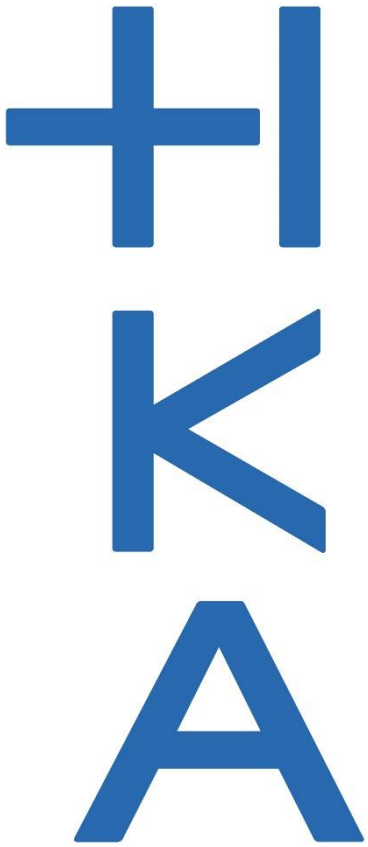
Hochschule Karlsruhe

EKG-Messgerät

EKG-Labor



Walter Orlov 67152
Gabriel Böhm 67832

Betreuer:
Prof. Tobias Baas

Inhaltsverzeichnis

Einleitung

Dieser Bericht dokumentiert die erfolgreiche praktische Umsetzung und Anwendung der in der Vorlesung Informationstechnik erworbenen theoretischen Kenntnisse. Gemeinsam haben wir ein EKG-Messgerät entwickelt, das nicht nur unser Verständnis für Informationstechnik stärkt, sondern auch als Übung und vertiefendes Erlernen von Programmierfähigkeiten dient.

Das EKG-Messgerät, auch Elektrokardiogramm genannt, ist ein medizinisches Instrument, das dazu verwendet wird, die elektrischen Aktivitäten des Herzens aufzuzeichnen. Dieses Instrument spielt eine entscheidende Rolle in der Medizin, da es Ärzten und medizinischem Personal ermöglicht, die Herzaktivität eines Patienten zu überwachen und mögliche Anomalien zu identifizieren. Die Relevanz eines EKG-Geräts liegt in seiner Fähigkeit, präzise und Echtzeitinformationen über den Zustand des Herzens zu liefern, was bei der Diagnose und Überwachung von Herzkrankheiten von entscheidender Bedeutung ist.

Im Rahmen unseres Projekts haben wir das EKG-Gerät in der Programmiersprache C++ auf dem **ESP32** implementiert, wobei **MATLAB** als unterstützendes Werkzeug zur Datenverarbeitung eingesetzt wurde. Die Echtzeit-EKG-Werte werden vom **ESP32** erfasst und verarbeitet. Dies umfasst auch die Anpassung und Programmierung von einem Filter, um genaue und qualitativ hochwertige Daten zu gewährleisten.

Diese praktische Erfahrung erlaubte es uns, nicht nur unsere Programmierkenntnisse zu vertiefen, sondern auch ein tieferes Verständnis für die Verbindung zwischen Informationstechnik und medizinischer Diagnostik zu entwickeln. Im Folgenden werden wir einige Programmabschnitte besprechen.

Aufgabenstellung

Die Aufgabenstellung umfasst die Schlüsselkomponenten, die implementiert werden müssen. Für detaillierte Informationen und weitere Teilaufgaben verweisen wir auf die Laborblätter.

Es ist die Aufgabe, ein EKG-Messgerät mit Hilfe des **ESP32** zu programmieren. Zur Implementierung des Filters wird **MATLAB** in Kombination mit dem **FilterDesigner**-Tool genutzt. Das Ziel besteht darin, den Puls alle vier Millisekunden präzise zu messen. Die erfassten Daten werden in einem Puffer mit ausreichender Kapazität für eine 30-sekündige Aufzeichnung gespeichert und danach zur weiteren Bearbeitung an **MATLAB** übertragen. In **MATLAB** erfolgt die Auslegung des Filters, dessen Implementierung dann im Code erfolgt, mit dem Ziel, das Rauschen von 50 Hz zu filtern. Das Endziel dieses Projekts ist die Entwicklung eines funktionsfähigen EKG-Messgeräts.

