation)

Diese Komponente realisiert die Kern-Logik der Anwendung, die Sammlung und Speicherung der Sensordaten. Dazu wird die implementierte C# Library „Raspberry.GrovePi“ verwendet.

Pro Sensor wurde ein eigenständiger Timer implementiert, welche periodisch gestartet und anschliessend zurückgesetzt wird. Pro Timer wurde ein Event-Listener registriert, welcher die spezifischen Sensordaten ausliest und anschliessend in ein Comma Separated Values (CSV)-File schreibt.

Nachfolgend wird die Implementation exemplarisch für den „Sound Sensor“ aufgezeigt:

```
// Create timer for sound / noise sensor
let nTimer, nEventStream = startTimerAndCreateObservable 100

// Subscribe to event
nEventStream |> Observable.subscribe (fun _ -> processNoiseEvent())
```

startTimerAndCreateObservable

```
// Creates and starts a new timer and returns an observable event
let startTimerAndCreateObservable timerInterval =
    // Setup timer
    let timer = new System.Timers.Timer(float timerInterval)

    // Autoreset and enable
    timer.AutoReset <- true
    timer.Enabled <- true

    // Return observable event
    (timer,timer.Elapsed)
```

writeData

```
// Writes the sensordata to a file. If the file doesn't exists it will be created
. Otherwise the data will be appended to the end of the file.
let writeData suffix (data:String) (date:DateTime) (location:String) =
    let filepath = "./data/SensorData-" + suffix + ".csv"

    if not(File.Exists(filepath)) then
        printfn "Data-File doesn't exists, creating file"
        use streamWriter = new StreamWriter(filepath,false)
        streamWriter.WriteLine "Date;Time;SensorData;Location"
        streamWriter.Flush()
        streamWriter.Close()

    use streamWriter = new StreamWriter(filepath,true)
    [ date.ToString("dd.MM.yyyy");
      date.ToString("hh:mm:ss.fff");
      data;
      location]
    |> List.fold (fun r s -> r + s + ";") ""
    |> streamWriter.WriteLine

    streamWriter.Flush()
    streamWriter.Close()
```

readNoiseSensor

```
// Reads the data from the sound / noise sensor.
let readNoiseSensor() =
  grovePiWrapper.readNoiseSensorData()
```

processNoiseEvent

```
// Processes the timer event for the sound / noise sensor.
let processNoiseEvent() =
     let sensorValue = readNoiseSensor()
     writeData "N" (sensorValue.ToString()) DateTime.Now currentLocation
```

Die Daten werden in folgendem Format in die CSV-Datei geschrieben:

```
Date;Time;SensorData;Location
06.05.2016;11:55:30.344;68;Brun@Home;
06.05.2016;11:55:30.367;70;Brun@Home;
06.05.2016;11:55:30.403;69;Brun@Home;
06.05.2016;11:55:30.426;71;Brun@Home;
[...]
```

KAPITEL 4

Sensordaten auswerten und aufbereiten

Im vorhergegangenen Kapitel wurde beschrieben, wie die Daten mittels Raspberry Pi und F# gesammelt wurden. Diese Informationen müssen jetzt aufbereitet und dargestellt werden. Dies wird in diesem Kapitel beschrieben.

4.1 Verwendete Hardware

4.2 Verwendete Software

Die Daten werden mittels F# auf einem Linux Mint (ein Ubuntu Ableger) dargestellt. Dazu muss Mono installiert werden.

Danach werden die Daten mit F# Data[1] ausgelesen, mit F# aufbereitet und mit FSharp.Charting[2] dargestellt.

4.2.1 Mono

Was Mono ist und wie es installiert wird, wurde im **?? ??** beschrieben.

.NET Core wird für F# Data und FSharp.Charting nicht benötigt. Deshalb entfällt auch die Installation von DNVM und DNX.

4.2.2 F# Data

F# Data ist eine Library für den Zugriff auf Daten. Unterstützt werden folgende Formate:

- CSV
- HTML

1 **FShaprp_Data_2016-06-17**.
2 **FSharp_Charting_2016-06-17**.

- JSON
- XML

In dieser Arbeit wurde als Datenformat CSV gewählt, weshalb die folgende Erklärung für dieses Format ausgelegt wurde. Zuerst muss ein Type erstellt werden, welcher für die Zugriff auf die Daten verwendet wird.

```
type Weather = CsvProvider<"http://www.some-weather-service.org/data.csv", ";">
```

Dies lädt eine CSV-Datei von der angegebenen Adresse. Es wird erwartet, dass die Erste Zeile die Namen der Spalten enthält. Zum Beispiel:

```
City;Temperature;Rain
London;6 degree celius;25mm/3h
New York;25 degree celius;<1mm/3h
[...]
```

Der Zweite Parameter im obigen Code ist ein Semicolon und definiert, welches Zeichen als Trenner der Spalten verwendet wird.

