

Frankfurt University of Applied Sciences

– Faculty of Computer Science and Engineering –

Analyse und Erhebung von Sensordaten

”schwellenwertbasierte Bewegungserkennung mit Bluetooth”

vorgelegt am 26.03.2025 von

Sebastian Aybar

Matrikelnummer: 1273441

Okan Süner

Matrikelnummer: 1295704

Muhammed Önal

Matrikelnummer: 1272691

Ferid Gökkaya

Matrikelnummer: 1250370

Referent : Daniel Helmer

Inhaltsverzeichnis

Abbildungsverzeichnis

Einleitung	1
Hardware Setup	2
Sensormodul	2
Mikrocontroller	2
Programmierung	3
Auslesen der Sensordaten	3
Bewegungserkennung	6
Erstellen und Aktualisieren einer Characteristic	15
Flash Firmware	22
Für Sensordata.py	23
Importierte Bibliotheken	23
Klasse Sensordata	24
Klasse SensordataList	25
Für Graphics.py	26
Fazit	30
Literatur	31

Abbildungsverzeichnis

1	Configuration_Accelerometer	4
2	Read_Sensor_Data	5
3	XY-Achsenausrichtung	6
4	Determine_Activities	7
5	Determine_Activities	8
6	Get_Y_Axis_Orientation	8
7	Get_XY_Axis_Orientation	10
8	Get_Posture	11
9	Determine_Activities	12
10	cusotm_stm.c	15
11	Custom_STM_Event_Handler	16
12	Custom_STM_App_Notification	17
13	Process_Read_Request_For_Data	18
14	Process_Read_Request_For_Data eigen	18
15	Determine_RawData	19
16	Ble_Update_Characteristic	19
17	Custom_STM_App_Update_Char	20
18	SVCCTL_InitCustomSvc	21
19	Fehlermeldung CubelDE Console	22

Einleitung

Bluetooth Low Energy (BLE) ermöglicht eine effiziente, energiearme Datenübertragung für diverse Sensoranwendungen. Diese Arbeit untersucht die Nutzung einer BMI323-Inertialmesseinheit (IMU) zur Bewegungserkennung und Datenverarbeitung über Bluetooth. Ziel des Projekts war es, die auf dem Sensor implementierten Algorithmen zur Bewegungserkennung zu verstehen, zu dokumentieren und selbst zu implementieren. Neben der Analyse der hardwareseitigen Zusammensetzung und der Konfiguration der BLE-Schnittstellen lag ein Schwerpunkt auf der praktischen Implementierung und Auswertung der Daten. Dieses Projekt soll nicht nur die technische Machbarkeit demonstrieren, sondern auch ein tieferes Verständnis der zugrunde liegenden Technologien vermitteln. Die Ergebnisse zeigen eine zuverlässige BLE-Kommunikation und eine präzise Bewegungserkennung mithilfe der Funktionen des Sensors sowie eigener Implementierung. Die Erkenntnisse unterstreichen das Potenzial BLE-fähiger IMUs für Echtzeit-Bewegungserkennung mit Anwendungen in Bereichen wie Gesundheitsüberwachung und Sport.

Hardware Setup

Sensormodul

Das Sensormodul, mit dem gearbeitet wird, ist der BMI323 der Firma Bosch. Bei dem Gerät handelt es sich um die sogenannte Shuttle-Board-Version. Das Shuttle-Board kann verwendet werden, um verschiedene Funktionen des BMI323 zu testen. Es ermöglicht einen Zugang zu den Sensorpins über einen einfachen Socket und kann direkt in das Board eingesteckt werden. Es basiert auf einer 6-Achsen-IMU (Inertial Measurement Unit). Drei orthogonal zueinander angebrachte Beschleunigungssensoren, oft auch translatorische Sensoren oder Accelerometer genannt sowie drei orthogonal aufeinander stehende gyroskopische Sensoren, oft auch rotatorische Sensoren genannt. Damit liefert der Sensor drei Beschleunigungswerte und drei Winkelgeschwindigkeiten. Zudem gibt er Auskunft über den Akkuladestand, den Betriebsmodus, die eingestellte Datenrate und liefert einen Schrittzähler bereit.

Mikrocontroller

Der verbaute Mikrocontroller ist der STM32WB55, der aus 2 Kernen besteht, die über den Inter-Processor Communications Controller (IPCC) miteinander kommunizieren. Er verfügt über einen ARM Cortex-M4-Kern als Hauptprozessor mit einer Taktfrequenz von bis zu 64 MHz und einen ARM Cortex-M0+-Kern, der dediziert für den Betrieb von Funkprotokollen wie Bluetooth Low Energy (BLE) und IEEE 802.15.4 verwendet wird. Voraussetzung für die Verwendung ist das Übertragen der Bluetooth-Firmware auf den M0-Kern, die nur einmal während der Lebensdauer des Chips durchgeführt werden darf. Jeder Fehler während dieses Vorgangs, z. B. das Hochladen der Firmware in ein falsches Register, kann zu einem Ausfall des Chips führen und ihn für die Bluetooth-Kommunikation unbrauchbar machen. Des Weiteren besitzt der STM32WB55 einen 1 MB Flash-Speicher, 256 KB RAM und mehrere integrierte Schnittstellen für Peripherie. Zur Verbindung des Mikrocontrollers mit dem PC wird der ST-Link V2 verwendet, für den bereits ein speziell angefertigter Steckverbinder für die Leiterplatte entwickelt wurde.

Programmierung

Auslesen der Sensordaten

Das Auslesen der Sensordaten geschieht über Inter-Integrated Circuit (I²C). I²C ist ein Kommunikationsprotokoll für elektronische Bauteile, auf die wir mithilfe der HAL-Bibliothek in C zugreifen können. Diese wird vom Hersteller des Mikrocontrollers zur Verfügung gestellt. Betrachten wir die Methode `Read_Sensor_Data` in der Datei `BMI323_eigen.c` genauer, da hier die wichtigsten Daten zur schwellenwertbasierten Bewegungserkennung aus dem Sensor ausgelesen werden. Die Methode speichert zunächst die ersten vier Bytes aus dem FIFO-Füllstandsregister des Sensors in das Array `receivedData`. Aus den Bytes an den Indizes 2 und 3 wird ein Wert generiert, der den FIFO-Füllstand des Sensors wiedergibt. Dieser Wert gibt wieder, wie viele Wörter im FIFO-Datenregister enthalten sind. Da immer vier Wörter ein Datenpaket ergeben, müssen wir den Wert des FIFO-Füllstands durch vier teilen, um zu erfahren, wie viele Datenpakete sich im FIFO-Speicher befinden. Da wir dies nun wissen, können wir durch alle Datenpakete des FIFO-Datenregisters iterieren und die übermittelten Beschleunigungswerte für die translatorische X-, Y-, und Z-Achse jeweils in einer Variable zwischenspeichern. Dies geschieht, indem immer die ersten zehn Bytes eines Datenpaketes in das Array `receivedData` geschrieben werden, wobei die Bytes an den Indizes 2 bis 7 die Beschleunigungswerte und die Bytes an den Indizes 8 und 9 die Sensorzeit darstellen. Somit wurden die vorherigen vier Bytes für den FIFO-Füllstand überschrieben. Um zu verstehen, wie wir die übermittelten Daten des Sensors richtig interpretieren können, müssen wir genauer auf die Zeilen 197 bis 205 in Abbildung 1 eingehen. Alle Werte, die aus den Datenregistern des BMI323 ausgelesen werden, bestehen aus einem Least Significant Byte (LSB) und einem Most Significant Byte (MSB). Diese beiden Bytes werden zu einem 16-Bit-Wert kombiniert, wobei das MSB um acht Bits nach links verschoben und das LSB hinzugefügt wird:

$$\text{Messwert} = (\text{MSB} \ll 8) | \text{LSB}$$

Dies gilt nicht nur für die Beschleunigungswerte, sondern auch für andere Sensordaten wie die Gyroskopwerte, Temperaturmessungen und Zeitstempel. Alle diese Werte sind im Zweierkomplement-Format gespeichert, sodass auch negative Werte korrekt dargestellt werden können.

