

Embedded Frontend-Entwicklung mit .Net Core und Blazor

William Mendat

BACHELORARBEIT

zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)

Studiengang Angewandte Informatik

Fakultät Elektrotechnik, Medizintechnik und Informatik
Hochschule für Technik, Wirtschaft und Medien Offenburg

28.02.2022

Durchgeführt bei der Firma Junker Technologies

Betreuer

Prof. Dr.-Ing. Daniel Fischer, Hochschule Offenburg

M. Sc. Adrian Junker, Junker Technologies

Mendat, William:

Embedded Frontend-Entwicklung mit .Net Core und Blazor / William Mendat. –
BACHELORARBEIT, Offenburg: Hochschule für Technik, Wirtschaft und Medien Offenburg, 2022. 38 Seiten.

Mendat, William:

Embedded Frontend-Development with .Net Core and Blazor / William Mendat. –
BACHELOR THESIS, Offenburg: Offenburg University, 2022. 38 pages.

Vorwort

—...—

Eidesstattliche Erklärung

Hiermit versichere ich eidesstattlich, dass die vorliegende Bachelor-Thesis von mir selbstständig und ohne unerlaubte fremde Hilfe angefertigt worden ist, insbesondere, dass ich alle Stellen, die wörtlich oder annähernd wörtlich oder dem Gedanken nach aus Veröffentlichungen, unveröffentlichten Unterlagen und Gesprächen entnommen worden sind, als solche an den entsprechenden Stellen innerhalb der Arbeit durch Zitate kenntlich gemacht habe, wobei in den Zitaten jeweils der Umfang der entnommenen Originalzitate kenntlich gemacht wurde. Ich bin mir bewusst, dass eine falsche Versicherung rechtliche Folgen haben wird.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Mannheim öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Offenburg, 28.02.2022

William Mendat

Zusammenfassung

Embedded Frontend-Entwicklung mit .Net Core und Blazor

In dieser Abschlussarbeit wird eine .Net Core Blazor Anwendung implementiert, um somit einen Ersatz, für die derzeitige GUI-Entwicklung auf einem Open Embedded System zu kreieren. Dabei werden verschiedene Ansätze implementiert, sowohl auf dem Raspberry Pi als auch auf einem externen Server, um einen aussagekräftigen Vergleich zwischen den hier verschiedenen Technologien zu schaffen. Ein Teil dieser Ausarbeitung besteht darin, eine Laufzeitanalyse zu erstellen.

Die Arbeit gibt zunächst einen Einblick in den momentanen Stand der Technik, wie bislang eine GUI für eine Open Embedded System implementiert wurde, um dann zu veranschaulichen, wie dass gleiche Verhalten mit .Net Core und Blazor widergespiegelt werden kann. Am Ende dieser Arbeit wird ein aussagekräftiges Fazit darüber abgegeben, ob die Technologie Blazor für die Frontend-Entwicklung im Open Embedded Bereich zu gebrauchen ist.

Abstract

Embedded Frontend-Development with .Net Core and Blazor

Englische Version von Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Inhaltsverzeichnis

1. Einleitung	1
1.1. Aufgabenstellung	2
1.2. Verwendet Hardware	3
1.3. Verwendete Software	4
2. Stand der Technik	5
2.1. Embedded Systems	5
2.1.1. Hardware	6
2.1.2. Software	7
2.2. Qt	7
2.2.1. Was ist Qt?	8
2.2.2. Programmierbeispiel	9
2.2.3. Widgets	10
2.2.4. Signal und Slot Konzept	11
2.2.5. Raspberry Pi und Qt	11
3. Blazor	15
3.1. Was ist Blazor?	15
3.2. Architekturen	16
3.2.1. Blazor WebAssembly	16
3.2.2. Blazor Server	18
3.3. Komponenten	19
3.4. Javascript Interoperation	21
3.4.1. Javascript Runtime	21
3.4.2. Javascript Invokable	23
3.5. Blazor Maui	24
4. Raspberry Pi mit .Net Core und Blazor	26
4.1. Entwicklungsumgebung	26
4.2. Installation	27
4.3. Blazor Demo Anwendung	28
4.3.1. Erstellen des Projektes	28
4.3.2. Microsoft Iot	29
4.3.3. Raspberry Pi Daten Anzeigen	30

4.3.4. LED-Matrix Ansteuern	32
5. Analyse	36
6. Fazit	37
7. Ausblick	38
Abkürzungsverzeichnis	
Tabellenverzeichnis	i
Abbildungsverzeichnis	ii
Quellcodeverzeichnis	iii
Literatur	iv
A. Ein Anhang	vi

1. Einleitung

Heutzutage ist die Menschheit darauf fokussiert, die komplette Welt zu digitalisieren. Dabei existiert ein Grundsatz, alles, was digitalisiert werden kann, soll digitalisiert werden. Um dies zu realisieren, ist es von Nöten, überall Hardware und Software zu verbinden. Sei es nun das Handy, mit welchem durch nur einem klick die Bankdaten angezeigt werden können, oder ein selbstfahrendes Auto, welches einen Anwender selbstständig zum Ziel fährt. Dies sind nur zwei Beispiele von einer unendlich langen Liste. Hinter diesen technischen Wundern stecken meist mehrere Tausend kleiner Mikrocomputern und Mikrocontrollern, die dann mittels Software zusammen interagieren. Die Kombination dieser zwei Komponenten werden durch den Oberbegriff „Embedded System“ oder auch zu Deutsch ein „Eingebettetes System“ definiert.

Embedded Systems können dabei grundsätzlich zwischen zwei Plattformen unterschieden werden:

- Deeply Embedded System
- Open Embedded System

Deeply Embedded Systems sind die wesentlichen Bausteine des Internet of Things [1]. Die Anwendung, die bei Deeply Embedded Systems implementiert wird, basiert auf speziell angepassten Echtzeitbetriebssystemen, den Programmiersprachen C oder C++ und ganz speziellen GUI¹-Frameworks wie zum Beispiel TouchGFX.

Anders als bei den Deeply Embedded Systems, die sehr auf speziellen Technologien aufbauen, bieten Open Embedded Systems eine höhere Flexibilität in sachen Technologien an. Dem Programmierer ist also die Möglichkeit gegeben, unterschiedliche Technologien, als auch unterschiedliche Programmiersprachen zu verwenden. Dort

¹Graphical User Interface

gilt bis dato die Kombination von C++ und Qt für GUI-lastige Systeme als „State of the Art“.

Die Konstellation zwischen C++ und Qt hat bislang auch funktioniert, jedoch kommt dieser Ansatz auch mit Problemen mit sich, denn die höheren Entwicklungszeiten für die Entwicklung von C++ Anwendungen, sowie die geringe Verfügbarkeit von Experten auf dem Arbeitsmarkt sorgen für schlechtere Qualität und längere Produktionszeiten.

Um diesen Problemen zu entgegnen, soll in dieser Abschlussarbeit ein anderer Ansatz betrachtet werden. Und zwar könnten sowohl die Anwendungsschicht als auch die Persistenzschicht als .Net Core Anwendungen implementiert werden. Als GUI-Technologie soll dabei die neue Microsoft-Technologie namens *Blazor* als Qt Ersatz zum Einsatz kommen. Somit kann erreicht werden, die komplette Codebasis mit .Net Core auszutauschen und eine Programmier-freundlichere Umgebung für Entwickler im Open Embedded Systems zu erschaffen.

1.1. Aufgabenstellung

Das Ziel dieser Arbeit ist die Entwicklung eines Blazor-basierten Frontend auf einem Raspberry Pi 4B um einen aussagekräftigen Vergleich zwischen den Technologien schaffen zu können und um eine mögliche verdrängung mittels Blazor zu demonstrieren. Dazu soll zunächst begutachtet werden, wie der momentane Stand der Technik für Open Embedded Systems ist, um anschließend die Codebasis auf .Net Core und Blazor zu wechseln. Insbesondere sollen dabei verschiedene Aspekte, wie zum Beispiel das Verhalten zur Laufzeit, dieses Ansatzes überprüft werden.

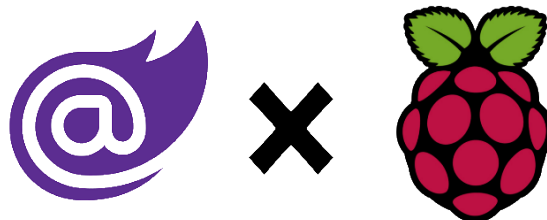


Abbildung 1.1: Blazor mit Raspberry Pi

1.2. Verwendet Hardware

Als Zielplattform für diese Thesis dient ein Raspberry Pi 4B. Dieser wird unter anderem deswegen verwendet, da es im Labor Embedded Systems 2 der Hochschule Offenburg verwendet wird, aber auch, da es sehr gut im Open Embedded Systems bereich eingesetzt werden kann. Der Raspberry Pi 4B verfügt dabei, unter anderem, über die folgenden technischen Spezifikationen: [2]

- 1,5 GHz ARM Cortex-A72 Quad-Core-CPU
- 1 GB, 2 GB oder 4 GB LPDDR4 SDRAM
- Gigabit LAN RJ45 (bis zu 1000 Mbit)
- Bluetooth 5.0
- 2x USB 2.0 / 2x USB 3.0
- 2x microHDMI (1x 4k @60fps oder 2x 4k @30fps)
- 5V/3A @ USB Typ-C
- 40 GPIO Pins
- Mikro SD-Karten slot

Und wird mit dem *Raspbian Buster with desktop* auf der SD-Karte betrieben.

Um noch mehr Funktionalität aufbringen zu können, wurde auf dem Raspberry Pi ein *RPI SENSE HAT Shield* aufgesteckt. Mit hilfe des *RPI SENSE HAT Shield* können dann unter anderem Daten wie zum Beispiel die momentane Temperatur oder auch die momentane Luftfeuchtigkeit gewonnen werden. Zudem ist auf dem SENSE HAT noch eine 8x8 LED-Matrix enthalten und ein Joystick mit 5 knöpfen, die angesteuert werden können.

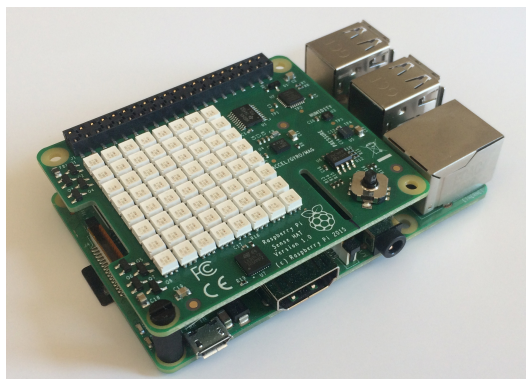


Abbildung 1.2: Verwendeter Raspberry Pi

1.3. Verwendete Software

Im Rahmen dieser Thesis werden die folgenden Softwaretools zur Entwicklung eingesetzt.