WLAN als Schnittstelle zu MATLAB

Im nachfolgenden Abschnitt erläutern wir die wesentlichen Aspekte zur Konfiguration des WLANs auf dem **ESP32** sowie die entsprechenden Einstellungen in **MATLAB**. Das WLAN fungiert dabei als Schnittstelle zwischen unserem **ESP32** und dem **MATLAB**-Programm. Durch diese Schnittstelle sind wir in der Lage, Daten sowohl vom **ESP32** an **MATLAB** zu übermitteln als auch umgekehrt.

Zu Beginn wird beschrieben, wie wir die Schnittstelle auf dem **ESP32** konfiguriert haben, gefolgt von einer Erläuterung der entsprechenden Einrichtung in **MATLAB**. Dieser Schritt

legt den Grundstein für die reibungslose Übertragung von Daten zwischen den beiden Plattformen und ermöglicht eine Kommunikation zwischen dem **ESP32** und dem **MATLAB**-Programm.

```
/* WiFi-Mode:  */
WiFi.mode(WIFI_STA);
WiFi.begin(myWifiSettings.wifiSSID, myWifiSettings.wifiPAS);

/* DoWhile-Schleife um 30sec versuchen die WiFi Verbindung aufzubauen */
do
{
  myDisplay.clear();

  if (indexDot == 4)
  {
    myWifiSettings.wifiCon = "Connecting to WiFi";
    indexDot = 0;
  }

  myDisplay.drawString(64, 32, myWifiSettings.wifiCon);
  myWifiSettings.wifiCon += ".";
  myDisplay.display();

  vTaskDelay(1000 / portTICK_PERIOD_MS); /* Delay von 1000ms */

  indexWait++;
  indexDot++;

  if (WiFi.status() == WL_CONNECTED)
  {
    break;
  }

} while (indexWait < 30);
```

Codesegment 1: Verbindungsaufbau

Im folgenden Codeausschnitt (Codesegment 1)für den **ESP32** ist ersichtlich, dass gemäß den Vorgaben der Aufgabenstellung der Verbindungsaufbau zum WLAN-Router für 30 Sekunden aktiv versucht wird, bevor die nachfolgenden Programmschritte ausgeführt werden. Zur Konfiguration der WIFI-Modi haben wir zuvor die erforderlichen Bibliotheken „Wifi.h“ und „WifiUdp.“ eingebunden. Auf dem Bildschirm wird der laufende Verbindungsvorgang angezeigt, und im Erfolgsfall wird die Meldung „Connected to WiFi“ ausgegeben. Der Status der Verbindung, ob erfolgreich oder nicht, wird durch eine Funktion aus der Bibliothek überwacht. Bei einer nicht erfolgreichen Verbindung erscheint eine entsprechende Fehlermeldung auf dem Bildschirm. Die Bedeutung der Fehlermeldungen ist in der nachfolgenden Tabelle (Tabelle 1) erläutert:

Tabelle 1: Fehlermeldungen bei nicht erfolgreicher Verbindung

Value	Constant	Meaning
0	`WL_IDLE_STATUS`	temporary status assigned when `WiFi.begin()` is called
1	`WL_NO_SSID_AVAIL`	when no SSID are available
2	`WL_SCAN_COMPLETED`	scan networks is completed
3	`WL_CONNECTED`	when connected to a WiFi network
4	`WL_CONNECT_FAILED`	when the connection fails for all the attempts
5	`WL_CONNECTION_LOST`	when the connection is lost
6	`WL_DISCONNECTED`	when disconnected from a network