Was wir also erhalten, ist letztendlich jeweils ein Integer-Wert für die drei Beschleunigungsachsen des Sensors. Nun stellt sich die Frage, in welcher Einheit wir diese Integer-Werte interpretieren können. Der BMI323 kann in verschiedenen Messbereichen arbeiten ($\pm 2g$, $\pm 4g$, $\pm 8g$, $\pm 16g$). Das „g“ steht für die Erdbeschleunigung (Gravitationskraft) und ist eine gebräuchliche Einheit zur Angabe von Beschleunigungen in der Sensorik. $1g$ entspricht $9,81 \text{ m/s}^2$. In unserem Fall ist der Sensor in der Datei BMI323_eigen.c in der Methode „Configure_Accelerometer“ auf $\pm 16g$ konfiguriert.

```
87 // configure scale of the accelerometer
88 transmitData[0] |= (0b011 << 4); // 16 g range
```

Abbildung 1: Configuration_Accelerometer

Bei $\pm 16g$ wird der gesamte 16-Bit-Wertebereich auf $-16g$ bis $+16g$ aufgeteilt. Das bedeutet, der Wert -32768 entspricht $-16g$ und der Wert $+32767$ entspricht $+16g$. Um nun herauszufinden, welcher 16-Bit Wert aus LSB und MSB genau $1g$ entspricht, können wir den maximalen Wert, den wir mit 16-Bit-Zahlen im Zweierkomplement darstellen können (32767), durch die maximale Gravitationskraft ($16g$) teilen. Das Ergebnis ist 2048 . Das bedeutet, dass ein Messwert auf einer der Beschleunigungsachsen genau 2048 betragen muss, um $1g$ zu entsprechen. Alle endgültigen Messwerte der Beschleunigungsachsen müssten also durch 2048 geteilt werden, um die tatsächliche Gravitationskraft in g zu erhalten. Da aber im Code durch $2,048$ geteilt wird, erhalten wir Werte in milli-g anstatt in g . Tatsächlich wird aus Laufzeit Gründen sogar nur durch 2 geteilt aber das hat keinen gravierenden Einfluss auf die Bewegungserkennung.

```

169 void Read_Sensor_Data()
170 {
171     uint8_t receivedData[10]={0};
172     HAL_StatusTypeDef ret=0;
173     int16_t accelerationValueX=0;
174     int16_t accelerationValueY=0;
175     int16_t accelerationValueZ=0;
176     uint16_t fifoFillLevel = 0;
177     uint16_t loopCounter = 0;
178     uint16_t sensorTime = 0;
179
180     ret = HAL_I2C_Mem_Read(&hi2c3, DEVICE_ADDRESS, SENSOR_FIFO_FILL_LEVEL_READ, I2C_MEMADD_SIZE_8BIT, receivedData, 4, HAL_MAX_DELAY);
181     if(ret != HAL_OK)
182     {
183         printf("BMI323_eigen.c error in communication I2C\r\n");
184     } else
185     {
186         fifoFillLevel = (receivedData[3] << 8) | receivedData[2]; // reads the current fill level (amount of data words) in the FIFO
187     }
188     loopCounter = 0;
189     while(loopCounter < fifoFillLevel/4) // during each read, 4 words are read
190     {
191         ret = HAL_I2C_Mem_Read(&hi2c3, DEVICE_ADDRESS, SENSOR_FIFO_DATA, I2C_MEMADD_SIZE_8BIT, receivedData, 10, HAL_MAX_DELAY);
192         if(ret != HAL_OK)
193         {
194             printf("BMI323_eigen.c error in communication I2C\r\n");
195         } else
196         {
197             // I do not know why sensorTime is max 255, it should be a 16 bit word acc. to datasheet
198             sensorTime = (receivedData[8] << 8) | receivedData[9];
199             accelerationValueZ = (receivedData[7] << 8) | receivedData[6];
200             accelerationValueZ = accelerationValueZ/2; // actually /2,05 acc. to data sheet but dividing by two is faster
201
202             accelerationValueY = (receivedData[5] << 8) | receivedData[4];
203             accelerationValueY = accelerationValueY/2;
204
205             accelerationValueX = (receivedData[3] << 8) | receivedData[2];
206             accelerationValueX = accelerationValueX/2;
207
208             Determine_Activities(accelerationValueX, accelerationValueY, accelerationValueZ);
209
210             //Methode für eigene Characteristic
211             Determine_RawData(accelerationValueX, accelerationValueY, accelerationValueZ);
212         }
213         loopCounter++;
214     }
215 }

```

Abbildung 2: Read_Sensor_Data

SENSOR_FIFO_FILL_LEVEL_READ ist das Register, in das der FIFO-Füllstand geschrieben wird, und SENSOR_FIFO_DATA ist das Register, in das die Messwerte geschrieben werden. Gut zu erkennen ist auch, wie in der Schleife die Werte für die Beschleunigungsachsen aus dem Datenregister entnommen werden und die Bewegungserkennung aufgerufen wird.

Bewegungserkennung

Die gesamte Bewegungserkennung spielt sich in der Methode `Determine_Activities` ab, die in der Datei `eigen_activities.c` implementiert ist. Die Methode wird für jedes Triple ausgelesener Beschleunigungswerte aufgerufen und zählt somit auch sukzessive die Bewegungen des Motion Trackers. Wichtig für das Verständnis der schwellenwertbasierten Bewegungserkennung ist Abbildung 2. Die Algorithmen zur Bewegungserkennung gehen davon aus, dass die vertikale Achse die translatorische Y-Achse der Sensordaten darstellt. Diese darf zwar um 180° gedreht sein beziehungsweise auf dem Kopf stehen, aber es muss die Y-Achse sein, die vertikal durch den Sensor verläuft. Außerdem muss die translatorische X-Achse nach vorne oder hinten zeigen, also mit anderen Worten muss der Sensor auf der linken oder rechten Seite des Gürtels angebracht sein, wobei die Y-Achse nach oben oder unten zeigen muss.