- Um eine Visualisierung des Images *Raspbian Buster with desktop* von dem Raspberry Pi zu erhalten, wird die Windows Desktop Anwendung *VNC Viewer* verwendet. Dadurch ist die Möglichkeit gegeben, bequem und einfach den Raspberry Pi über einen Bildschirm zu bedienen.
- Für die Demonstration des Kapitels *Stand der Technik* wird eine beispielhafte grafische Oberfläche mithilfe des Qt-Creators implementiert.
- Den Zugriff auf die Funktionalitäten des *RPI SENSE HAT Shield* wird mittels der *RTIMULib* Bibliothek realisiert.
- Eine ausschlaggebende Technologie dieser Thesis wird durch, dass Framework *Blazor* von Microsoft abgebildet.
- Die Programmiersprachen dieser Thesis werden sich hauptsächlich aus C++ und C# beziehungsweise .Net Core zusammensetzen.
- Um den Zugriff auf die Funktionalitäten des *RPI SENSE HAT Shield* mittels .Net zu gewährleisten, wird die Iot Bibliothek von Microsoft verwendet.
- Die Implementierung der Blazor Anwendung wird dann letztendlich mittels der kostenlosen IDE *Visual Studio Code* realisiert.
- Um zwischen dem Host Rechner und dem Raspberry Pi Dokumente und Ordner auszutauschen, wird die Desktop-Applikation *WinSCP* verwendet.

Die oben vorgestellten Tools und Programmiersprachen wurden nicht explizit vorgegeben, sind dementsprechend auch selbst ausgesucht und sind zu Beginn dieser Thesis auch schon alle komplett eingerichtet.

2. Stand der Technik

Dieses Kapitel soll ein grundlegendes Verständnis wiedergeben, wie der momentane Stand der Technik aussieht. Dabei soll zunächst betrachtet werden, wie die momentane Entwicklung bei Open Embedded Systems vonstattengeht, um anschließend eine beispielhafte Anwendung zu implementieren.

2.1. Embedded Systems

Ein Embedded System oder auf Deutsch ein eingebettetes System, wird als eine integrierte, mikroelektronische Steuerung angesehen, welches meist darauf ausgelegt ist eine spezifische Aufgabe zu erledigen. Dabei setzt sich ein Embedded System auch aus dem Zusammenspiel zwischen Hardware und Software zusammen. Solche Embedded Systems haben meist kein ausgeprägtes Benutzerinterface und können weitergehend in die zwei unterklassen, Open und Deeply Embedded Systems unterteilt werden.

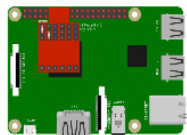
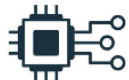
Open Embedded	Deeply Embedded
	
32/64 Bit Multicore Prozessor	8/16 Bit Singlecore Prozessor
Systemsoftware Applikation	Monitorprogramm Applikation
Für komplexe Aufgaben gedacht	Für einfache Aufgaben gedacht

Abbildung 2.1: Open vs Deeply Embedded Systeme [3][vgl.]

Neben der logischen Korrektheit, die Embedded Systeme an den Tag legen müssen, lassen sie sich durch eine Reihe unterschiedlicher Anforderungen und eigenschaften von den heutzutage üblichen Anwendungen abgrenzen. Unter anderem wird bei Embedded Systems ein sogenanntes *Instant on* gefordert, welches besagt, dass das Gerät unmittelbar nach dem Einschalten betriebsbereit sein muss [3][vgl.].

Weitere Anforderungen die bei Embedded Systemen zustande kommen, sind in der Folgenden Tabelle abgebildet:

Anforderung	Beschreibung
Funktionalität	Die Software muss schnell und korrekt sein
Preis	Die Hardware darf nicht zu kostenspielig sein
Robustheit	Muss auch in einem rauen Umfeld funktionieren
Fast poweroff	Muss in der Lage sein schnell das komplette System Abzuschalten
Räumliche Ausmaße	Muss klein sein, um sich in ein System einbinden zu können
Nonstop-Betrieb	Muss in der Lage sein, im Dauerbetrieb laufen zu können
Lange Lebensdauer	Muss in der Lage sein, teilweise mehr als 30 Jahre zu laufen ohne groß zu verschlechtern

Tabelle 2.1.: Anforderungen an Embedded Systems [3]

2.1.1. Hardware

Die einzelnen Komponenten, die in einem Embedded Systems verbaut worden sind, entscheiden über die vorhandene Leistung, den Stromverbrauch und die Robustheit, die dieses System ausmachen. Die kern Komponente eines Embedded System wird durch einen Prozessor repräsentiert und wird häufig als *System on Chip* eingebaut. Dabei ist die Tendenz zu den ARM-Core Modellen steigend. Neben dem Prozessor befinden sich typischerweise auf den Embedded Systems noch weitere Komponenten, wie zum Beispiel dem Hauptspeicher, dem persistenten Speicher und diversen Schnittstellen, um weitere peripherie Geräte anzuschließen [3][vgl.].

Damit weitere peripherie Geräte angeschlossen werden können, müssen mittels einigen Leitungen, digitale Signale übertragen werden. Aufgrund dessen, dass Leitungen typischerweise nur über eine begrenzte Leistung verfügen, werden Treiber eingesetzt, um peripherie Geräte zu verbinden. Nicht selten kommt es vor, dass in

einem Embedded System darüber hinaus noch eine galvanische Entkopplung eingebaut wird, damit die Hardware vor Störungen von außen, wie Beispiel Motoren, geschützt ist [3][vgl.].

2.1.2. Software

Genauso, wie sich Embedded Systems in zwei Bereiche unterscheiden lassen können, kann die Software eines Embedded System in Systemsoftware und funktionsbestimmende Anwendungssoftware unterteilt werden. Da *Deeply Embedded Systems* meist nur für kleine und einfach Aufgaben ausgelegt sind, benötigen diese meist nur sehr schwache Software. Diese funktionsbestimmende Anwendungssoftware ist dann meist ein spezielles Echtzeitbetriebssystem, welches auf die Aufgabe und der Hardware angepasst ist.

Ganz anders sieht es im *Open Embedded Systems* bereich aus, welche seine Software als Kennzeichen hat. Da diese für sehr komplexe Aufgaben zum einsatz kommen, kommt es nicht selten vor, dass auch eine GUI für ein solches System vonnöten ist. Deswegen basiert ein *Open Embedded Systems* auf einer Systemsoftware, die dem Programmierer mehr Möglichkeiten beim Entwickeln gibt. Unter anderem ist somit auch die Möglichkeit gegeben, die Programmiersprache und das GUI-Framework, nach Belieben selbst auszusuchen und nicht auf spezielle Software angewiesen zu sein [3][vgl.].

2.2. Qt

In der vorherigen Sektion wurde ein allgemeines Bild dargestellt, was ein Embedded System ist und in welchen Varianten diese auftauchen. Weitergehend soll nun, in dieser Sektion, vorgestellt werden, wie Üblicherweise im *Open Embedded Systems* bereich programmiert wurde.

Ein großer Anteil eines *Open Embedded Systems* wird heutzutage durch die GUI repräsentiert. Die GUI sollte intuitive und zuverlässig sein. Zudem ist es auch noch enorm wichtig, dass die GUI, wenige Ressourcen verbraucht, schnell reagiert und

einfach einzubinden ist. Um diese Anforderung zu erreichen, wurde bis dato *Qt* in Kombination mit der Programmiersprache C++ verwendet [4][vgl.].

2.2.1. Was ist Qt?

Qt ist ein Framework zum Erzeugen von GUI's auf mehreren Betriebssystemen. Es wurde 1990 von *Haarvard Nord* und *Eirik Chambe-Eng* unter dem Vorwand entwickelt, benutzer freundliche GUI's mithilfe der Programmiersprache C++ zu entwickeln. Der Name *Qt* entstand dabei, da Haarvard den Buchstaben *Q* in Emacs als sehr schön empfand. Zudem entstand das *t* in Qt als abkürzung für das englische Wort *Toolkit* [5][vgl.].

Die Intension hinter Qt war es damals nicht nur ein Framework zu erschaffen, welches benutzer freundliche GUI's kreiert, sondern dies auch, mit nur einer Code-Basis über alle Betriebssysteme hinweg. Die schwierigkeit bestand also darin, das selbe Aussehen, die selbe benutzer Erfahrung und die gleiche Funktionalität über die verschiedenen Betriebssysteme zu schaffen [6][vgl.].

Qt ist in C++ entwickelt worden und verwendet zusätzlich noch einen Compiler welcher *moc*¹ genannt wird, womit C++ um weitere Elemente erweitert wird. Der daraus compilierte Code folgt dem C++-Standard und ist somit mit jedem anderen C++ Compiler kompatibel. Obwohl Qt ursprünglich dafür gedacht war, rein auf C++ zu basieren, wurden im Laufe der Zeit mehrere Erweiterungen für Qt von der Community entwickelt, um Qt mit mehreren Programmiersprachen benutzen zu können, wie zum Beispiel Python oder Java.



Abbildung 2.2: Qt Logo

¹meta object compiler

2.2.2. Programmierbeispiel

Das Programmierbeispiel welches im Folgenden dargestellt wird, erzeugt ein Fenster mit dem Titel *Meine erste Qt App*, ein Label welches *Hello World* anzeigt und ein Button mit der Aufschrift *Exit*. Sobald der Button oder die Tastenkombination *Alt + E* gedrückt wird, schließt sich das Fenster.

```
1 #include "mainwindow.h"
2
3 int main(int argc, char *argv[])
4 {
5     // Set the Application
6     QApplication app(argc, argv);
7     QWidget window;
8     // Set fixed Width of the Window
9     window.setFixedWidth(300);
10
11    // Set the Title of the Window
12    window.setWindowTitle("Meine erste Qt App");
13
14    // Create a Label and align it to Center
15    QLabel *lblHello = new QLabel("Hello World!");
16    lblHello->setAlignment(Qt::AlignCenter);
17
18    // Create a Button and connect it to close the Window
19    QPushButton *btnExit = new QPushButton("&Exit");
20    // Connect button with the App
21    QObject::connect(btnExit, SIGNAL(clicked()), &app, SLOT(quit()));
22
23    // Sowohl das Label als auch die Schaltfläche vertikal ausrichten
24    QVBoxLayout *layout = new QVBoxLayout;
25
26    // Add the Widgets
27    layout->addWidget(lblHello);
28    layout->addWidget(btnExit);
29    window.setLayout(layout);
30
31    window.show();
32    return app.exec();
33 }
```

Listing 2.1: Qt Hello World Sourcecode

Das Programm welches erzeugt wird, wird folgend dargestellt:

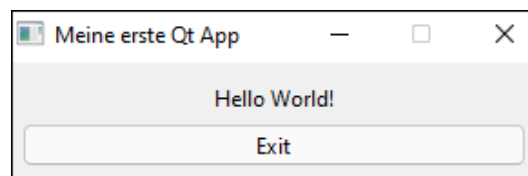


Abbildung 2.3: Qt Hello World App

2.2.3. Widgets

Um eine GUI gestalten zu können braucht es Komponenten, die auf der GUI angezeigt werden können. Qt verwendet dafür sogenannte *Widgets*. Widgets sind also grafische Komponenten, die dafür genutzt werden können, um die Benutzeroberfläche nach belieben zu gestalten. Ein Beispiel für eine solche Komponente wäre ein Button, welcher in der Sektion *Programmierbeispiel* als *btnExit* vorkam.