Danach können die Daten geladen werden:

```
let weatherData = SensorData.Load(""http://www.some-weather-service.org/data.csv"
)
```

Die Variable weatherData enthält eine Sequenz von dem zuvor erstellten Typen, über welche nun zugegriffen werden kann.

```
let londonWeather = weatherData.Rows |> Seq.Head
let londonTemperature = londonWeather.Temperature
```

4.2.3 FSharp.Charting

FSharp.Charting ist eine Library zur Visualisierung von Daten. Es werden verschiedene Charts unterstützt:

Um ein Chart darzustellen, müssen zuerst die benötigen Libraries geladen werden:

```
#I "packages/FSharp.Charting"
#load "FSharp.Charting.fsx"

open FSharp.Charting
open System
```

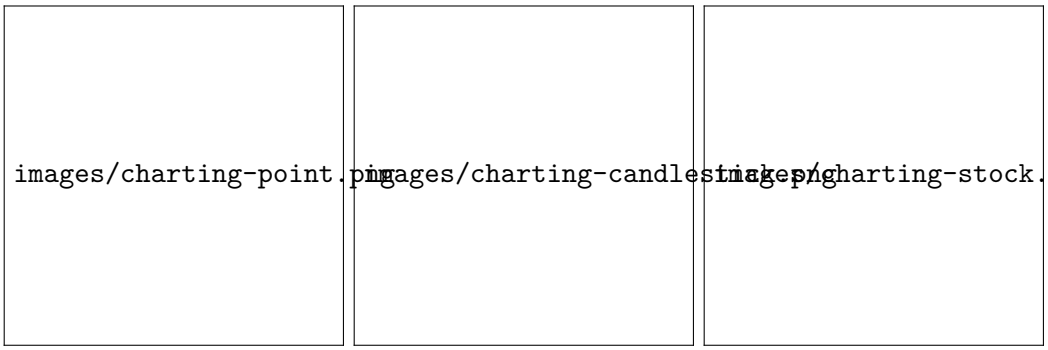
Danach muss eine Sequenz mit Daten befüllt werden:

```
let timesTwo = for x in 1.0 .. 100.0 -> (x, x ** 2.0)
```

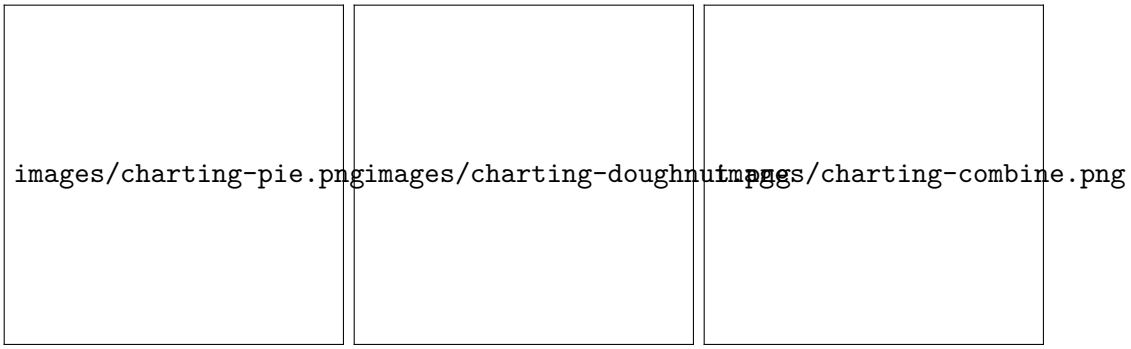
Schlussendlich kann das Chart dargestellt werden:

```
Chart.Line timesTwo
```

(a) Line **(b)** Bar **(c)** Column



(d) Point **(e)** Candlestick **(f)** Stock



(g) Pie **(h)** Doughnut **(i)** Combine

Abbildung 4.1: Chart Typen von FSharp.Charting

Die Ausgabe dieses Codes ist die folgende:

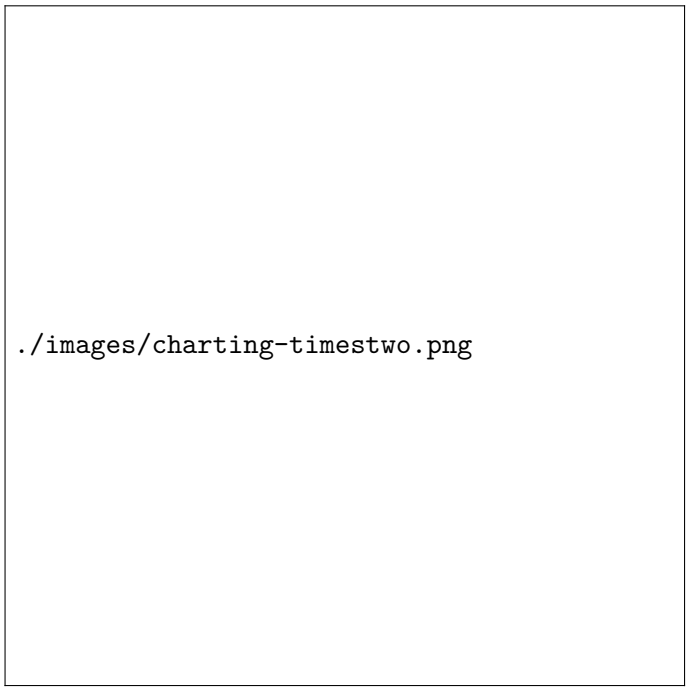


Abbildung 4.2: Beispiel Chart

4.3 Dokumentation der Implementation

Im **??** **??** wurde die Software für die Implementation vorgestellt. In diesem Abschnitt wird auf die Umsetzung für dieses Projekt eingegangen. Zuerst wird das Auslesen der Daten, danach die Aufbereitung und zuletzt die Darstellung erläutert.

4.3.1 Auslesen

Das Auslesen der Daten mit F# Data funktionierte wie im **??** **??** beschrieben. Die Library konnte über den NuGet Package Manager[3] geladen und installiert werden.

```
1 > Install-Package FSharp.Data -Version 2.3.0
```

Danach wurde der Typ erstellt und die Daten geladen:

```
1 type SensorData = CsvProvider<"SensorData-L.csv", ";">
2 let light = SensorData.Load("SensorData-L.csv")
```

Das Skript wurde „Program.fsx“ genannt und lag im gleichen Ordner wie das File „Sensor-Datan-L.csv“. Diese Datei enthält die Daten der Lichtmessung und hatte dieselbe Struktur wie im **??** **??** beschrieben.

3 **NuGet_2016-06-18**.

4.3.2 Aufbereitung

Während der Aufbereitung der Daten wurden zwei Methoden definiert:

```
let chunk size list =
    let chunkedValues = list |> Seq.chunkBySize size
    chunkedValues |> Seq.map (fun r -> r |> Seq.averageBy (fun e -> e |> float))

let format (sensorData:SensorData) =
    let values = sensorData.Rows |> Seq.map (fun r -> r.SensorData)
    chunk 1000 values
```

format nimmt SensorData entgegen und erstellt eine Sequenz mit den Daten der Sensoren. Die CSV-Datei, mit welcher getestet wurde, hat über 100'000 Einträge. Deshalb wurden die Einträge mit der Funktion *chunk* gebündelt.

chunk nimmt eine Anzahl und eine Sequenz entgegen. Der erste Parameter gibt an, wie viele Elemente des der Liste zusammengefasst werden sollen. Um im Chart eine begradigte Linie zu erhalten, wird von den zusammengefassten Einträgen noch der Durchschnitt errechnet.

4.3.3 Darstellung

Mit FSharp.Charting wurde nun die zuvor aufbereiteten Daten dargestellt. Das Verwendung dieser Library gestaltete sich jedoch komplizierter als angenommen.

Zuerst wurde die Library über NuGet installiert:

```
> Install-Package FSharp.Charting
```

Diese funktioniert jedoch nur auf Windows. Deshalb musste sie deinstalliert und statt dessen „FSharp.Charting.Gtk“ verwendet werden.

Nun wurde versucht, die Daten darzustellen mit dem Code für das Auslesen und die Aufbereitung (siehe **??** **??** und **??** **??**):

```
let lightValues = format light
let chart = Chart.Line lightValues
```

Mit dem folgenden Befehl wurde versucht, das Chart darzustellen:

```
> fsharpi Program.fsx
```

Es erschien jedoch folgende Fehlermeldung:

```
[mono-rt] =================================================================
[mono-rt] Got a SIGSEGV while executing native code. This usually indicates
[mono-rt] a fatal error in the mono runtime or one of the native libraries
[mono-rt] used by your application.
[mono-rt] =================================================================
```

Es gab verschiedene Probleme:

1. Falsche Imports
2. Darstellung des Charts
3. X Berechtigung
4. Asynchrone Ausführung

1. Falsche Imports
Es stellte sich heraus, dass das einfache Einbinden von „FSharp.Charting.Gtk.fsx“ nicht genügte. Mit Backtracking der Fehlermeldungen zeigte sich, dass mehrere DLL's benötigt wurden:

```
#I "/usr/lib/cli/gtk-sharp-2.0/"
#I "/usr/lib/cli/gdk-sharp-2.0/"
#I "/usr/lib/cli/atk-sharp-2.0/"
#I "/usr/lib/cli/glib-sharp-2.0/"
#I "bin/Debug/"

#r "gtk-sharp.dll"
#r "gdk-sharp.dll"
#r "atk-sharp.dll"
#r "glib-sharp.dll"
#r "FSharp.Charting.Gtk.dll"
#r "FSharp.Data.DesignTime.dll"
```

2. Darstellung des Charts
Gemäss Dokumentation reicht für die Darstellung des Charts folgender Code:

```
Chart.Line lightValues
```

Dies ist jedoch nur der Fall, wenn über F# Interactive die einzelnen Befehle einzeln ausgeführt werden. Dies führt einen Handler aus welcher das Chart anzeigt[4]. Als fsx Programm muss zusätzlich noch die Methode „ShowChart“ ausgeführt werden.

```
Chart.Line(lightValues).ShowChart()
```

3. X Berechtigung
Das Programm hat keine Berechtigung, um ein Fenster zu öffnen. Deshalb muss mit dem Befehl „xhost +localhost“ dem localhost (der eigenen Computer) erlaubt werden, ein Fenster zu öffnen.

4. Asynchrone Ausführung
Mit dem Code bis an hin konnte ein Fenster geöffnet werden, und es wurde auch ein Chart angezeigt. Dieses konnte jedoch nicht mehr geschlossen werden, weil der Thread,

4 **FSharp_Charting_Point_and_Line_Charts_2016-06-18**.

auf welchem das Programm läuft, vom Chart besetzt war. Deshalb musste das Skript asynchron ausgeführt werden.

Das Resultat

```
#I "/usr/lib/cli/gtk-sharp-2.0/"
#I "/usr/lib/cli/gdk-sharp-2.0/"
#I "/usr/lib/cli/atk-sharp-2.0/"
#I "/usr/lib/cli/glib-sharp-2.0/"
#I "bin/Debug/"
#I "../packages/Rx-Linq.2.2.5/lib/net40/"

#r "gtk-sharp.dll"
#r "gdk-sharp.dll"
#r "atk-sharp.dll"
#r "glib-sharp.dll"
#r "FSharp.Charting.Gtk.dll"
#r "FSharp.Data.DesignTime.dll"
#r "FSharp.Data.dll"
#r "FSharp.Control.Reactive.dll"
#r "System.Reactive.Linq.dll"

#load "../packages/FSharp.Charting.Gtk.0.90.14/FSharp.Charting.Gtk.fsx"

open FSharp.Charting
open System
open System.Threading
open FSharp.Data

type SensorData = CsvProvider<"SensorData-L.csv", ";">

let chunk size list =
    let chunkedValues = list |> Seq.chunkBySize size
    chunkedValues |> Seq.map (fun r -> r |> Seq.averageBy (fun e -> e |> float))

let format (sensorData:SensorData) =
    let values = sensorData.Rows |> Seq.map (fun r -> r.SensorData)
    chunk 1000 values

let runApp = async{
    let light = SensorData.Load("SensorData-L.csv")
    let lightValues = format light
    let chart = Chart.Line(lightValues).ShowChart()
    Gtk.Application.Run()
    ignore 0
    }

let runMain  = async{

```

```
45      printfn "Starting application"
46      let! app = Async.StartChild runApp
47
48      printfn "Press any key to exit..."
49      let n = Console.ReadKey()
50      Gtk.Application.Quit()
51      printfn "Bye..."
52      }
53
54  // run the whole workflow
55  Async.RunSynchronously runMain
56
57  //1. xhost +localhost
58  //2. fsharpi Program.fsx
```

KAPITEL 5

Schlusswort

In diesem Kapitel wird ein Fazit zu dieser Seminararbeit gezogen und eine Reflexion über die Arbeit und das bearbeitete Themengebiet gemacht.

5.1 Fazit

Im Rahmen dieser Arbeit konnten wir nicht alle Ziele erreichen, welche wir uns ursprünglich vorgenommen hatten. Der Grund dafür lag vor allem an den eingesetzten Technologien. Die konkreten Punkte werden in den nachfolgenden Abschnitten kurz beschrieben.