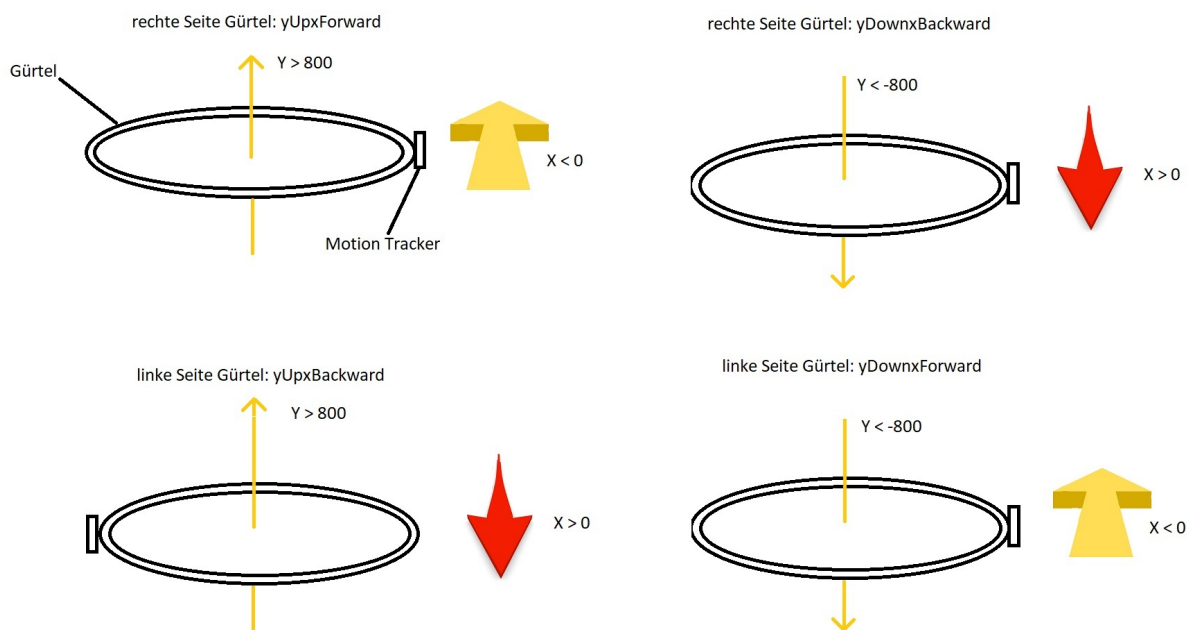


Abbildung 3: XY-Achsenausrichtung

Beim Aufruf der Methode werden zunächst die Variablen für die Startzeit festgelegt, sofern eine neue Messreihe beginnt. Anschließend werden die Arrays für die drei Beschleunigungsachsen befüllt, und für jedes der drei Arrays wird eine Summe gebildet, aus der später Durchschnittswerte errechnet werden, die wiederum wichtig für die Bewegungserkennung sind. Danach werden die Methoden zum Erfassen der Bewegungsmuster aufgerufen (Zeile 145 bis 151).

```

103 void Determine_Activities(int16_t accelerationValueX, int16_t accelerationValueY, int16_t accelerationValueZ)
104 {
105     RTC_TimeTypeDef rtcTime;
106     RTC_DateTypeDef rtcDate;
107
108     uint8_t yAxisOrientationNew = yUndefined;
109     uint8_t xyAxisOrientationNew = xyUndefined;
110     uint8_t postureNew = postureUndefined;
111     uint8_t shortTermActivityLevel = activityLevelZero;
112     uint8_t longTermActivityLevel = activityLevelZero;
113     int16_t meanX = 0;
114     int16_t meanY = 0;
115     int16_t meanZ = 0;
116
117     if(accelerationValueCounter == 0 && receivedSignalsCounter == 0)
118     { // Start of transmission unit
119         HAL_RTC_GetTime(&hrtc, &rtcTime, RTC_FORMAT_BIN);
120         HAL_RTC_GetDate(&hrtc, &rtcDate, RTC_FORMAT_BIN);
121         printf("\r\n%d:%d:%d:%lu eigen_activities.c accel counter start time\r\n",
122             rtcTime.Hours, rtcTime.Minutes, rtcTime.Seconds,
123             1000* (rtcTime.SecondFraction - rtcTime.SubSeconds) / (rtcTime.SecondFraction +1));
124         printf("%d:%d:%d eigen_activities.c accel counter start date\r\n", rtcDate.Year, rtcDate.Month, rtcDate.Date);
125         printf("Got datetimeStart. \r\n\r\n");
126         datetimeStart.Year = rtcDate.Year;
127         datetimeStart.Month = rtcDate.Month;
128         datetimeStart.Date = rtcDate.Date;
129         datetimeStart.Hours = rtcTime.Hours;
130         datetimeStart.Minutes = rtcTime.Minutes;
131         datetimeStart.Seconds = rtcTime.Seconds;
132     }
133
134     accX[accelerationValueCounter] = accelerationValueX;
135     accY[accelerationValueCounter] = accelerationValueY;
136     accZ[accelerationValueCounter] = accelerationValueZ;
137
138     sumAccX += accelerationValueX;
139     sumAccY += accelerationValueY;
140     sumAccZ += accelerationValueZ;
141
142     accelerationValueCounter++;
143
144     // Detect and count movements
145     Count_Jumps(accelerationValueY, yAxisOrientation);
146     Count_Runs(accelerationValueY, yAxisOrientation);
147     // Use function Count_WalkingSteps or Count_WalkingSteps_And_Squats to avoid double counting of walking steps!!!
148     // Count_WalkingSteps(accelerationValueY, yAxisOrientation);
149     Count_WalkingSteps_And_Squats(accelerationValueX, accelerationValueY, xyAxisOrientation, yAxisOrientation);
150     Count_Situps(accelerationValueX, accelerationValueY, xyAxisOrientation, yAxisOrientation);
151     Count_Pushups(accelerationValueX, accelerationValueY, accelerationValueZ, xyAxisOrientation);

```

Abbildung 4: Determine_Activities

Sobald eine Messreihe vollständig ist (256 Werte), beginnt das eigentliche Speichern der Bewegungserkennung, was jedoch in eine gewisse Logik eingebettet ist. Zunächst werden die Durchschnittswerte der Beschleunigungsachsen berechnet, und mithilfe dieser Werte wird die Y-Achsenausrichtung, die XY-Achsenausrichtung sowie die Haltung bestimmt.

```
156 // Signal is complete
157 if(accelerationValueCounter == ACCELEROMETER_VALUES_PER_SIGNAL)
158 {
159     accelerationValueCounter = 0;
160     receivedSignalsCounter ++;
161     // Calculate mean for each axis (meanX = sumAccX / ACCELEROMETER_VALUES_PER_SIGNAL)
162     meanX = sumAccX >> 8;
163     meanY = sumAccY >> 8;
164     meanZ = sumAccZ >> 8;
165
166     sumAccX = 0;
167     sumAccY = 0;
168     sumAccZ = 0;
169
170     yAxisOrientationNew = Get_Y_Axis_Orientation(meanY);
171     xyAxisOrientationNew = Get_XY_Axis_Orientation(meanX, meanY);
172     postureNew = Get_Posture(meanX, meanY, meanZ, xyAxisOrientation);
```

Abbildung 5: Determine_Activities

Für die Y-Achsenausrichtung wird geprüft, ob die Beschleunigungswerte der Y-Achse im Durchschnitt größer als 800 milli-g oder kleiner als -800 milli-g sind. Da die Erdbeschleunigung dauerhaft auf die Beschleunigungsachse wirkt (ca. 1000 milli-g), welche vertikal ausgerichtet ist, und da der Sensor so am Körper angebracht ist, dass die Y-Achse diejenige ist, die vertikal ausgerichtet ist, muss der Durchschnitt der Beschleunigungswerte für die Y-Achse außerhalb des Intervalls [-800, 800] liegen. Ein Wert innerhalb dieses Intervalls würde bedeuten, dass die Y-Achse einen Schiefstand hat, da sich ein zu großer Teil der Gravitationskraft auf eine andere Achse verlagert hat.

```
229 uint8_t Get_Y_Axis_Orientation(int16_t meanY)
230 {
231     if (meanY > 800) { return yUp; }
232     else if (meanY < -800) { return yDown; }
233     else { return yUndefined; }
234 }
```

Abbildung 6: Get_Y_Axis_Orientation

Als nächstes wird in Zeile 171 in `Determine_Activities` die XY-Achsenausrichtung ermittelt. Hierbei wird die gleiche Prüfung wie für die Y-Achsenausrichtung verwendet aber zusätzlich wird geprüft ob der Durchschnitt der Beschleunigungswerte der X-Achse größer oder kleiner 0 ist um zu ermitteln in welche Richtung sie zeigt. Um das zu verstehen müssen wir uns folgendes vor Augen führen. Wenn wir das typische vorwärts Laufen als Beispiel nehmen, dann lässt sich die Bewegung in 3 Abschnitte aufteilen. Die X-Achse ist nach horizontal vorne gerichtet.

1. **Anlaufen**

Die Person drückt sich nach vorne ab weshalb der Sensor eine positive Beschleunigung auf der X-Achse misst.

2. **Gleichmäßiges Laufen**

Die Person bewegt sich mit konstanter Geschwindigkeit.

Da keine weitere Beschleunigung erfolgt, misst der Sensor nahe 0 milli-g auf der X-Achse.

3. **Abbremsen**

Wenn die Person langsamer wird oder stoppt, drückt sie ihren Körper nach hinten. Der Sensor misst eine negative Beschleunigung auf der X-Achse.