Die Widgets, die Qt zur Verfügung stellt sind in einer großen Klassenhierarchie zusammengesetzt und diese Hierarchie könnte sich wie folgt vorgestellt werden:

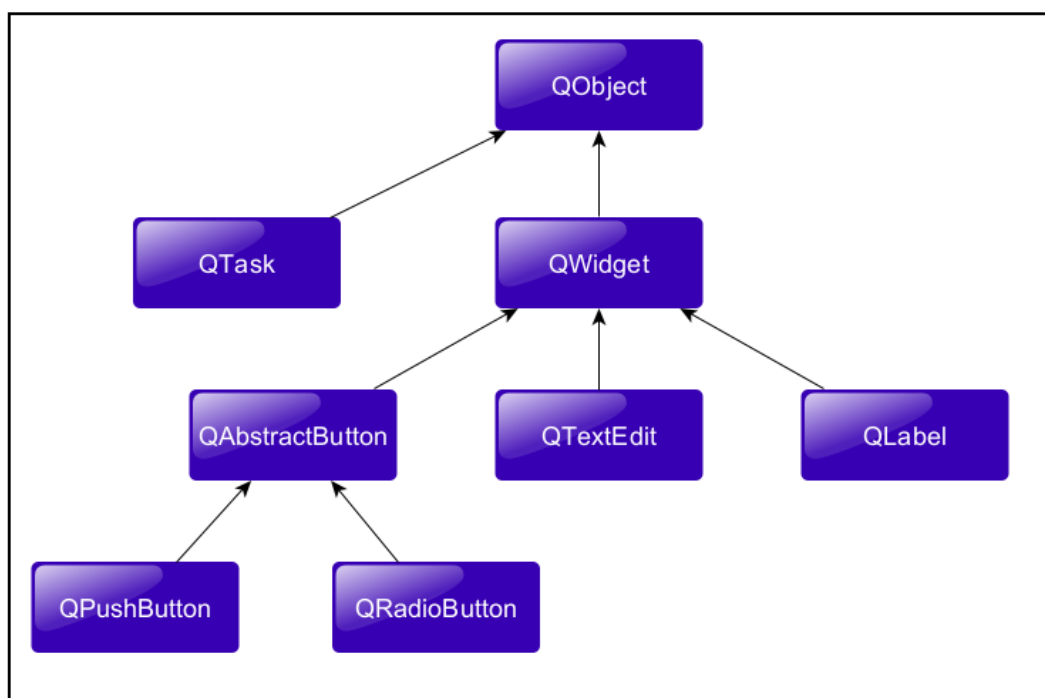


Abbildung 2.4: Qt Widgets Klassenhierarchie [6][vgl.]

Wie zu sehen ist, ist ganz oben in der Klassenhierarchie die `QObject` Klasse. Diese enthält unter anderem den Signal und Slot mechanismus, welcher später noch genau erklärt wird. Weitergehend, werden Widgets, die gemeinsame Funktionalitäten aufweisen zusammen gruppiert. Diese Verhalten ist bei `QPushButton` und `QRadioButton` erkennbar, denn beide Widgets sind Buttons, die sich Teilweise die gleichen Eigenschaften und Funktionen teilen [6][vgl.].

2.2.4. Signal und Slot Konzept

Um eine Benutzeroberfläche wirklich interaktiv zu gestalten, ist es vonnöten, auf bestimmte Ereignisse zu reagieren. Beispielsweise ist das betätigen eines Buttons ein Auslöser für ein Ereignis auf welches reagiert werden kann. Es gibt viele Methoden und Muster in welche ein solches Ereignis-Aktions Konzept implementiert werden kann.

Qt benutzt um diese Ereignis-Aktion Muster zu bewerkstelligen, das selbst entwickelte Signal und Slot Konzept. Ein *Signal* ist im einfachen sinne eine Nachricht, die Versendet wird, um zu signalisieren, dass der momentane Status eines Objektes sich geändert hat. Dahingegen ist ein *Slot* eine spezielle Funktion von einem Objekt, welche immer dann aufgerufen wird, wenn ein bestimmtes *Signal* gesendet wird [6][vgl.].

Damit jeder *Slot* auch weiß, auf welches *Signal* reagiert werden soll, müssen diese zusammen verbunden werden. In der Sektion *Programmierbeispiel* war folgende Zeile zu sehen:

```
1 // Connect button with the App
2 QObject::connect(btnExit, SIGNAL(clicked()), &app, SLOT(quit()));
```

Listing 2.2: Signal und Slot beispiel

In dieser Codezeile fand die Verbindung zwischen einem Signal und einem Slot statt. Dabei wurde sich mit dem Button *btnExit* verbunden und ein Signal gesendet, wenn der Button betätigt wird. Weitergehend wurde sich noch mit *app* verbunden, auf welchem dann, beim Betätigen von *btnExit*, die Funktion *quit* aufgerufen wird. Sobald also das Signal gesendet wird, schließt sich die Application.

Einer der größten Vorteile dieser Methode im gegensatz zu anderen Methoden, ist es, dass dadurch N zu M Beziehungen abgebildet werden können. Das bedeutet also, dass sich ein Signal mit beliebig viele Slots verbinden kann und dass sich ein Slot auf mehrere Signale verbinden kann [7][vgl.].

2.2.5. Raspberry Pi und Qt

Nachdem nun ein grober Überblick zu Qt geschaffen wurde, soll nun eine beispielhafte Anwendung mithilfe von Qt auf dem Raspberry Pi implementiert werden. Bei dieser Anwendung sollen beim Betätigen eines Buttons, Daten von dem Raspberry

Pi gelesen werden und auf der GUI angezeigt werden. Dieses Beispiel ist von dem Embedded Systems 2 Labor inspiriert.

RTIMULib

Die RTIMU Bibliothek, ist eine Bibliothek, die anfänglich dazu gedacht war, mithilfe von C++ oder Python, einfach Daten von einem Open Embedded System zu lesen. Miteiweile existiert diese Bibliothek auch für andere Sprachen wie zum Beispiel C#.

Mithilfe dieser Bibliothek sollen für dieses Beispiel die Daten von dem Raspberry Pi gelesen werden, um diese dann anschließend auf der GUI anzeigen zu können. Um RTIMU in einem QT Projekt nutzen zu können, muss die Bibliothek zunächst in der *.pro* Datei eingebunden werden, dies geschieht, indem die Zeile *LIBS += -lRTIMULib* hinzugefügt wird.

Hilfsklasse

Nachdem nun die Bibliothek hinzugefügt wurde, soll eine Hilfsklasse, als repräsentation der Daten erzeugt werden, die wie folgt, aufgebaut ist:

```
1 class ReadData
2 {
3 private:
4     float m_Temperature = 0.1f;
5     float m_AirPressur = 0.1f;
6     float m_Humidity = 0.1f;
7     float m_xMagnetometer = 0.1f;
8     float m_yMagnetometer = 0.1f;
9     float m_zMagnetometer = 0.1f;
10
11     RTIMUSettings* m_RTIMUSettings = nullptr;
12     RTIMU* m_RTIMU = nullptr;
13     RTPressure* m_RTPressure = nullptr;
14     RTHumidity* m_RTHumidity = nullptr;
15
16 public:
17     ReadData();
18     void vRead(void);
19 };
```

Listing 2.3: RTIMU-Hilfsklasse

Die Klasse enthält neben dem im Listing 2.3 gezeigten Code, noch weitere Methoden um auf die Privaten *Member Variablen* zuzugreifen. Nun müssen im Konstruktor noch die Variablen konfiguriert und initialisiert werden, dies geschieht folgendermaßen:

```
1 ReadData::ReadData()
2 {
3     // Define variables
4     m_RTIMUSettings = new RTIMUSettings("RTIMULib");
5     m_RTIMU = RTIMU::createIMU(pRTIMUSettings);
6     m_RTPressure = RTPressure::createPressure(pRTIMUSettings);
7     m_RTHumidity = RTHumidity::createHumidity(pRTIMUSettings);
8
9     // Init
10    pRTIMU->IMUInit();
11    pRTIMU->setCompassEnable(true);
12    pRTPressure->pressureInit();
13    pRTHumidity->humidityInit();
14 }
```

Listing 2.4: RTIMU-Hilfsklasse-Konstruktor

Die letzte Komponente der Hilfsklasse, bildet die *vRead* Methode ab, in der dann schlussendlich die Daten, wie folgt, gelesen werden:

```
1 void ReadData::vRead(void)
2 {
3     if (m_RTIMU->IMURead())
4     {
5         RTIMU_DATA RTIMUData = m_RTIMU->getIMUData();
6         m_RTPressure->pressureRead(RTIMUData);
7         m_RTHumidity->humidityRead(RTIMUData);
8
9         m_AirPressur = RTIMUData.pressure;
10        m_Temperature = RTIMUData.temperature;
11        m_fHumidity = RTIMUData.humidity;
12
13        RTIMUData.compass.normalize();
14        m_xMagnetometer = RTIMUData.compass.x();
15        m_yMagnetometer = RTIMUData.compass.y();
16        m_zMagnetometer = RTIMUData.compass.z();
17    }
18 }
```

Listing 2.5: RTIMU-Hilfsklasse-vRead

Benutzeroberfläche

Die Benutzeroberfläche in diesem Beispiel wurde schlicht gehalten. Sie besitzt insgesamt zwölf QtLabels wovon sechs dafür gedacht sind, um die Raspberry Pi Daten

abzubilden und einem Button, der die Daten abrufen soll.

Der Button wurde mit einem Slot versehen der aufgerufen wird, wenn der Button betätigt wird. In dem Slot wird die vRead Methode aufgerufen, um dann die Labels zu aktualisieren. Die Implementierung des Slots sieht dann folgend aus:

```
1 void MainWindow::on_pbUpdate_clicked()
2 {
3     m_readData->vRead();
4
5     double dValue = static_cast<double>(m_readData->getAirPressur());
6     ui->lblLuftdruck->setText(QString::number(dValue));
7
8     dValue = static_cast<double>(m_readData->getTemperature());
9     ui->lblTemperatur->setText(QString::number(dValue));
10
11    dValue = static_cast<double>(m_readData->getHumidity());
12    ui->lblLuftfeuchtigkeit->setText(QString::number(dValue));
13
14    dValue = static_cast<double>(m_readData->getMagnetometerX());
15    ui->lblKompassX->setText(QString::number(dValue));
16
17    dValue = static_cast<double>(m_readData->getMagnetometerY());
18    ui->lblKompassY->setText(QString::number(dValue));
19
20    dValue = static_cast<double>(m_readData->getMagnetometerZ());
21    ui->lblKompassZ->setText(QString::number(dValue));
22 }
```

Listing 2.6: Update Button Slot

Das vollständige Programm wird dann in Abbildung Open Embedded System GUI dargestellt.



Abbildung 2.5: Open Embedded System GUI

3. Blazor

Nachdem nun, in der vorherigen Sektion, das C++ Framework Qt vorgestellt wurde, soll in diesem Kapitel das Framework dieser Thesis vorgestellt werden, welches den Namen *Blazor* trägt.