5.1.1 Sensordaten sammeln

Ursprünglich sollte der Teilbereich „Sensordaten sammeln“ nur einen kleinen Teil der gesamten Projektarbeit ausmachen. Der grösste Teil der Arbeit sollte für den BigData-Teil, sprich die Aufbereitung, Aggregierung und Auswertung der gesammelten Sensordaten, aufgewendet werden.

Relativ rasch hat sich gezeigt, dass die Anbindung der Raspberry Pi GrovePi Sensoren mit einer Sprache des .NET Frameworks nicht so einfach zu bewerkstelligen ist. Die vorhandene .NET-Library für den Zugriff auf das Sensorboard erforderte das neue .NET Core Framework. Aufgrund der geringen Dokumentationen und Erfahrungen mit dem neuen .NET Core Framework hat es viel Aufwand gekostet herauszufinden, dass wir die Library unter Linux 32-Bit aktuell nicht verwenden können.

Aufgrund dessen mussten wir eine andere Möglichkeit finden, um die Sensordaten abzugreifen. Am Ende waren wir mit Hilfe von zwei allgemeineren Raspberry Pi .NET Libraries in der Lage über den I2C-Bus auf das Sensorboard zuzugreifen. Die finale Lösung wurde mit Hilfe dieser .NET Libraries, einem C# Wrapper und einem F# Programm realisiert. mehr?

5.1.2 Sensordaten aufbereiten und auswerten

Der erste Schritt bei der Aufbereitung und Auswertung war die Visualisierung in einem Chart....

Erste probleme mit Packes / GTK...

impl

5.2 Reflexion

Das gewählte Themengebiet war für uns sehr spannend und interessant, da es sich noch um ein relativ neues Themengebiet und eine erst aufkommende Kombination von Technologien handelt. F# kann man nicht mehr unbedingt als „junge Programmiersprache“ bezeichnen (Erscheinungsjahr: 2002), jedoch ist die Verwendung unter einem Nicht-Microsoft / Windwos-System noch nicht so verbreitet. Vor eine grössere Herausforderung hat uns dann erst die Verwendung einer Libary, welche das neue .NET Core Framework benötigt, gestellt. Aufgrund der noch fehlenden, unvollständigen und teilweise bereits wieder veralteten Dokumentationen mussten wir viel Zeit und Energie auf Recherchen und Analysen in diesem Bereich verwenden.

Zusätzlicher Abschnitt

Trotz allem war es für uns sehr interessant ein Projekt mit solchen neuen und innovativen Technologien und Technologiekombinationen zu realisieren.

Quellenverzeichnis

[Ins] (Siehe S. 12).

[Win] *Develop Windows 10 IoT apps on Raspberry Pi 3 and Arduino - Windows IoT* (siehe S. 6).

[Fsh] *F# Data: Library for Data Access* (siehe S. 7).

[Fro] *FrontPage - Raspbian* (siehe S. 6).

[Mon] *Home | Mono* (siehe S. 6).

[Mas15] Massi, Beth: *Understanding .NET 2015*. EN. Microsoft Inc. 2015. URL: https://blogs.msdn.microsoft.com/bethmassi/2015/02/25/understanding-net-2015/.

[Edoa] *.NET Core*. EN. .NET Foundation. URL: https://www.dotnetfoundation.org/netcore.

[Ewi] *.NET Core*. DE. Wikimedia Foundation. 2016. URL: https://de.wikipedia.org/wiki/.NET_Core.

[Nug] *NuGet Gallery | GrovePi for Windows IoT 1.0.6* (siehe S. 7, 9).

[Rasa] *Raspberry Pi - Sensor* (siehe S. 3).

[Rasb] *Raspberry Pi - Teach, Learn, and Make with Raspberry Pi* (siehe S. 3).

[Rasc] *Raspberry Pi 3 on sale now at $35 - Raspberry Pi* (siehe S. 4).

[Rasd] *Raspberry Pi: Die Erfolgsgeschichte des Scheckkarten-Computers | Gründerszene* (siehe S. 3).

[Gro] *Raspberry Pi Internet of Things Kit* (siehe S. 7).

[Noo] *Raspberry Pi NOOBS Setup* (siehe S. 12).

[Edob] *What is the .NET Core Platform?* EN. .NET Foundation. URL: https://dotnet.github.io/docs/getting-started/what-is-dotnet.html.

[Rpi] *What the Raspberry Pi's Rainbow Boot Screen and Rainbow Box Mean* (siehe S. 12).

Abbildungsverzeichnis