Warum ist `meanX` jetzt aber im Durchschnitt negativ wenn wir ja vermehrt Bewegungen nach vorne machen? Das hat zwei Hauptgründe. Zum einen gibt es beim Laufen viele kurze, starke Vorwärts-Beschleunigungen, aber die negative Beschleunigung (beim Bremsen) hält oft länger an. Wenn man z.B. rennt und abbremst, misst der Sensor während des Bremsens eine negative X-Beschleunigung für längere Zeit. Ein weiterer, schwerwiegenderer Grund ist allerdings der kleine Kippwinkel nach vorne, der in fast allen Bewegungsmustern entsteht. Die dadurch entstehende Gravitationskraft auf der X-Achse verrät in welche Richtung sie zeigt. Dadurch kann wie in Abbildung 7 ermittelt werden, welche XY-Achsenausrichtung wir haben und wir bekommen die vier verschiedenen Möglichkeiten, die auch Abbildung 3 schon gezeigt hat.

```

236 uint8_t Get_XY_Axis_Orientation(int16_t meanY, int16_t meanX)
237 {
238     if (meanY > 800 && meanX < 0)
239     {
240         return yUpXForward;
241     }
242     if (meanY > 800 && meanX > 0)
243     {
244         return yUpXBackward;
245     }
246     if (meanY < -800 && meanX < 0)
247     {
248         return yDownXForward;
249     }
250     if (meanY < -800 && meanX > 0)
251     {
252         return yDownXBackward;
253     }
254     return xyUndefined;
255 }

```

Abbildung 7: Get_XY_Axis_Orientation

Der letzte wichtige Punkt bevor die Daten der Bewegungserkennung gespeichert werden ist die Ermittlung der Körperhaltung (Zeile 172 in Determine_Activities). Die Methode Get_Posture versucht zu ermitteln ob die Person eine aufrechte Körperhaltung hat oder ob sie liegt. Sofern sie liegt, soll zusätzlich ermittelt werden in welcher Position die Person liegt. Hierbei wird wieder anhand der durchschnittlichen Beschleunigungswerte der Y-Achse (meanY) ermittelt, ob diese im Intervall [-600, 600] liegen. Wenn so wenig Gravitation auf der vertikalen Beschleunigungsachse vorhanden ist, muss die Y-Achse so stark geneigt sein, dass man von einer liegenden Position sprechen kann.

```
964 uint8_t Get_Posture(int16_t meanX, int16_t meanY, int16_t meanZ, uint8_t xyAxisOrientation)
965 {
966     if(meanY >= MEAN_Y_LIMIT_FOR_POSTURE_UPRIGHT_Y_UP || meanY <= MEAN_Y_LIMIT_FOR_POSTURE_UPRIGHT_Y_DOWN)
967     {
968         return postureUpright;
969     }
970
971     if(xyAxisOrientation == xyUndefined)
972     {
973         if(meanY < MEAN_Y_LIMIT_FOR_POSTURE_UPRIGHT_Y_UP && meanY > MEAN_Y_LIMIT_FOR_POSTURE_UPRIGHT_Y_DOWN)
974         {
975             return postureLying;
976         }
977     }
```

Abbildung 8: Get_Posture

Hierbei ist uns aufgefallen, dass die Methode Get_Posture nur den Teil ausführen kann der auch in der Abbildung 8 zu sehen ist. Der Rest der Methode kann in keinem Fall ausgeführt werden, weshalb die Rückgabewerte der Methode nur postureUpright oder postureLying sein können.

Kommen wir zu dem Teil in dem die Aktivitätsdaten und die Ergebnisse der Bewegungserkennung gespeichert werden. Der wichtigste Punkt hierbei ist der Aufruf der Methode Save_Data.

```
197 // Posture changed
198 if(postureOld != postureNew)
199 {
200     shortTermActivityLevel = Get_Short_Term_Activity_Level();
201     longTermActivityLevel = Get_Long_Term_Activity_Level();
202     Get_Average_Speed();
203     Save_Data(shortTermActivityLevel, longTermActivityLevel);
204     Get_maxAbsoluteOverallAcceleration_And_sumMeanAbsoluteOverallAcceleration(meanX, meanY, meanZ);
205
206     // Set variables for new signal
207     postureOld = postureNew;
208     receivedSignalsCounter = 1;
209     datetimeStart = datetimeEnd;
210 }
211
212 // Posture didn't change
213 else
214 {
215     Get_maxAbsoluteOverallAcceleration_And_sumMeanAbsoluteOverallAcceleration(meanX, meanY, meanZ);
216     shortTermActivityLevel = Get_Short_Term_Activity_Level();
217     longTermActivityLevel = Get_Long_Term_Activity_Level();
218     Get_Average_Speed();
219     // No activity
220     if(shortTermActivityLevel == activityLevelZero && longTermActivityLevel == activityLevelZero)
221     {
222         noActivitySignalsCounter ++;
223         // Limit for lack of activity is reached
224         if(receivedSignalsCounter == SIGNALS_PER_TRANSMISSION_CASE_NO_ACTIVITY)
225         {
226             Save_Data(shortTermActivityLevel, longTermActivityLevel);
227             postureOld = postureUndefined;
228             receivedSignalsCounter = 0;
229         }
230         if(noActivitySignalsCounter == MAX_NUMBER_NO_ACTIVITY_SIGNALS)
231         {
232             printf("SEIT 10 STUNDEN KEINE AKTIVITÄT!!!"); // Replace by a method to send a warning
233             noActivitySignalsCounter = 0;
234         }
235     }
236     // One activity level is greater zero
237     else
238     {
239         // Transmission unit is complete
240         if(receivedSignalsCounter >= SIGNALS_PER_TRANSMISSION)
241         {
242             shortTermActivityLevel = Get_Short_Term_Activity_Level();
243             longTermActivityLevel = Get_Long_Term_Activity_Level();
244             Get_Average_Speed();
245             Save_Data(shortTermActivityLevel, longTermActivityLevel);
246             postureOld = postureUndefined;
247             receivedSignalsCounter = 0;
248             noActivitySignalsCounter = 0;
249         }
250     }
251 }
```

Abb. 9: Determine_Activities

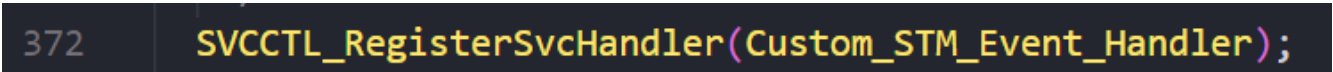
Die Methode beginnt mit der Überprüfung, ob die vorherige Haltung (`postureOld`) undefiniert ist. Falls dies der Fall ist (immer für den ersten Wert einer neuen Messreihe), wird sie auf die aktuelle Haltung (`postureNew`) gesetzt. Wenn die Haltung definiert war, wird überprüft, ob sie sich geändert hat. Bei einer Haltungsänderung werden alle Aktivitätsdaten der vorherigen Periode gespeichert, und eine neue Periode beginnt. Bei keiner Haltungsänderung wird überprüft, ob eine Aktivität erkannt wurde oder ob eine längere Inaktivität vorliegt. Bei Inaktivität wird eine Warnung ausgegeben, wenn die maximale Anzahl von Inaktivitätssignalen erreicht ist. Bei Aktivität werden die Daten gespeichert, sobald eine bestimmte Anzahl von Signalen erreicht ist. Bei erfolgreichem Durchlaufen der Methode werden die kurzfristigen und langfristigen Aktivitätsdaten, die durchschnittliche Geschwindigkeit, die gesamten Daten zu den verschiedenen Bewegungsmustern und die erfasste Endzeit der Datenübertragung im Array `savedData` hinterlegt. Die Daten in `savedData` werden dann regelmäßig von einer `Characteristic` ausgelesen aber darauf gehen wir in dem Abschnitt für das Erstellen und Aktualisieren einer `Characteristic` noch genauer ein.

Anhand der Methode Count_Jumps wollen wir nochmal genauer darauf eingehen, wie mithilfe von Schwellenwerten ein bestimmtes Bewegungsmuster erkannt werden kann. Die Methode orientiert sich an 2 wichtigen Schwellenwerten. Der erste ist das feste Abdrücken vom Boden während eines Sprunges, wofür der Schwellenwert auf 4000 milli-g festgelegt wurde. Wenn wir uns vom Boden nach oben abdrücken entsteht eine verstärkte Gravitation auf der Y-Achse in die positive Richtung. Der zweite Schwellenwert der verwendet wird, ist der kurze freie Fall nach dem ein Sprung getätigt wurde. Bei einem freien Fall wirkt keine Beschleunigung mehr gegen die Gravitation, die standardmäßig auf der Y-Achse nach unten wirkt. Dadurch bekommen wir einen Wert nahe null und es wurde hierfür als Schwellenwert 100 verwendet. Wenn also eine starke Beschleunigung nach oben, gefolgt von einer Beschleunigung nahe null entsteht, wird ein Sprung gezählt. Damit die Methode nicht dauerhaft auf die Bestätigung des Sprunges durch einen freien Fall wartet, wird nach 40 Messwerten trotzdem ein Sprung gezählt.