3.1. Was ist Blazor?

Blazor ist ein Framework von Microsoft zum Erzeugen von Webseiten. Der Name *Blazor* entstand aus der Kombination der zwei Wörter *Browser* und *Razor*. Browser zum einen, da das Ziel war, C# in den Browser zu bekommen und Razor zum anderen, da Blazor gebrauch von der Razor syntax macht [8][vgl.]. Es wurde 2019 erstmalig von Microsoft veröffentlicht, mit der Intension Webseiten oder auch SPA's¹ mithilfe von C# zu entwickeln. Dabei existieren 2 Varianten von Blazor:

- Blazor Server
- Blazor WebAssembly

Der essenzielle Unterschied der beiden varianten besteht darin, dass Blazor Server auf einem Server gehostet wird und Blazor WebAssembly native im Browser läuft, dazu aber im späteren verlauf dieser Thesis mehr [9][vgl.].

Die Idee, die hinter dem Framework von Microsoft steckt, ist es nicht nur C# in den Browser zu bekommen, um somit Javascript zu verdrängen, sondern auch mit nur einer Codebasis sowohl im Frontend, als auch im Backend zu entwickeln. Somit wurde geschaffen, dass langjährige C# Entwickler mit ihrem vorhandenen Wissen als Full-Stack entwickler, eingesetzt werden können.

¹Single Page Applications

3.2. Architekturen

Da Blazor in zwei Varianten existiert, existieren dementsprechend auch zwei Architekturen für dieses Framework. Bevor die zwei Architekturen jedoch im Detail erklärt werden, sollte zuerst ein grundlegendes Wissen darüber veranschaulicht werden, wie bis jetzt andere SPA's, wie zum Beispiel Angular oder React sowas gehandhabt haben. Die heutigen SPA's basieren auf der Client-Server Architektur, das bedeutet der Client stellt eine Anfrage an den Server und erhält dann, sollte alles stimmen, die passende Antwort. Die Kommunikation der beiden Teilnehmer geschieht dann in den meisten Fällen über eine REST Api. Heutzutage gibt es noch andere Alternativen zu REST, wie zum Beispiel gRPC, jedoch ist bis heute der Standard für die Kommunikation noch REST. In dem folgenden Schaubild ist eine solche Architektur zu sehen:

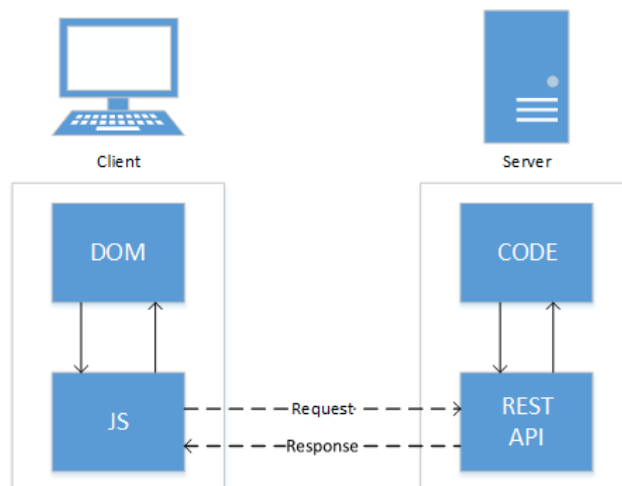


Abbildung 3.1: Client-Server Architektur mit Javascript

Wie zu sehen ist, kommuniziert der Client mithilfe von Javascript mit dem Server. Javascript holt sich also die Daten, welche der Client braucht und gibt diese dem DOM zum Verarbeiten weiter.

3.2.1. Blazor WebAssembly

Bei Blazor WebAssembly ist es so, dass sich die Architektur von den anderen SPA's wie Angular nicht groß verändert. Tatsächlich verändert sich hierbei nur die Programmiersprache, die auf dem Client läuft. Es handelt sich dabei um die Programmiersprache C#, die dann auf dem Client ausgeführt wird, wie im folgenden zu

sehen ist.

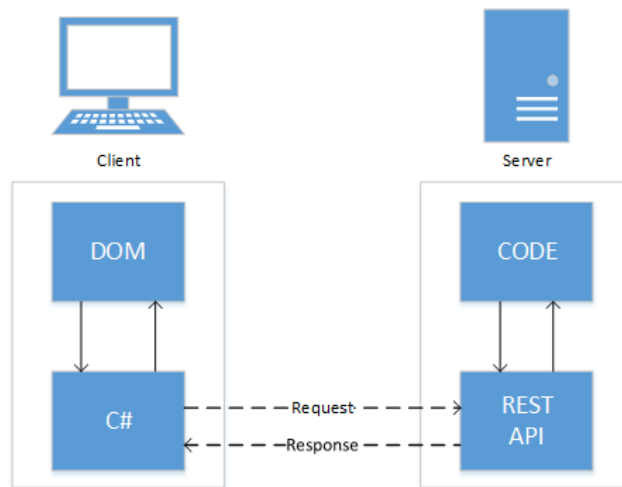


Abbildung 3.2: Blazor WebAssembly Architektur

Das ganze Konzept C# überhaupt auf dem Client laufen lassen zu können, funktioniert durch WebAssembly. WebAssembly wandelt Programmcode in nativen Bytecode um, der dann in einer Sandbox im Browser ausgeführt werden kann. Dabei wird von der Sandbox aus die DOM mit Hilfe von Javascript kontinuierlich manipuliert. Javascript verschwindet in dem Sinne also nicht komplett sondern wird lediglich ergänzt. Da für dieses Konzept also WebAssembly vonnöten ist, kann Blazor WebAssembly nicht auf Browsern funktionieren, die WebAssembly nicht unterstützen [8][vgl.].

Im folgenden werden noch die Vor- und Nachteile von Blazor WebAssembly dargestellt:

- + Sehr Skalierbar
- + Sehr Performant
- + Es kann komplett eigenständig auf dem Client laufen und ist nicht unbedingt auf dem Server angewiesen
- Große Anwendungsdatei, die komplett auf dem Client geladen werden muss
- Lange Ladezeit beim ersten Aufruf
- Kompletter Code ist auf dem Client zu sehen

3.2.2. Blazor Server

Anders als es bei Blazor WebAssembly der Fall ist, wird bei Blazor Server nicht C# in den Browser geladen, sondern das ganze Konzept spielt sich auf dem Server ab. Deswegen wird auch die Web-Anwendung als SPA auf dem Server gerendert. Zum Client werden nur JavaScript und Markup gesendet und die Daten und Benutzereingaben laufend mittels SignalR zwischen Client und Server ausgetauscht. Dementsprechend ist es auch vonnöten, dass immer zwischen dem Client und dem Server eine offene Verbindung vorhanden ist [8][vgl.].

Dadurch dass die komplette Seite auf dem Server gerendert wird, lädt die Seite auf dem Client sehr schnell, muss nur eine kleine Javascript Datei auf den Client laden und kann auf sehr Leistungsschwachen Clients verwendet werden. Zudem kommt auch noch, anders als bei Blazor WebAssembly, dass sich jeder Browser verwenden lässt unabhängig davon ob dieser WebAssembly unterstützt oder nicht.

Das ganze Konzept dieser Architektur, baut drauf auf, dass beim ersten Aufruf der Seite eine Javascript Datei names *Blazor.js* auf dem Client geladen wird. Diese Datei kommuniziert dann mit dem DOM und baut die Verbindung zum Server auf. Sobald die Verbindung aufgebaut ist, werden kontinuierlich Nachrichten zwischen Client und Server ausgetauscht, wie im folgenden zu sehen ist:

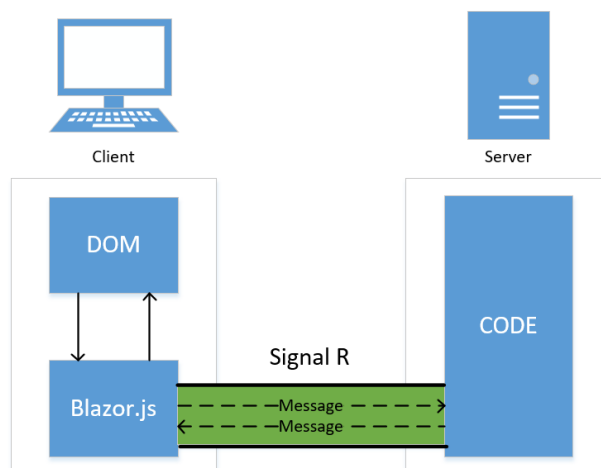


Abbildung 3.3: Blazor Server Architektur

Im folgenden werden noch die Vor- und Nachteile von Blazor Server zusammengestellt:

- + Kurze Ladezeiten

- + Es ist komplett Browserunabhängig
- + Client hat keinen Zugriff auf den SourceCode
- Nicht Skalierbar, da alle Benutzer auf einem Server zugreifen
- Lange Netzwerklatenz führt zu Verzögerungen in der Benutzeroberfläche
- Es muss immer eine Verbindung bestehen

3.3. Komponenten

Anders als es bei Qt der Fall ist, basiert Blazor wie andere Web-Frameworks auf *Html* und *Css*. Das bedeutet, dass der Entwickler auf jedes *Html* element zugreifen kann, das existiert. Weitergehend hat der Entwickler noch die möglichkeit auf zusätzliche Infragistics wie zum Beispiel *MadBlazor* oder *Ignite UI* zurückzugreifen.

Außerdem bietet Blazor zusätzlich noch die Möglichkeit eigene Komponenten zu erstellen. Eine Komponente, im Sinne von Web-Frameworks, kann sich so wie eine Methode in einer Programmiersprache vorgestellt werden, das heißt, es kann eine *Html* und *Css* Sequenz ausgelagert werden, um diese an mehrere Stellen zu verwenden. Hinzukommt kann eine Komponente, in Blazor, in zwei Varianten vorkommen. Einmal als eine *Page-Komponente* und einer *Non-Page-Komponente*. Der Unterschied zwischen den beiden ist, dass eine *Page-Komponente* durch eine *URL* adressieren kann, das heißt, dass diese Komponente als eigenständige Seite einer Webseite angesehen werden kann und eine *Non-Page-Komponenten*, wirklich nur die Sequenz von *Html* und *Css* darstellt [10][vgl.].

```
1 @page "/counter"
2
3 <PageTitle>Counter</PageTitle>
4
5 <h1>Counter</h1>
6
7 <p role="status">Current count: @currentCount</p>
8
9 <button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
10
11 @code {
12     private int currentCount = 0;
13
14     private void IncrementCount()
15     {
16         currentCount++;
```

```
17     }  
18 }
```

Listing 3.1: Page-Komponente Beispiel

Listing 3.1 zeigt eine Page-Komponente, die sowohl einen Text und einen Button anzeigt und den momentane wert von *currentCount* um eins erhöht wenn der Button betätigt wird. Bei der Komponente handelt es sich um eine Page-Komponenten, da dies mit dem *@page* in der ersten Zeile ausgezeichnet ist.

Einer Komponente, sei es nun eine Page-Komponente oder Non-Page-Komponente, können auch einen oder mehrere Parameter von der Oberkomponente mitgegeben werden. Dies hat den Vorteil, dass Komponenten mehr an Flexibilität gewinnen und somit besser wiederverwendbar sind, wie im folgenden Beispiel zu sehen ist:

```
1 <h2>@Title</h2>  
2  
3 @code {  
4     [Parameter]  
5     public string Title { get; set; }  
6 }
```

Listing 3.2: Kind Komponente

```
1 @page "/parent"  
2  
3 <h1>Parent</h1>  
4  
5 <Child Title="Child-Title"/>  
6  
7 @code {  
8  
9 }
```

Listing 3.3: Eltern Komponente

Wie in Listing 3.2 zu sehen ist, hat diese Komponente einen Parameter, die sie dann, wie hier im Beispiel, als Title in einem *<h2>-Tag* darstellt. Listing 3.3 kann dann gebrauch von der Kind Komponente machen und ihr einen Title mitgeben. Der Html Tag der Kind Komponente, wird durch den Dateinamen der Komponente erzeugt, das bedeutet, da die Komponente *Child.razor* im Projekt heißt, wird diese dann auch mittels dem *<Child>-Tag* verwendet.

3.4. Javascript Interoperation

Durch Blazor ist nun also die Möglichkeit gegeben, eine Webanwendung komplett mit C# zu schreiben. Jedoch ist diese Technologie noch sehr neu und verfügt derzeit noch nicht über die bandbreite an Funktionalitäten, wie Javascript es tut. Deswegen kann nicht pauschal gesagt werden, dass Blazor nun Javascript komplett ersetzt und es auch in Blazor eine Möglichkeit geben muss, mit javascript code zu hantieren. Aufgrund dessen existiert in Blazor ein Mechanismus namens *Javascript Interoperation*, den es einem ermöglicht, von C# Javascript Methoden aufzurufen und von Javascript C# Methoden aufzurufen.

3.4.1. Javascript Runtime

Mithilfe des Javascript Runtime Interfaces, lassen sich Javascript Methoden aus dem C# Code aufrufen. In dem Interface befindet sich eine Methode namens *InvokeAsync*, die jedoch in zwei Varianten zur Verfügung steht, wie im folgenden zu sehen ist:

- `ValueTask<TValue> InvokeAsync<TValue>(string, object?[]?)`
- `ValueTask<TValue> InvokeAsync<TValue>(string, CancellationToken, object?[]?)`

Beide Varianten tun dass gleiche, mit dem einzigen Unterschied, dass bei der zweiten Variante zusätzlich noch eine *CancellationToken* mitgegeben werden kann, um den Vorgang manuell abubrechen [11][vgl.].

Des Weiteren gilt für die Verwendung des Javascript Runtime Interface folgendes:

- Der *string* Parameter in *InvokeAsync*, steht für den Javascript Methodennamen
- Der *object?[]?* Parameter ist ein Array von objekten, die dazu genutzt werden um der Javascript Methode, die Parameter zu übergeben
- *TValue* ist ein Template Objekt, welches dann als Rückgabewert benutzt werden kann, zudem muss das Objekt mithilfe von *JSON* serialisierbar sein
- Der Javascript code muss unter dem Verzeichnis *wwwroot* vorhanden sein
- Um von diesem Mechanismus gebrauch machen zu können muss in die Komponente, das Interface *IJSRuntime injected* werden

- Bei Blazor Server können Javascript Methoden erst aufgerufen werden, sobald eine der Signal R Channel aufgebaut ist.

Um nun diesen Mechanismus zu demonstrieren, soll im Folgenden ein Beispiel dargestellt werden, in welchem ein String von einer Javascript Methode erzeugt wird, und beim Betätigen eines Buttons, dieser String auf der Benutzeroberfläche angezeigt wird:

```
1 function generateString(name) {  
2     return "Hello " + name + ", u have clicked the button!";  
3 }
```

Listing 3.4: Javascript Methode Generate String

```
1 @page "/jsExample"  
2 @inject IJSRuntime _js  
3  
4 <h1>Javascript Runtime</h1>  
5  
6 <p>@output</p>  
7  
8 <button @onclick="CallJS">Click Me</button>  
9  
10 @code {  
11     private string output = string.Empty;  
12  
13     private async Task CallJS()  
14     {  
15         output = await _js.InvokeAsync<string>("generateString", "Mustermann");  
16     }  
17 }
```

Listing 3.5: Javascript Beispiel Komponente

Im obigen Beispiel, wurde die Javascript datei global eingebunden, so dass diese von überall erreichbar ist. Seit .Net 5 existiert die Möglichkeit, eine Javascript datei direkt in eine Komponente einzubinden [12][vgl.]. Dies nennt sich dann *Javascript Isolation* und würde wie folgt aussehen:

```
1 export function generateString(name) {  
2     return "Hello " + name + ", u have clicked the button!";  
3 }
```

Listing 3.6: Javascript Isolation Generate String

Damit die Javascript nun gefunden werden kann, muss diese mit dem Keyword *export* versehen werden.

```
1 @page "/jsIsolation"  
2 @inject IJSRuntime _js  
3  
4 <h1>Javascript Isolation</h1>
```

```
5
6 <p>@output</p>
7
8 <button @onclick="CallJS">Click Me</button>
9
10 @code {
11     private string output = string.Empty;
12
13     private async Task CallJS()
14     {
15         var jsModule = await _js.InvokeAsync<IJSObjectReference>("import", "./js/
16             testIsolation.js");
17         output = await jsModule.InvokeAsync<string>("generateString", "Mustermann");
18     }
19 }
```

Listing 3.7: Javascript Isolation Beispiel

Wie zu sehen ist, kann mit der *InvokeAsync* Methode auch die komplette Javascript datei geladen werden. Sobald die Javascript Datei geladen wurde, ist der Aufruf identisch zu Listig 3.4.

3.4.2. Javascript Invokable

Nun kann es auch den Fall geben, dass C# code von Javascript aufgerufen werden muss. Dies ist in Blazor ebenso möglich und kann vollbracht werden, mit dem *JSInvokable* Attribut, die über einer Methode angebracht werden muss. Zudem muss die Methode als *static* implementiert werden.

Auf Javascript seite, wird von Blazor die Methoden

- `DotNet.invokeMethod (string, string, object[])`
- `DotNet.invokeMethodAsync (string, string, object[])`

bereitgestellt, mit denen dann eine statische C# Methode aufgerufen werden kann. Dabei gibt der Entwickler mithilfe des ersten Parameters, den Project Namen an, in der sich die statische Methode befindet, der zweite Parameter, ist für den Namen der aufzurufenden Methode gedacht und zu guter letzt können mit dem dritten Parameter noch weitere optionale Argumente übergeben werden.

Im folgenden, wird eine Komponente gezeigt welche beim Betätigen eines But-

tons, eine Javascript Methode aufruft, welche wiederum einen generierten string von einer C# Methode erhält und diese in einem *alert* ausgibt:

```
1 @page "/jsInvokable"  
2 @inject IJSRuntime _js  
3  
4 <h1>Javascript Invokable</h1>  
5  
6 <button onclick="showGeneratedMessage()">Click Me</button>  
7  
8 @code {  
9  
10     [JSInvokable]  
11     public static Task<string> GenerateString()  
12     {  
13         return Task.FromResult("This is called from Javascript");  
14     }  
15  
16 }
```

Listing 3.8: Javascript Invokable Beispiel

```
1 function showGeneratedMessage() {  
2     DotNet.invokeMethodAsync('Test', 'GenerateString')  
3         .then(data => {  
4             alert(data)  
5         });  
6 }
```

Listing 3.9: Javascript Invokable Show Generated Message

3.5. Blazor Maui

Mit dem Release von .Net 6 kam auch der Zusammenschluss von Blazor und .Net Maui erstmalig zum Vorschein. Die Abkürzung *Maui* steht dabei für *Multi-platform Application UI*. Mit Blazor Maui sollen zukünftig Native Desktop oder Mobile Applikationen mit Blazor erstellt werden können. Somit können nicht nur Native Applikationen mit Hilfe von alt bekannten Html und Css Komponenten erstellt werden, sondern auch schon erstellte Blazor Komponenten, die für eine Webseite erstellt wurden, können wiederverwendet werden.

Die Architektur von Blazor Maui sieht vor, dass der Blazor code durch Maui in Nativen code umgewandelt wird, um dann auf der jeweiligen Plattform ausgeführt zu werden, wie im folgenden Bild zu erkennen ist:

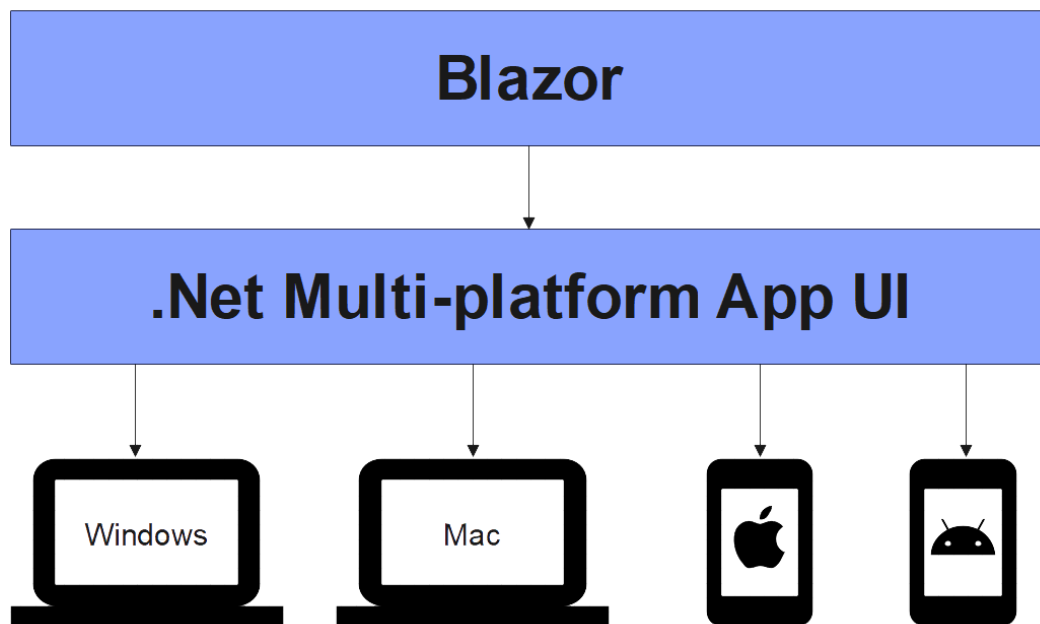


Abbildung 3.4: Blazor Maui Architektur

Wie zu sehen ist, werden Windows, Mac, Ios und Android mithilfe von Maui unterstützt, Linux hingegen bleibt zu dem Zeitpunkt dieser Thesis noch außen vor.

Aufgrund dessen, dass Linux zum derzeitigen Zeitpunkt noch nicht unterstützt wird, und Open Embedded System meist auf Linux basieren, wird Blazor Maui nicht weitergehend in dieser Thesis behandelt.