Erstellen und Aktualisieren einer Characteristic

Da wir uns einerseits tiefer damit beschäftigen wollten, wie es unseren Kollegen jetzt möglich ist, von einem beliebigen Computer die Ergebnisse der Bewegungserkennung auszulesen und wir andererseits auch neue Rohdaten zum Testen der eigen implementierten Bewegungserkennung benötigten, haben wir eine neue Characteristic angelegt die uns Rohdaten liefern sollte. Wir gehen in der chronologischen Reihenfolge durch die einzelnen Methodenaufrufe, die zum Auslesen der Characteristic führen. Dabei zeigen wir die Anpassungen die benötigt werden um eine neue Characteristic anzulegen die uns Rohdaten liefert.

1. Aufruf des Event Handlers



```
372 SVCCTL_RegisterSvcHandler(Custom_STM_Event_Handler);
```

Abbildung 10: cusotm_stm.c

Die Methode SVCCTL_RegisterSvcHandler dient dazu eine Verbindung zwischen dem BLE-Stack und der Anwendung herzustellen. Sie registriert eine Funktion, die auf ein bestimmtes Bluetooth Event (Leseanfrage auf Service) aufgerufen wird. Daher ist sie die Brücke zwischen dem Device, der sich via Bluetooth mit dem Motion Tracker verbindet und der Programmierung, die auf dem Mikrocontroller liegt. Die Funktion, die durch das Bluetooth Event aufgerufen wird, ist die Event Handler Methode Custom_STM_Event_Handler, welche auch der Übergabeparameter von der Methode SVCCTL_RegisterSvcHandler ist. SVCCTL_RegisterSvcHandler gehört zu den Funktionen des BLE-Stack welche bereits vom Hersteller zur Verfügung gestellt wird.

2. Event Handler

Nach dem das Event-Paket vom BLE-Stack übergeben wird, wird der Event-Typ analysiert, um festzustellen, um welche Art von Event es sich handelt. Die relevanten Informationen (Handle-Nummer und Daten) werden aus dem Event-Paket extrahiert und abhängig vom Event-Typ, wird die entsprechende Logik ausgeführt. Die Methode prüft, ob das Event eine Leseanfrage für unsere Characteristic ist und wenn dies der Fall ist, wird eine Notification erstellt, die den Event-Typ enthält. Die Notification wird dann an die Funktion Custom_STM_App_Notification übergeben. Die nachstehende Abbildung ist die Stelle, in der dies für die Characteristic der Bewegungserkennung geschieht. Hierbei ist zu sehen wie wir die Logik erweitert haben, so dass unsere eigen angelegte Characteristic abgerufen werden kann.

```
231 case ACI_GATT_READ_PERMIT_REQ_VSEVT_CODE :
232     /* USER CODE BEGIN EVT_BLUE_GATT_READ_PERMIT_REQ_BEGIN */
233     printf("custom_stm.c read permit request\r\n");
234     /* USER CODE END EVT_BLUE_GATT_READ_PERMIT_REQ_BEGIN */
235     read_req = (aci_gatt_read_permit_req_event_rp0*)blecore_evt->data;
236
237     // ----- Characteristic 1 / Activity Data -----
238     if (read_req->Attribute_Handle == (CustomContext.CustomSactdatHdle + CHARACTERISTIC_VALUE_ATTRIBUTE_OFFSET))
239     {
240         return_value = SVCCTL_EvtAckFlowEnable;
241         /*USER CODE BEGIN CUSTOM_STM_Service_1_Char_1_ACI_GATT_READ_PERMIT_REQ_VSEVT_CODE_1 */
242         Notification.Custom_Evt_Opcode = CUSTOM_STM_SACTDAT_READ_EVT;
243         Custom_STM_App_Notification(&Notification);
244         /*USER CODE END CUSTOM_STM_Service_1_Char_1_ACI_GATT_READ_PERMIT_REQ_VSEVT_CODE_1*/
245         aci_gatt_allow_read(read_req->Connection_Handle);
246         /*USER CODE BEGIN CUSTOM_STM_Service_1_Char_1_ACI_GATT_READ_PERMIT_REQ_VSEVT_CODE_2 */
247
248         /*USER CODE END CUSTOM_STM_Service_1_Char_1_ACI_GATT_READ_PERMIT_REQ_VSEVT_CODE_2*/
249     } /* if (read_req->Attribute_Handle == (CustomContext.CustomSactdatHdle + CHARACTERISTIC_VALUE_ATTRIBUTE_OFFSET))*/
250
251     // ----- Für eigen angelegte Characteristic -----
252     else if (read_req->Attribute_Handle == (CustomContext.CustomNewCharHdle + CHARACTERISTIC_VALUE_ATTRIBUTE_OFFSET))
253     {
254         return_value = SVCCTL_EvtAckFlowEnable;
255         Notification.Custom_Evt_Opcode = CUSTOM_STM_NEW_CHAR_READ_EVT;
256         Custom_STM_App_Notification(&Notification);
257         aci_gatt_allow_read(read_req->Connection_Handle);
258     }
```

Abbildung 11: Custom_STM_Event_Handler

3. Notification

An dieser Stelle im Code wird untersucht um welchen Event-Typen es sich handelt und die entsprechende Logik wird aufgerufen. Die Funktion dient der Entkopplung von BLE-Stack und Anwendungslogik und durch die zentrale Event-Verteilung wird die Wartbarkeit, Erweiterbarkeit und die Lesbarkeit des Systems erhöht. Für die bereits implementierte Characteristic mit der Bewegungserkennung ist der zweite Case in dem dann die Methode aufgerufen wird, in der die Daten aus savedDate ausgelesen werden. An dieser Stelle haben wir einen Case für die Characteristic angelegt in der die Rohdaten ausgelesen werden sollen.

```
133  /* Functions Definition -----*/
134  void Custom_STM_App_Notification(Custom_STM_App_Notification_evt_t *pNotification)
135  {
136      /* USER CODE BEGIN CUSTOM_STM_App_Notification_1 */
137      printf("custom_app.c app_notification\r\n");
138
139      /* USER CODE END CUSTOM_STM_App_Notification_1 */
140      switch (pNotification->Custom_Evt_Opcode)
141      {
142          /* USER CODE BEGIN CUSTOM_STM_App_Notification_Custom_Evt_Opcode */
143
144          /* USER CODE END CUSTOM_STM_App_Notification_Custom_Evt_Opcode */
145
146          /* eigene Characteristic */
147          case CUSTOM_STM_NEW_CHAR_READ_EVT:
148              /* USER CODE BEGIN CUSTOM_STM_SACTDAT_READ_EVT */
149              printf("custom_app.c s new char read event\r\n");
150              Process_Read_Request_For_Data_v2();
151              /* USER CODE END CUSTOM_STM_SACTDAT_READ_EVT */
152              break;
153
154          /* activityData */
155          case CUSTOM_STM_SACTDAT_READ_EVT:
156              /* USER CODE BEGIN CUSTOM_STM_SACTDAT_READ_EVT */
157              printf("custom_app.c s activity data read event\r\n");
158              Process_Read_Request_For_Data();
159              /* USER CODE END CUSTOM_STM_SACTDAT_READ_EVT */
160              break;
```

Abbildung 12: Custom_STM_App_Notification

4. Vorbereitung der Daten

In der Methode `Process_Read_Request_For_Data` werden dann die 36 Werte aus `savedData` in ein Array kopiert, welches dann von der `Characteristic` ausgelesen werden soll. Innerhalb dieser Methode wird dann `Ble_Update_Characteristic` aufgerufen. Diese Schicht dient wieder als Vermittler zwischen BLE-Stack und Anwendungslogik.

```
495 void Process_Read_Request_For_Data(void)
496 {
497     printf("custom_app.c Process_Read_Request_For_Data\r\n");
498     // this function is called each time the host reads the characteristic "savedActData"
499     // after this function is executed, the value of the characteristic will be returned to the host
500
501     uint8_t transmitData[36] = {0}; // array to be read, the number of bytes to be read are specified in CubeIDE
502     uint8_t returnValue = 0;
503     //
504     // filling the dataBuffer with dummy data
505     for(uint8_t i=0; i<36; i++)
506     {
507         transmitData[i]=savedData[nextOut][i];
508     }
509     // updates the value of the characteristic to be read
510     returnValue = Ble_Update_Characteristic(CUSTOM_STM_SACTDAT, transmitData, sizeof(transmitData));
511
512     if(returnValue != BLE_STATUS_SUCCESS)
513     {
514         printf("custom_app.c updating data char NOT successful\r\n");
515     }
516
517     if(returnValue == BLE_STATUS_SUCCESS)
518     {
519         Move_NextOut();
520     }
521 }
```

Abb. 13: `Process_Read_Request_For_Data`

An dieser Stelle haben wir dann die Methode zum Auslesen der Rohdaten implementiert. Hierbei wollen wir die `Characteristic` ein Array auslesen lassen, welches 256 Tripels an rohen Beschleunigungswerten beinhaltet.

```
477 void Process_Read_Request_For_Data_v2(void)
478 {
479     uint8_t transmitRawData[768] = {0};
480     uint8_t returnValue = 0;
481
482     for(uint8_t i = 0; i < 768; i++)
483     {
484         transmitRawData[i] = rawData[i];
485     }
486     returnValue = Ble_Update_Characteristic(CUSTOM_STM_EIGEN_ACT, transmitRawData, sizeof(transmitRawData));
487     if(returnValue != BLE_STATUS_SUCCESS)
488     {
489         printf("custom_app.c updating data char NOT successful\r\n");
490     }
491 }
```

Abb. 14: `Process_Read_Request_For_Data` eigen

Wie schreiben wir die Daten in das Array `rawData`? Wie schon in Abbildung 2, Zeile 210 sichtbar wurde, rufen wir beim Auslesen der Sensordaten die Methode `Determine_RawData` auf. Hierbei schreiben wir im Prinzip einfach nur die ausgelesenen Beschleunigungswerte in das Array `rawData`, welches global angelegt ist.

```
92 void Determine_RawData(int16_t accelerationValueX, int16_t accelerationValueY, int16_t accelerationValueZ) {
93
94     rawData[3 * accelerationValueCounterForRawData + 1] = accelerationValueX;
95     rawData[3 * accelerationValueCounterForRawData + 2] = accelerationValueY;
96     rawData[3 * accelerationValueCounterForRawData + 3] = accelerationValueZ;
97
98     accelerationValueCounterForRawData ++;
99
100 }
```

Abb. 15: Determine_RawData

5. Update der Characteristic

In der Methode `Ble_Update_Characteristic`, wird dann die Methode `Custom_STM_App_Update_Char` aufgerufen.

```
197 uint8_t Ble_Update_Characteristic(uint8_t characteristic, uint8_t *data, uint8_t numberOfBytesToTransmit)
198 {
199     uint8_t ret = 0;
200
201     ret = Custom_STM_App_Update_Char(characteristic, (uint8_t *)data); // update value for read / notify
```

Abb. 16: Ble_Update_Characteristic

Im letzten Schritt, dem Aktualisieren der Characteristic haben wir einen neuen Case für unsere eigen angelegte Characteristic erstellt.

```
615 tBleStatus Custom_STM_App_Update_Char(Custom_STM_Char_Opcode_t CharOpcode, uint8_t *pPayload)
616 {
617     tBleStatus ret = BLE_STATUS_INVALID_PARAMS;
618     /* USER CODE BEGIN Custom_STM_App_Update_Char_1 */
619
620     /* USER CODE END Custom_STM_App_Update_Char_1 */
621
622     switch (CharOpcode)
623     {
624         // Eigen angelegte Characteristic
625         case CUSTOM_STM_EIGEN_ACT:
626             ret = aci_gatt_update_char_value(CustomContext.CustomActivitydatHdle,
627                                             CustomContext.CustomNewCharHdle,
628                                             0,
629                                             SizeNewChar,
630                                             (uint8_t *) pPayload);
631
632         case CUSTOM_STM_SACTDAT:
633             ret = aci_gatt_update_char_value(CustomContext.CustomActivitydatHdle,
634                                             CustomContext.CustomSactdatHdle,
635                                             0, /* charValOffset */
636                                             SizeSactdat, /* charValueLen */
637                                             (uint8_t *) pPayload);
638
639             if (ret != BLE_STATUS_SUCCESS)
640             {
641                 APP_DBG_MSG(" Fail : aci_gatt_update_char_value SACTDAT command, result : 0x%x \n\r", ret);
642             }
643             else
644             {
645                 APP_DBG_MSG(" Success: aci_gatt_update_char_value SACTDAT command\n\r");
646             }
647
648             /* USER CODE BEGIN CUSTOM_STM_App_Update_Service_1_Char_1*/
649
650             /* USER CODE END CUSTOM_STM_App_Update_Service_1_Char_1*/
651
652             break;
653     }
654 }
```

Abb. 17: Custom_STM_App_Update_Char

Hierfür mussten wir die Characteristic auch initialisieren. Wir haben der Characteristic eine uuid gegeben, die Größe festgelegt und die Characteristic dem gleichen Service zugeordnet wie der Characteristic für die Bewegungserkennung.

```
358 void SVCCTL_InitCustomSvc(void)
414
415 /**
416  * eigene Characteristic
417  */
418 uuid.Char_UUID_16 = 0x2ad2;
419 ret = aci_gatt_add_char(CustomContext.CustomActivitydatHdle,
420                        UUID_TYPE_16, &uuid,
421                        768,
422                        CHAR_PROP_READ | CHAR_PROP_NOTIFY,
423                        ATTR_PERMISSION_NONE,
424                        GATT_NOTIFY_READ_REQ_AND_WAIT_FOR_APPL_RESP,
425                        0x10,
426                        CHAR_VALUE_LEN_CONSTANT,
427                        &(CustomContext.CustomNewCharHdle));
428 if (ret != BLE_STATUS_SUCCESS)
429 {
430     APP_DBG_MSG(" Fail : aci_gatt_add_char command : NEWCHAR, error code: 0x%x \n\r", ret);
431 }
432 else
433 {
434     APP_DBG_MSG(" Success: aci_gatt_add_char command : NEWCHAR \n\r");
435 }
436
437 /**
438  * savedActData
439  */
440 uuid.Char_UUID_16 = 0x2ad3;
441 ret = aci_gatt_add_char(CustomContext.CustomActivitydatHdle,
442                        UUID_TYPE_16, &uuid,
443                        SizeSactdat,
444                        CHAR_PROP_READ | CHAR_PROP_NOTIFY,
445                        ATTR_PERMISSION_NONE,
446                        GATT_NOTIFY_READ_REQ_AND_WAIT_FOR_APPL_RESP,
447                        0x10,
448                        CHAR_VALUE_LEN_CONSTANT,
449                        &(CustomContext.CustomSactdatHdle));
```

Abb. 18: SVCCTL_InitCustomSvc

Flash Firmware

Nun haben wir die Firmware um eine neue Bluetooth-Characteristic erweitert und wollen diese natürlich auch auslesen und testen. Die Programmierung eines STM32-Mikrocontrollers erfolgt in der Entwicklungsumgebung STM32CubeIDE in mehreren Schritten. Dabei werden der Quellcode kompiliert, das resultierende Programm generiert und anschließend auf den Mikrocontroller übertragen. Sofern der Code fehlerfrei ist, wird eine PROJECT-Datei und eine ausführbare .elf-Datei erstellt, welche dann in den internen Flash-Speicher des Mikrocontrollers geschrieben wird. Der Mikrocontroller kann anschließend automatisch gestartet werden, um die neue Firmware auszuführen.

Analyse der Stromversorgung bei der Übertragung der Firmware

Während der Übertragung der Firmware auf den STM32-Mikrocontroller trat ein Problem auf. Die Fehlermeldung in der Konsole der STM32CubeIDE deutete darauf hin, dass keine stabile Stromversorgung zum Mikrocontroller hergestellt werden konnte. Ein möglicher Grund für dieses Problem ist eine unzureichende Stromversorgung des Boards. Da das Board über den USB-C-Anschluss mit Strom versorgt wird, wurde die Spannungsversorgung entlang des Strompfads überprüft. Besonders auffällig war hierbei die D1-Diode, die direkt hinter dem USB-C-Anschluss auf dem Board liegt und vermutlich zur Spannungsregelung dient. Mit einem Multimeter wurde die Spannung an den Anschlüssen der D1-Diode gemessen. Dabei wurde ein ungewöhnlicher Wert festgestellt, der nicht plausibel für eine funktionierende Diode ist, was darauf hindeuten könnte, dass die Diode defekt ist oder einen Kurzschluss aufweist. Falls die D1-Diode tatsächlich beschädigt ist, könnte dies zu einem erhöhten Widerstand oder einer vollständigen Unterbrechung der Stromversorgung führen. Dadurch würde der Mikrocontroller nicht ausreichend mit Spannung versorgt werden, sodass die Übertragung fehlschlägt.