4. Raspberry Pi mit .Net Core und Blazor

In diesem Kapitel soll nun ein einfaches Open Embedded System mithilfe von Blazor auf einem Raspberry Pi 4B erstellt werden. Dabei soll gezeigt werden, welche Entwicklungsumgebung, unter anderem, genutzt werden kann und was auf dem Target installiert werden muss.

4.1. Entwicklungsumgebung

Die Entwicklungsumgebung, die in dieser Thesis eingesetzt wird, ist Visual Studio Code. Dabei wurde Visual Studio Code, deswegen ausgesucht, da viele Community Plugins zur Verfügung stehen, die das Entwickeln auf dem Raspberry Pi unterstützen. Zum einen ist die Möglichkeit gegeben, mithilfe von Visual Studio Code, sich Remote mit dem Raspberry Pi zu verbinden und zum anderen existiert noch die Möglichkeit des Remote Debuggings. Beim Remote Debugging wird der Code auf dem Host Rechner entwickelt und auf dem Target ausgeführt.

Im Zuge dieser Thesis wird sich Remote mit dem Raspberry Pi verbunden, um dann auf dem Target zu programmieren. Dafür muss die Extension *Remote - SSH* von Microsoft in Visual Studio Code installiert werden. Nun kann mit der Tastenkombination *Strg - shift - p* ein neuer Dialog geöffnet werden, in welchem dann die Option *Remote-SSH Connect to host* ausgewählt werden kann. Nachdem *Remote-SSH Connect to host* ausgewählt wurde, muss der Befehl `<benutzernamen>@<ip adresse>` eingegeben werden. Dabei steht der Benutzername für den Benutzernamen des Raspberry Pi's, der standardmäßig *pi* lautet, und die IP Adresse für die IP Adresse, von dem Raspberry Pi. Zuletzt muss lediglich noch das Passwort eingegeben werden und wenn alles geklappt hat, kann nun mithilfe von Visual Studio Code auf den kompletten Raspberry Pi zugegriffen werden.

4.2. Installation

Da in der vorherigen Sektion *Entwicklungsumgebung*, Visual Studio Code eingerichtet wurde und mit dem Raspberry Pi Remote verbunden wurde, kann nun auch das integrierte Terminal von Visual Studio Code für die Installation von .Net Core beziehungsweise .Net verwendet werden.

Zuallererst muss überprüft werden, ob es sich bei dem Raspberry Pi um die 32-bit oder die 64-bit version handelt. Dies kann mit dem befehl `uname -a` überprüft werden. Dabei können folgende zwei ergebnisse erfolgen:

```
Linux raspberrypi 4.19.97-v7l+ 1294 SMP Thu Jan 30 13:23:13 GMT 2020
armv7l GNU/Linux
Linux raspberrypi 5.10.60-v8+ 1291 SMP Thu Jan 30 13:21:14 GMT 2020
aarch64 GNU/Linux
```

Sollte das Resultat nun *armv7l GNU/Linux* anzeigen, dann handelt es sich bei dem Raspberry Pi um die 32-bit version, sollte aber *aarch64 GNU/Linux* angezeigt werden, dann ist es die 64-bit version.

Nun kann entweder auf der offiziellen Microsoft seite, die richtige SDK heruntergeladen werden, oder mithilfe von `wget` die SDK vom Terminal heruntergeladen werden. Nachdem der Download beendet ist, können mit den folgenden befehlen, die Installation beendet werden:

```
mkdir -p $HOME/dotnet
tar xzf dotnet-sdk-6.0.100-rc.1.21458.32-linux-arm.tar.gz -C $HOME/dotnet
export DOTNET_ROOT=$HOME/dotnet
export PATH=$PATH: $HOME/dotnet
```

Diese Befehle erstellen einen neuen Ordner namens *dotnet*, entpacken die SDK und packen den Inhalt in den neu erstellten Ordner und setzen zu guter letzt noch die *Path-Variable* zu dem dotnet Ordner.

Mit dem Befehl `dotnet --info` kann anschließend überprüft werden, ob die Installation erfolgreich war.

4.3. Blazor Demo Anwendung

Nachdem in der letzten Sektion *Installation* .Net installiert wurde, soll in dieser Sektion eine beispielhafte Blazor Anwendung auf dem Raspberry Pi erstellt werden. Die Anwendung wird auf der *Blazor Server* architektur basieren. Dabei sollen kontinuierlich Daten von dem Raspberry Pi abgefragt und auf der Benutzeroberfläche repräsentiert werden.

4.3.1. Erstellen des Projektes

Um eine Blazor Anwendung zu erstellen, gibt es zwei Möglichkeiten. Zum einem vom *Scratch* aus, aus einer einfachen Konsolen Anwendung, um den notwendigen Code zu implementiert, oder zum anderen mit hilfe eines Templates. Microsoft bietet einige Templates für verschiedenste Anwendungen an, die alle samt mit der Installation von .Net kommen.

Um nun die Anwendung mithilfe des Templates zu erstellen, muss der folgende Befehl in dem Terminal eingegeben werden:

```
dotnet new blazorserver -o MyApp --no-https
```

Dieser Befehl erstellt eine neue Blazor Server Anwendung, mit dem Namen *MyApp* und konfiguriert die Anwendung ohne das Https Protokol. Der name sowie, dass kein Https konfiguriert wird, sind optionale Parameter, die nicht mit angegeben werden müssen. Nachdem die Anwendung erfolgreich Installiert wurde, muss noch die Codezeile `webBuilder.UseUrls("Http://*:5000");` in der *Program.cs* angegeben werden, um die Anwendung für alle Geräte im LAN verfügbar ist. Der Code in der *Program.cs* sieht dann folgendermaßen aus:

```
1  public class Program
2  {
3      // Main
4
5      public static IHostBuilder CreateHostBuilder(string[] args) =>
6          Host.CreateDefaultBuilder(args)
7              .ConfigureWebHostDefaults(webBuilder =>
8              {
9                  webBuilder.UseStartup<Startup>();
10                 webBuilder.UseUrls("Http://*:5000"); // <----
11             });
```

12 }

Listing 4.1: Program.cs Code

Mit dem Befehl *dotnet run* kann die Anwendung dann gestartet werden. Sobald das Programm gestartet ist, kann mit dem Link *http://<ip>:5000* die Seite erreicht werden. Sollte alles geklappt haben sollte nun die Seite, mit den schon vom Template gegebenen Komponenten, wie folgt angezeigt werden:

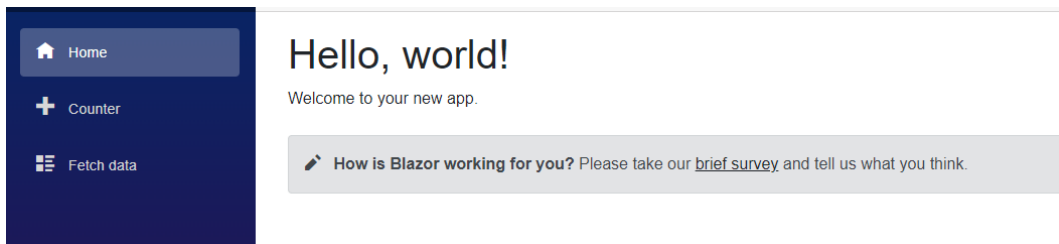


Abbildung 4.1: Blazor Server Template

4.3.2. Microsoft Iot

Damit mit dem Raspberry Pi kommuniziert werden kann, können Bibliotheken verwendet werden. Microsoft bietet eine Iot Bibliothek an, mit der dann die Sensoren auf dem Gerät oder die LED's auf dem Gerät angesteuert werden können. Die Bibliothek kann in einem Projekt entweder durch den *Nuget Packager* eingebunden werden, oder indem die *PackageReference* wie folgt, in der Solution Datei konfiguriert wird:

```
1 <ItemGroup>
2 <PackageReference Include="Iot.Device.Bindings" Version="1.5.0-*" />
3 </ItemGroup>
```

Listing 4.2: Iot Nuget Package

Nun kann der Programmier zum Beispiel *using Iot.Device.SenseHat* in den Code einbinden um auf die Sensoren des Sensehat's zuzugreifen. Damit veranschaulicht wird, wie auf die Daten zugegriffen werden kann, wird ein Beispielhaftes Konsolen Programm demonstriert:

```
1 public class Program
2 {
3     static void Main(string[] args)
4     {
5         using SenseHat _senseHat = new();
6     }
```

```
7         Console.WriteLine($"Temperatur: {_senseHat.Temperature.DegreesCelsius  
            :0.##}\u00B0C");  
8         Console.WriteLine($"Temperatur 2: {_senseHat.Temperature2.DegreesCelsius  
            :0.##}\u00B0C");  
9         Console.WriteLine($"Luftdruck: {_senseHat.Pressure.Hectopascals:0.##}  
            hPa");  
10        Console.WriteLine($"Luftfeuchtigkeit: {_senseHat.Humidity.Percent:0.##}"  
            );  
11    }  
12 }
```

Listing 4.3: SenseHat Beispiel Programm

Der Output der bei dem Beispiel erzeugt wird, könnte so aussehen:

```
Temperatur: 38,4°C  
Temperatur 2: 38,5°C  
Luftdruck: 984,03 hPa  
Luftfeuchtigkeit: 31%
```

4.3.3. Raspberry Pi Daten Anzeigen

In dieser Sektion soll nun die Logik implementiert werden, um die Daten, die vom Raspberry Pi kommen, auf der Benutzeroberfläche anzuzeigen. Dafür wird ein *Timer* implementiert werden, der jede Sekunde auslöst, um dann die Daten zu lesen.

Damit etwas auf der Benutzeroberfläche angezeigt werden kann, muss das *Html-Markup* implementiert werden:

```
1 <div class="divHeader">  
2     <div>  
3         <h1>Raspberry Pi</h1>  
4  
5         <h2>Uhrzeit: @_uhrzeit</h2>  
6     </div>  
7  
8     <div>  
9         <h3>Temperature Sensor 1: @_temperatur</h3>  
10        <h3>Temperature Sensor 2: @_temperatur2</h3>  
11        <h3>Luftdruck: @_pressure</h3>  
12        <h3>Luftfeuchtigkeit: @_humidity</h3>  
13    </div>  
14 </div>
```

Listing 4.4: Html-Markup

Dabei signalisiert das `@<name>`, dass es sich dabei um eine Variable handelt, die im Code deklariert wurde.