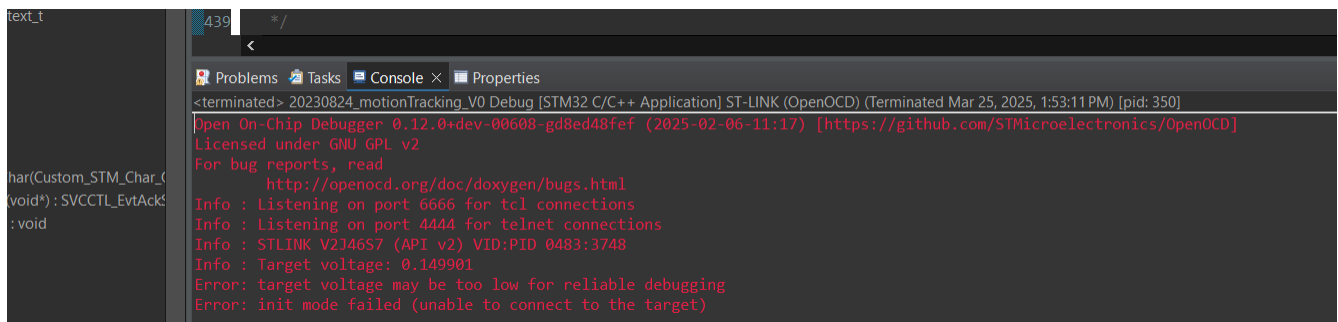


Abb. 19: Fehlermeldung CubeIDE Console

Für Sensordata.py

Importierte Bibliotheken

Um diverse Funktionalitäten zu ermöglichen, werden zu Beginn des Skripts zwei wichtige Bibliotheken importiert: `dataclass` und `datetime`. Diese Bibliotheken erleichtern den Umgang mit Daten und Zeitstempeln und tragen dazu bei, dass der Code klarer, kürzer und fehlerresistenter wird. Die erste Bibliothek `dataclass` in Python stellt eine Methode dar, um Klassen zu definieren, deren Aufgabe es ist, Dateien zu speichern. Sie übernimmt die Automatisierung der Erstellung grundlegender Methoden, wie etwa des Initialisierers (`__init__`). Die zweite Bibliothek, `datetime`, ist ein Modul, welches dem User eine Vielzahl von Klassen und Funktionen bereitstellt, um mit Datumsangaben, als auch mit Zeitangaben arbeiten zu können. `Datetime`, welches sowohl das Datum als auch die Uhrzeit speichert und Operationen wie Zeitdifferenzen, Formatierungen und Umwandlungen ermöglicht. In diesem Skript wird die `datetime`-Klasse verwendet, um die Start- und Endzeiten der erfassten Sensordaten zu speichern und die Dauer der Aktivitätsperioden zu berechnen. Die Pythondatei hat zwei Hauptklassen: `Sensordata` und `SensordataList`, welche für den Benutzer die Erfassung, Speicherung und Analysieren von Sensordaten ermöglichen.

Klasse Sensordata

Die Klasse Sensordata ist eine der beiden Hauptkomponenten des Skripts. Sie enthält von relevanten Attributen, welches Elementar für die Sensordaten eines Benutzers sind. Jedes Objekt dieser Klasse repräsentiert eine spezifische Aufzeichnung von Aktivitätsdaten. Dazu gehören folgende Datentypen: die Start- und Endzeiten (startTime und endTime), die mit der datetime-Klasse gespeichert werden. Diese Zeitstempel werden für jede Aktivität gesetzt und dienen als Grundlage für die Berechnung der Dauer der Aktivität in Sekunden. Weitere Attribute geben dem User Informationen über Körperhaltung (posture), die Aktivitätslevel auf kurzfristiger und langfristiger Basis (shortTermActivityLevel und longTermActivityLevel) und verschiedene Messwerte zu körperlichen Aktivitäten, wie z.B. Anzahl der Sprünge (jumps), Anzahl der Läufe (runs), Anzahl der Schritte beim Gehen (walkingSteps), Anzahl der Squats, Situps und Pushups, sowie die durchschnittliche Geschwindigkeit (averageSpeed). Es gibt auch das Attribut stepCounter, das die Gesamtzahl der Schritte während des Zeitraums zählt, wo der Sensor aktiv ist. Die Methode __init__ der Klasse ist dafür verantwortlich, die Daten zu verarbeiten, die beim Erstellen eines neuen Objekts übergeben werden. Die übergebene Zeichenkette, die die Sensordaten enthält, wird in einzelne Werte aufgeteilt und die entsprechenden Attribute des Objekts werden mit diesen Werten gefüllt. Dabei wird besonders darauf geachtet, dass die Zeitstempel korrekt in datetime-Objekte umgewandelt werden, und alle anderen Werte als Ganzzahlen gespeichert werden. Die Dauer der Aktivität wird anhand der Differenz zwischen startTime und endTime berechnet und in Sekunden gespeichert.

Klasse `SensordataList`

Die zweite wichtige Klasse im Skript ist `SensordataList`. Diese Klasse verwaltet eine Sammlung von `Sensordata`-Objekten und stellt dem User nützliche Methoden zur Verfügung, um auf die Daten zuzugreifen und Berechnungen durchzuführen. Bei der Initialisierung wird die Datei `ReaderData.txt` geöffnet, aus der Zeile für Zeile die Sensordaten ausgelesen werden. Für jede Zeile wird ein neues `Sensordata`-Objekt erstellt und in der Liste `sensorList` gespeichert. Auf diese Weise können alle Sensordaten, die in der Datei gespeichert sind, in einer einzigen Liste verwaltet werden. Eine der Hauptmethoden der Klasse ist `writeData`, die es ermöglicht, neue Sensordaten zu der Datei hinzuzufügen und gleichzeitig die Liste `sensorList` zu erweitern. Diese Methode stellt sicher, dass neue Daten sowohl auf der Festplatte gespeichert als auch im Arbeitsspeicher verfügbar sind. Dies ist besonders praktisch, wenn regelmäßig neue Sensordaten hinzukommen und gespeichert werden müssen. Die Methode `maxForDay` ermöglicht es, den höchsten Wert eines bestimmten Attributs (wie z. B. die Anzahl der Sprünge oder Läufe) für einen bestimmten Tag zu ermitteln. Diese Methode filtert zunächst alle Daten für das angegebene Datum heraus und berechnet dann den maximalen Wert des angegebenen Attributs. Ähnlich funktioniert die Methode `countForDay`, die die Werte eines bestimmten Attributs summiert. Beide Methoden verwenden intern die Methode `filterForDay`, die dafür zuständig ist, die Sensordaten nach dem angegebenen Datum zu filtern und eine Liste der entsprechenden Daten zurückzugeben.

Für Graphics.py

In diesem Code wird eine GUI mit der tkinter-Bibliothek erstellt und es werden verschiedene Aktivitätsdaten wie Sprünge, Schritte, Squats, Situps, Pushups und die durchschnittliche Geschwindigkeit graphisch dargestellt. Benutzer können aus einem Dropdown-Menü eine Aktivität auswählen, und daraufhin wird ein Diagramm für die jeweilige Aktivität angezeigt. Die Diagramme werden mit Matplotlib erstellt, sowie in eine die tkinter-Oberfläche integriert.

1. Elementare Bibliotheken, die von Gebrauch gemacht werden:

- `sys`: Programm wird beendet, wenn der Benutzer auf "Beenden" klickt.
- `tkinter (tk)`: Eine Bibliothek, die eine grafische Benutzeroberfläche (GUI) erstellt.
- `matplotlib.pyplot (plt)`: Diagramm zu erstellen.
- `FigureCanvasTkAgg`: Diese Klasse ermöglicht die Einbettung von Matplotlib-Diagrammen in Tkinter.
- `SensorData`: Eine benutzerdefinierte Klasse (`SensordataList`), die Daten verwaltet und speichert.