Weitergehend bietet Blazor verschiedene *Render-Funktionen* die nach bestimmten ereignissen aufgerufen werden. Wie zum Beispiel die *OnInitializedAsync*, die aufgerufen wird, wenn die Komponente geladen wird. Diese *OnInitializedAsync* kann nun dazu gebraucht werden, um die Variablen zu initialisieren.

```
1  protected override Task OnInitializedAsync()
2  {
3      _senseHat = new();
4      _cultureInfo = new("de-DE");
5      _uhrzeit = DateTime.Now.ToString("HH:mm:ss", _cultureInfo);
6
7      _temperatur = string.Empty;
8      _temperatur2 = string.Empty;
9      _pressure = string.Empty;
10     _humidity = string.Empty;
11
12     return base.OnInitializedAsync();
13 }
```

Listing 4.5: Render-Funktion: OnInitializedAsync

Nun soll zudem noch eine Hilfsfunktion *SetRaspValues* geschaffen werden, die die Daten ausliest. Diese Funktion wird dann auch in der *OnInitializedAsync* Funktion aufgerufen.

```
1  private void SetRaspValues()
2  {
3      _temperatur = $"{_senseHat.Temperature.DegreesCelsius:0.##}\u00B0C";
4      _temperatur2 = $"{_senseHat.Temperature2.DegreesCelsius:0.##}\u00B0C";
5      _pressure = $"{_senseHat.Pressure.Hectopascals:0.##} hPa";
6      _humidity = $"{_senseHat.Humidity.Percent:0.##}%";
7  }
```

Listing 4.6: Funktion: SetRaspValues

Damit die Daten jedoch nicht nur einmal am Anfang gelesen werden, sondern sich kontinuierlich aktualisieren, soll ein *Timer* jede Sekunde getriggert werden, um die Daten neu zu lesen und dem *DOM* mitzuteilen, dass sich der *State* der Seite geändert hat.

```
1  private void StartTimer()
2  {
3      _readTimer = new(1000);
4      _readTimer.Elapsed += GetData;
5      _readTimer.Enabled = true;
6  }
7  }
```

```
8 private void GetData(Object source, System.Timers.ElapsedEventArgs e)
9 {
10     _uhrzeit = DateTime.Now.ToString("HH:mm:ss", cultureInfo);
11     SetRaspValues();
12     InvokeAsync(StateHasChanged);
13 }
```

Listing 4.7: Timer: ReadTimer

Wichtig hierbei ist jedoch, dass die Funktion *StateHasChanged* manuell aufgerufen wurde. Dies übernimmt Blazor normalerweise schon automatisch, wenn sich Variablen der Komponente verändern, die angezeigt werden aber da hier die Variablen nicht auf dem *Ui-Thread* geändert wurden, muss manuell angegeben werden, dass sich der *State* geändert hat.

Der momentane Stand der Seite zieht so aus:

Raspberry Pi	Temperature Sensor 1: 38,3°C
Uhrzeit: 15:36:12	Temperature Sensor 2: 38,5°C
	Luftdruck: 1005,67 hPa
	Luftfeuchtigkeit: 25,2%

Abbildung 4.2: Zwischenstand der Blazor Demo

4.3.4. LED-Matrix Ansteuern

In dieser Sektion soll nun eine *8x8 Button-Matrix* erstellt werden, die die *8x8 LED-Matrix* auf dem SenseHat des Raspberry Pi's repräsentieren soll. Dabei soll der vorhandene Joystick auf dem SenseHat, dazu genutzt werden um über die Button-Matrix zu navigieren und die LED's zu platzieren.

Dadurch, dass Blazor die *Razor-Syntax* verwendet, kann jegliche C# kontrollstruktur im *Html-Markup* verwendet werden. So kann dies auch wie folgt genutzt werden, um die *8x8 Button-Matrix* zu erzeugen:

```
1 <div class="divGrid">
2     @for (var y = 0; y < LengthY; y++)
3     {
4         @for (var x = 0; x < LengthX; x++)
5         {
6             int copyY = y;
7             int copyX = x;
8             <Button class="buttonBox @classes[copyY,copyX]" />
9         }
10    }
```


11 </div>

Listing 4.8: Button-Matrix

Das *@classes* element, ist ein pseudo Zwei-Dimensional Array aus strings, mit der dann Css-Klassen hinzugefügt und wieder gelöscht werden sollen. Pseudo Zwei-Dimensionales Array deswegen, weil diese in C# nur statisch existieren, aufgrund von inperformance.

Um zu ermitteln, ob und wenn ja welcher Joystick button betätigt wurde, soll auch hier wieder ein *Timer* zum einsatz kommen, der jedoch diesmal alle 15 Millisekunden getriggert wird.

```
1 private void StartTimer()
2 {
3     // More Code
4
5     _setButtonTimer = new(15);
6     _setButtonTimer.Elapsed += WriteStateToChannel;
7     _setButtonTimer.Enabled = true;
8 }
9
10
11 private async void WriteStateToChannel(Object source, System.Timers.
    ElapsedEventArgs e)
12 {
13     if((ticks - lastTicks) > 9){
14         _senseHat.ReadJoystickState();
15         if(_senseHat.HoldingButton){
16             await _stateChannel.Writer.WriteAsync(JoystickState.Holding);
17         }
18         else if(_senseHat.HoldingUp){
19             await _stateChannel.Writer.WriteAsync(JoystickState.Up);
20         }
21         else if(_senseHat.HoldingDown){
22             await _stateChannel.Writer.WriteAsync(JoystickState.Down);
23         }
24         else if(_senseHat.HoldingLeft){
25             await _stateChannel.Writer.WriteAsync(JoystickState.Left);
26         }
27         else if(_senseHat.HoldingRight){
28             await _stateChannel.Writer.WriteAsync(JoystickState.Right);
29         }
30         lastTicks = ticks;
31     }
32     ticks++;
33 }
```

Listing 4.9: Timer: ButtonTimer

Bei *JoystickState* handelt es sich um ein Enum, welches den *State* des Joysticks repräsentieren soll. Wie zu sehen ist, wird der aktuelle *State* des Joysticks gelesen um dann das Ergebnis in einen *Channel* zu schreiben. Für diesen Timer, wurde bewusst das Konzept von *Channels* genutzt, anstatt die Berechnungen und die State-Änderung direkt im Timer zu Implementieren, da der *Timer* alle 15 Millisekunden ausgelöst wird und dadurch vermieden werden sollte, dass das Programm im Timer hängen bleibt.

Für die Verarbeitung des *States*, wird in *OnInitializedAsync* ein Task gestartet, der in einer *endless-loop* läuft und darauf wartet, bis etwas in den *Channel* hinzugefügt wurde. Sobald sich der *State* geändert hat, passiert abhängig von dem Event, wie folgend zu sehen sein wird, das entweder die Position berechnet wird, oder die LED und der Button an dieser Position, die Farbe wechselt.

```
1      _setButtonTask = Task.Run(async () =>
2      {
3          while (true)
4          {
5              if(cancellationToken.IsCancellationRequested)
6                  cancellationToken.ThrowIfCancellationRequested();
7              var state = await _stateChannel.Reader.ReadAsync();
8              int x = 0;
9              int y = 0;
10             if(state == JoystickState.Holding){
11                 SetButtonBackground(_currentX, _currentY);
12             }
13             else if(state == JoystickState.Up){
14                 y--;
15             }
16             else if(state == JoystickState.Down){
17                 y++;
18             }
19             else if(state == JoystickState.Left){
20                 x--;
21             }else if(state == JoystickState.Right){
22                 x++;
23             }
24             SetPositions(x, y);
25             RemoveButtonBorder(_previousX, _previousY);
26             SetButtonBorder(_currentX, _currentY);
27             await InvokeAsync(StateHasChanged);
28         }
29     });
30
31     private void SetButtonBackground(int x, int y)
32     {
33         if (classes[y, x].Contains(" setBackground"))
34         {
35             _senseHat.SetPixel(x, y, Color.Blue);
```

```
36         classes[y, x] = classes[y, x].Replace(" setBackground", string.Empty);  
37     }  
38     else  
39     {  
40         _senseHat.SetPixel(x, y, Color.Red);  
41         classes[y, x] += " setBackground";  
42     }  
43 }
```

Listing 4.10: Task zum Verarbeiten des States

Wie zu sehen ist, wird, sollte der Joystick gedrückt werden, die LED und der Button je nach vorherigem Zustand entweder Blau oder Rot. In den anderen Fällen, werden lediglich die neuen Positionen berechnet und ein Border wird gesetzt, um zu signalisieren an welcher Positon auf der Matrix, der momentane Fokus ist.

Die komplette Anwendung sieht dann wie folgt aus:

Raspberry Pi
Uhrzeit: 15:36:12

Temperature Sensor 1: 38,3°C
Temperature Sensor 2: 38,5°C
Luftdruck: 1005,67 hPa
Luftfeuchtigkeit: 25,2%

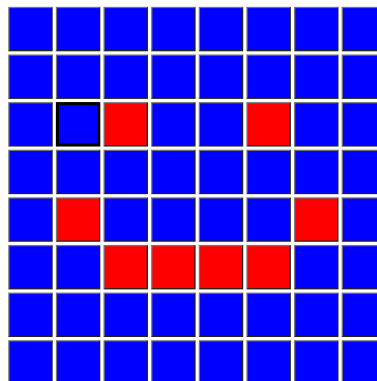


Abbildung 4.3: Blazor Demo

Da nur Technisch relevante Code abschnitte in diesem Kapitel präsentiert wurden, ist der komplette Code im Anhang A.1 zu finden.

5. Analyse

Test

6. Fazit

Test

7. Ausblick

Test

Abkürzungsverzeichnis

GUI	Graphical User Interface	1
moc	meta object compiler	8
SPA 's	Single Page Applications	15

Tabellenverzeichnis

2.1. Anforderungen an Embedded Systems	6
--	---

Abbildungsverzeichnis

1.1. Blazor mit Raspberry Pi	2
1.2. Raspberry Pi 4 B	3
2.1. Open vs Deeply Embedded Systeme	5
2.2. Qt Logo	8
2.3. Qt Hello World App	9
2.4. Qt Widgets Klassenhierarchie	10
2.5. Open Embedded System GUI	14
3.1. Client-Server Architektur mit Javascript	16
3.2. Blazor WebAssembly Architektur	17
3.3. Blazor Server Architektur	18
3.4. Blazor Maui Architektur	25
4.1. Blazor Server Template	29
4.2. Zwischenstand der Blazor Demo	32
4.3. Blazor Demo	35

Listings

2.1.	Qt Hello World Sourcecode	9
2.2.	Signal und Slot beispiel	11
2.3.	RTIMU-Hilfsklasse	12
2.4.	RTIMU-Hilfsklasse-Konstruktor	13
2.5.	RTIMU-Hilfsklasse-vRead	13
2.6.	Update Button Slot	14
3.1.	Page-Komponente Beispiel	19
3.2.	Kind Komponente	20
3.3.	Eltern Komponente	20
3.4.	Javascript Methode Generate String	22
3.5.	Javascript Beispiel Komponente	22
3.6.	Javascript Isolation Generate String	22
3.7.	Javascript Isolation Beispiel	22
3.8.	Javascript Invokable Beispiel	24
3.9.	Javascript Invokable Show Generated Message	24
4.1.	Program.cs Code	28
4.2.	Iot Nuget Package	29
4.3.	SenseHat Beispiel Programm	29
4.4.	Html-Markup	30
4.5.	Render-Funktion: OnInitializedAsync	31
4.6.	Funktion: SetRaspValues	31
4.7.	Timer: ReadTimer	31
4.8.	Button-Matrix	32
4.9.	Timer: ButtonTimer	33
4.10.	Task zum Verarbeiten des States	34
A.1.	Kompletter Demo Code	vi

Literatur

- [1] Hochschule niederrhein, *CAS Embedded Systems Professional - Hochschule Niederrhein*, 29.10.2021. Adresse: <https://www.hs-niederrhein.de/weiterbildung/sichere-software/cas-embedded-systems-professional/>.
- [2] *Raspberry Pi 4 Model B specifications – Raspberry Pi*, 5.11.2021. Adresse: <https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/>.
- [3] J. Quade, *Embedded Linux lernen mit dem Raspberry Pi: Linux-Systeme selber bauen und programmieren*, 1. Auflage. Heidelberg: dpunkt.verlag, 2014, ISBN: 9783864915093. Adresse: http://ebooks.ciando.com/book/index.cfm/bok_id/1423957.
- [4] Q. Jinhui, L. D. Hui und Y. Junchao, „The Application of Qt/Embedded on Embedded Linux“, in *2012 International Conference on Industrial Control and Electronics Engineering*, 2012, S. 1304–1307. DOI: 10.1109/ICICEE.2012.346.
- [5] *the-qt-story*, 22.03.2016. Adresse: <https://rtime.felk.cvut.cz/osp/prednasky/gui/the-qt-story/>.
- [6] B. Baka, *Getting Started with Qt 5: Introduction to programming Qt 5 for cross-platform application development*, 1. Aufl. Birmingham: Packt Publishing Limited, 2019, ISBN: 9781789955125. Adresse: https://www.wiso-net.de/document/PKEB__9781789955125136.
- [7] M. Lobur, I. Dykhta, R. Golovatsky und J. Wrobel, „The usage of signals and slots mechanism for custom software development in case of incomplete information“, in *2011 11th International Conference The Experience of Designing and Application of CAD Systems in Microelectronics (CADSM)*, 2011, S. 226–227.
- [8] bbv, *Hier kommt Blazor*, 4.12.2021. Adresse: <https://www.bbv.ch/blazor/>.