2. Variablen und Initialisierung

- `options`: Eine Liste von möglichen Aktivitäten, die der Benutzer auswählen kann.
- `root`: Das Hauptfenster der GUI.
- `figure`: Ein Objekt, das das aktuell angezeigte Diagramm speichert.
- `listing`: Eine Instanz der `SensordataList`-Klasse, die die Sensordaten verwaltet.

3. Funktionen im Code

- `writeData(data)`: Diese Funktion schreibt die übergebenen Daten in die `SensordataList`-Instanz `listing`.
- `display()`: Diese Funktion richtet das Hauptfenster (`root`) ein, setzt den Titel und die Fenstergröße und wendet eine Hintergrundfarbe an.
- `beenden()`: Beendet das Programm, indem es das Fenster schließt und die `sys.exit()`-Methode aufruft.
- `on_select_activity(activity, frame_bottom, diagramFrame)`: Diese Funktion wird aufgerufen, wenn der Benutzer eine Aktivität aus dem Dropdown-Menü auswählt. Abhängig von der Auswahl wird die entsprechende Funktion zum Anzeigen des Diagramms aufgerufen.
- `show_chart(canvas_bottom)`: Diese Funktion zeigt das Diagramm in dem übergebenen Canvas-Widget an. Dabei wird das Diagramm aus der globalen `figure`-Variable abgerufen und angezeigt.
- `hide_chart(canvas_bottom)`: Diese Funktion entfernt alle Diagramme aus dem Canvas-Widget.
- `setup()`: Diese Funktion richtet die GUI ein, platziert den Kalender, das Dropdown-Menü und den Beenden-Knopf auf der linken Seite und den Diagramm-Frame auf der rechten Seite. Sie zeigt außerdem das Standarddiagramm für die Jumps-Aktivität an.
- `selecting(activity, frame_bottom, diagramFrame)`: Diese Funktion wird verwendet, um die entsprechende Diagramm-Anzeige basierend auf der Auswahl des Benutzers im Dropdown-Menü anzuzeigen. Sie ruft eine der spezifischen Anzeige-Funktionen für Aktivitäten wie "Jumps", "Walking Steps" usw. auf.

4. Aktivitätsdiagramm-Funktionen

- `display_jumps(window)`: Diese Funktion zeigt ein Balkendiagramm für die Anzahl der Sprünge (Jumps) an. Die Daten sind fiktiv und stellen die Anzahl der Sprünge an fünf Tagen dar. Die matplotlib-Bibliothek wird verwendet, um das Balkendiagramm zu erstellen, und das Diagramm wird im window-Frame angezeigt.
- `display_walking_steps(window, canvas_bottom)`: Diese Funktion zeigt ein Balkendiagramm für die Anzahl der Schritte an fünf Tagen an. Auch hier werden fiktive Daten verwendet. Zusätzlich werden zwei Schaltflächen zum Ein- und Ausblenden des Diagramms bereitgestellt.
- `display_squats(window, canvas_bottom)`: Dieses zeigt ein Balkendiagramm für die Anzahl der Squats an. Wie bei den "Walking Steps" gibt es auch hier Schaltflächen zum Ein- und Ausblenden des Diagramms.
- `display_situps(window, canvas_bottom)`: Diese Funktion zeigt ein Balkendiagramm für die Anzahl der Sit-ups an. Auch hier gibt es Schaltflächen für die Steuerung der Anzeige des Diagramms.
- `display_pushups(window, canvas_bottom)`: Diese Funktion zeigt ein Balkendiagramm für die Anzahl der Push-ups an. Wie in den anderen Diagrammen wird auch hier die Möglichkeit zum Ein- und Ausblenden des Diagramms bereitgestellt.
- `display_average_speed(window, canvas_bottom)`: Diese Funktion zeigt ein Balkendiagramm für die durchschnittliche Geschwindigkeit pro Tag an. Die Daten repräsentieren fiktive Durchschnittsgeschwindigkeiten an fünf Tagen. Wie bei den anderen Diagrammen können auch hier Schaltflächen verwendet werden, um das Diagramm anzuzeigen oder zu verbergen.

5. Benutzerinteraktion

- Dropdown-Menü: Der Benutzer kann im Dropdown-Menü eine der Aktivitäten auswählen (z. B. „Jumps“, „Walking Steps“, „Squats“, usw.). Sobald der Benutzer eine Auswahl trifft, wird die zugehörige Funktion aufgerufen, die das Diagramm für die ausgewählte Aktivität anzeigt.
- Schaltflächen: Für die Diagramme der Aktivitäten wie „Walking Steps“, „Squats“, „Situps“, „Pushups“ und „Average Speed“ gibt es jeweils zwei Schaltflächen: „Show“ und „Hide“. Mit der „Show“-Schaltfläche wird das Diagramm angezeigt, und mit der „Hide“-Schaltfläche wird es wieder entfernt.

6. Zusammenfassung der Funktionsweise

- Beim Starten der Anwendung wird die `setup()`-Funktion aufgerufen, die das Fenster mit einem Kalender, einem Dropdown-Menü und einem Beenden Knopf einrichtet.
- Das Dropdown-Menü enthält die verfügbaren Aktivitäten (Jumps, Walking Steps, Squats, Situps, Pushups, Average Speed). Der Benutzer wählt eine Aktivität aus, und das entsprechende Diagramm wird im rechten Fensterbereich angezeigt.
- Jedes Aktivitätsdiagramm wird als Balkendiagramm erstellt und mit Matplotlib dargestellt.
- Es gibt Schaltflächen zum Anzeigen und Verbergen von Diagrammen, um die Benutzerinteraktion zu erleichtern.
- Am Ende können alle Diagramme entfernt werden, und das Programm kann über den Beenden Knopf geschlossen werden.

Fazit

Im Rahmen dieses Projekts konnte wertvolle Erfahrung in der Programmierung einer vollständigen Bewegungserkennung gesammelt werden. Die Erweiterung der Firmware und die Umsetzung neuer Funktionen haben nicht nur das technische Verständnis vertieft, sondern auch praktische Anwendungsmöglichkeiten für zuvor erlernte Konzepte aus anderen Modulen aufgezeigt.

Für zukünftige Projekte ergeben sich mehrere spannende Ansätze. Einerseits könnte der bestehende, defekte Motion Tracker wieder lauffähig gemacht werden. Andererseits wäre es möglich, eigene Hardware bereitzustellen und eine vollständig maßgeschneiderte Firmware zu entwickeln. Dadurch könnte eine Bewegungserkennung implementiert werden, die alle sechs Achsen der IMU nutzt und zusätzlich auf Gyroskopdaten basiert.

Darüber hinaus könnten Machine-Learning Methoden zur Bewegungsanalyse eingesetzt werden, um Genauigkeit weiter zu verbessern. Das Projekt hat gezeigt, wie verschiedene technische Disziplinen ineinandergreifen und unterstreicht das Potenzial für weiterführende Entwicklungen in diesem Bereich.

Literatur

[1] *Anleitung BLE.pdf* - Verfügbar unter:

<https://nextcloud.frankfurt-university.de/s/7Ybstx76qTD7PBE>

[2] *Info Sensor.pdf* - Verfügbar unter:

<https://nextcloud.frankfurt-university.de/s/7Ybstx76qTD7PBE>

[3] *Datenblatt BMI323* - Verfügbar unter: [https:](https://www.mouser.de/datasheet/2/783/Bosch_9_8_2022_BST_BMI323_SF000_00-3049471.pdf)

[//www.mouser.de/datasheet/2/783/Bosch_9_8_2022_BST_BMI323_SF000_00-3049471.pdf](https://www.mouser.de/datasheet/2/783/Bosch_9_8_2022_BST_BMI323_SF000_00-3049471.pdf)

[4] *Programming How To.pdf* - Verfügbar unter:

<https://nextcloud.frankfurt-university.de/s/7Ybstx76qTD7PBE>

[5] *Mikrocontroller* - Verfügbar unter:

<https://www.st.com/en/microcontrollers-microprocessors/stm32wb55rg.html>