- [9] Kexugit, *Web Development - C# in the Browser with Blazor*, 3.12.2021.
Adresse: <https://docs.microsoft.com/en-us/archive/msdn-magazine/2018/september/web-development-csharp-in-the-browser-with-blazor>.
- [10] Guardrex, *ASP.NET Core Razor components*, 10.12.2021. Adresse:
<https://docs.microsoft.com/en-us/aspnet/core/blazor/components/?view=aspnetcore-6.0>.
- [11] ———, *Aufrufen von JavaScript-Funktionen über .NET-Methoden in ASP.NET Core Blazor*, 10.12.2021. Adresse:
<https://docs.microsoft.com/de-de/aspnet/core/blazor/javascript-interoperability/call-javascript-from-dotnet?view=aspnetcore-6.0>.
- [12] G. Barré, „JavaScript Isolation in Blazor Components“, 21.09.2020.
Adresse:
<https://www.meziantou.net/javascript-isolation-in-blazor-components.htm>.

A. Ein Anhang

```
1 @page "/"
2 @implements IAsyncDisposable
3 @using System.Drawing;
4 @using System.Threading
5 @using System.Threading.Channels
6 @using System.Globalization
7 @using Iot.Device.Common;
8 @using Iot.Device.SenseHat;
9
10 <div class="divHeader">
11     <div>
12         <h1>Raspberry Pi</h1>
13
14         <h2>Uhrzeit: @_uhrzeit</h2>
15     </div>
16
17     <div>
18         <h3>Temperature Sensor 1: @_temperatur</h3>
19         <h3>Temperature Sensor 2: @_temperatur2</h3>
20         <h3>Luftdruck: @_pressure</h3>
21         <h3>Luftfeuchtigkeit: @_humidity</h3>
22     </div>
23 </div>
24
25 <div class="divGrid">
26     @for (var y = 0; y < LengthY; y++)
27     {
28         @for (var x = 0; x < LengthX; x++)
29         {
30             int copyY = y;
31             int copyX = x;
32             <Button class="buttonBox @classes[copyY,copyX]" @onclick="() =>
33                 SetButtonBackground(copyX, copyY)" />
34         }
35     }
36 </div>
37 @code {
38
39     private enum JoystickState {
40         EMPTY = 0,
```

```
41     Holding,
42     Up,
43     Down,
44     Right,
45     Left,
46 }
47
48 private const int LengthY = 8;
49 private const int LengthX = 8;
50
51 private string _uhrzeit;
52
53 private System.Timers.Timer _timeTimer;
54 private System.Timers.Timer _setButtonTimer;
55 private Random _randomNumberGenerator;
56
57 private CultureInfo cultureInfo;
58
59 private Channel<JoystickState> _stateChannel;
60 private Task _setButtonTask;
61 private SenseHat _senseHat;
62
63 private CancellationTokenSource _tokenSource;
64
65 string _temperatur = string.Empty;
66 string _temperatur2 = string.Empty;
67 string _pressure = string.Empty;
68 string _humidity = string.Empty;
69
70 int _previousX;
71 int _previousY;
72
73 int _currentX;
74 int _currentY;
75
76 string[,] classes = new string[LengthY, LengthX];
77
78 uint ticks = 0;
79 uint lastTicks = 0;
80
81 protected override Task OnInitializedAsync()
82 {
83     _stateChannel = Channel.CreateUnbounded<JoystickState>();
84     _randomNumberGenerator = new();
85     cultureInfo = new("de-DE");
86     _uhrzeit = DateTime.Now.ToString("HH:mm:ss", cultureInfo);
87     _senseHat = new();
88     _tokenSource = new();
89     var cancellationToken = _tokenSource.Token;
90
91     for (var y = 0; y < LengthY; y++)
92     {
93         for (var x = 0; x < LengthX; x++)
```

```

94         {
95             classes[y, x] = string.Empty;
96         }
97     }
98
99     _senseHat.Fill(Color.Blue);
100
101     StartTimer();
102     InitPositions();
103     SetRaspValues();
104
105     SetButtonBorder(_currentX, _currentY);
106
107     _setButtonTask = Task.Run(async () =>
108     {
109         while (true)
110         {
111             if(cancellationToken.IsCancellationRequested)
112                 cancellationToken.ThrowIfCancellationRequested();
113
114             var state = await _stateChannel.Reader.ReadAsync();
115
116             int x = 0;
117             int y = 0;
118
119             if(state == JoystickState.Holding){
120                 SetButtonBackground(_currentX, _currentY);
121             }
122             else if(state == JoystickState.Up){
123                 y--;
124             }
125             else if(state == JoystickState.Down){
126                 y++;
127             }
128             else if(state == JoystickState.Left){
129                 x--;
130             }else if(state == JoystickState.Right){
131                 x++;
132             }
133
134             SetPositions(x, y);
135
136             RemoveButtonBorder(_previousX, _previousY);
137             SetButtonBorder(_currentX, _currentY);
138
139             await InvokeAsync(StateHasChanged);
140         }
141     });
142
143     return base.OnInitializedAsync();
144 }
145
146 private void StartTimer()

```

```
147 {
148     // Every Second
149     _timeTimer = new(1000);
150     _timeTimer.Elapsed += GetData;
151     _timeTimer.Enabled = true;
152
153     // Every 15 Milliseconds
154     _setButtonTimer = new(15);
155     _setButtonTimer.Elapsed += WriteStateToChannel;
156     _setButtonTimer.Enabled = true;
157 }
158
159 private void InitPositions(){
160     _previousX = 0;
161     _previousY = 0;
162     _currentX = 0;
163     _currentY = 0;
164 }
165
166 private void SetPositions(int newX, int newY){
167     _previousX = _currentX;
168     _previousY = _currentY;
169
170     _currentX = (_currentX + LengthX + newX) % 8;
171     _currentY = (_currentY + LengthY + newY) % 8;
172 }
173 private void GetData(Object source, System.Timers.ElapsedEventArgs e)
174 {
175     _uhrzeit = DateTime.Now.ToString("HH:mm:ss", cultureInfo);
176
177     SetRaspValues();
178
179     InvokeAsync(StateHasChanged);
180 }
181
182 private async void WriteStateToChannel(Object source, System.Timers.
    ElapsedEventArgs e)
183 {
184     if((ticks - lastTicks) > 9){
185
186         _senseHat.ReadJoystickState();
187
188         if(_senseHat.HoldingButton){
189             await _stateChannel.Writer.WriteAsync(JoystickState.Holding);
190         }
191         else if(_senseHat.HoldingUp){
192             await _stateChannel.Writer.WriteAsync(JoystickState.Up);
193         }
194         else if(_senseHat.HoldingDown){
195             await _stateChannel.Writer.WriteAsync(JoystickState.Down);
196         }
197         else if(_senseHat.HoldingLeft){
198             await _stateChannel.Writer.WriteAsync(JoystickState.Left);
```



```
199     }
200     else if(_senseHat.HoldingRight){
201         await _stateChannel.Writer.WriteAsync(JoystickState.Right);
202     }
203
204     lastTicks = ticks;
205 }
206 ticks++;
207 }
208
209 private void SetRaspValues()
210 {
211     _temperatur = $"{_senseHat.Temperature.DegreesCelsius:0.##}\u00B0C";
212     _temperatur2 = $"{_senseHat.Temperature2.DegreesCelsius:0.##}\u00B0C";
213     _pressure = $"{_senseHat.Pressure.Hectopascals:0.##} hPa";
214     _humidity = $"{_senseHat.Humidity.Percent:0.##}%";
215 }
216
217 public async ValueTask DisposeAsync()
218 {
219     _timeTimer.Dispose();
220     _setButtonTimer.Dispose();
221     _tokenSource.Cancel();
222     _tokenSource.Dispose();
223     _senseHat.Dispose();
224 }
225
226 private void SetButtonBackground(int x, int y)
227 {
228     if (classes[y, x].Contains(" setBackground"))
229     {
230         _senseHat.SetPixel(x, y, Color.Blue);
231         classes[y, x] = classes[y, x].Replace(" setBackground", string.Empty);
232     }
233     else
234     {
235         _senseHat.SetPixel(x, y, Color.Red);
236         classes[y, x] += " setBackground";
237     }
238 }
239
240 private void RemoveButtonBorder(int x, int y){
241     if (classes[y, x].Contains(" setBorder"))
242     {
243         classes[y, x] = classes[y, x].Replace(" setBorder", string.Empty);
244     }
245 }
246
247 private void SetButtonBorder(int x, int y){
248     if (!classes[y, x].Contains(" setBorder"))
249     {
250         classes[y, x] += " setBorder";
251     }
252 }
```

```
252     }
253
254 }
255
256 Css:
257 .setBorder {
258     border: solid 5px black;
259 }
260
261 .setBackground {
262     background-color: red !important;
263 }
264
265 .buttonBox {
266     width: 100%;
267     height: 100%;
268     background: blue;
269 }
270
271 .divGrid {
272     display: grid;
273     grid-template-columns: repeat(8, 70px);
274     grid-template-rows: repeat(8, 70px);
275     grid-gap: 5px;
276     justify-content: center
277 }
278
279 .divHeader {
280     display: flex;
281     flex-direction: row;
282     justify-content: space-between;
283 }
284
285 .divHeader div {
286     display: flex;
287     flex-direction: column;
288 }
289
290 .divHeader div:last-child {
291     align-items: flex-end;
292     margin-right: 100px;
293 }
294
295 .divHeader div:last-child h3 {
296     text-align: right;
297 }
```

Listing A.1: Kompletter Demo Code