Um die Verbindung zum **MATLAB**-Programm herzustellen, werden die IP-Adresse und der UDP-Port des **ESP32** benötigt. Der Port wurde zuvor mithilfe eines Makros auf 2020 festgelegt. Sowohl die IP-Adresse als auch der Port werden auf dem Bildschirm angezeigt, um bei der ersten Verbindung des **ESP32s** mit dem **MATLAB**-Programm die

Informationen leicht ändern bzw. anpassen zu können. Diese Anzeige ermöglicht eine einfache Konfiguration der Verbindungseinstellungen für eine reibungslose Kommunikation zwischen dem **ESP32** und dem **MATLAB**-Programm, insbesondere bei der erstmaligen Einrichtung.

Um eine sichere Schnittstelle über das WLAN einzurichten, wird im **MATLAB**-Code die Variable mit der korrekten IP-Adresse des **ESP32** sowie dem zugehörigen UDP-Port angepasst. Um die Verbindung zwischen dem **ESP32** und dem **MATLAB**-Programm abzuschließen, sind die folgenden Codeabschnitte im **MATLAB**-Programm maßgeblich:

```
4   ESPipAdresse = "192.168.177.89";
5   ESPudpPort = 123;
6   BUFFERSIZE = 7500;
7
8   testVar = 777;
9
10  % Initialisierung
11  uBroadcaster = udpport("datagram")
12  uBroadcaster.EnableBroadcast = true;
13
14  uReceiver = udpport("byte", "LocalPort", 2020, "EnablePortSharing", true)
15
16  write(uBroadcaster, testVar, "uint16", ESPipAdresse, ESPudpPort);
```

Codesegment 2: MATLAB – Verbindungsaufbau zum ESP32

Bei erfolgreichem Versand erreicht die Nachricht von **MATLAB** den **ESP32**-UDP-Port, wo sie nun gelsen werden kann. Sobald eine Nachricht eingegangen ist, wird folgendes auf dem Bildschirm angezeigt (siehe Abbildung 1).

Durch die erfolgreiche Übermittlung der Nachricht kann der **ESP32** auch die IP-Adresse des Senders sowie den zugehörigen UDP-Port sichern. Dadurch ist die Verbindung hergestellt, und die Daten können nun sicher übertragen und verarbeitet werden.

Abbildung 1: Erfolgreicher Dateneingang auf dem ESP32

Implementierung des Analogausgangs

Der **ESP32** verfügt über zwei Digital-Analog-Wandler (DACs). Dies erlaubt die Ausgabe einer variablen Spannung zwischen 0 und 3,3V mit einer Auflösung von 8 Bit. Wir haben einen dieser DAC-Ausgänge (Pin 26) zur Erprobung der Analogmessung verwendet.

Während der ersten Inbetriebnahme des **ESP32** haben wir eine Sinuskurve über den Ausgang ausgegeben. Um die Entwicklung der Messung und der Displayausgabe nicht zu beeinträchtigen, haben wir die Berechnung und das Schreiben des Werts in einen eigenen Task ausgelagert. Die Erstellung des Tasks war aufgrund der nativen Integration des Betriebssystems **FreeRTOS** auf dem **ESP32** keine Herausforderung. **FreeRTOS** ist ein Echtzeit-Betriebssystem für Mikrocontroller, das als Open-Source-Projekt kostenlos verwendet werden kann. Es ermöglicht eine einfache Verwendung von Threads und Prozessen.

Da die Ausgabe der Sinuskurve in einen separaten Task ausgelagert wurde, konnte die Implementierung vereinfacht werden. In diesem Teil des Programms ist eine hohe Performance und Echtzeitfähigkeit nicht notwendig, weshalb das Scheduling mittels unkomplizierter Delay-Befehle durchgeführt werden konnte.

Im Laufe des Projekts hat sich die Bedeutung des Analogausgangs drastisch erhöht, seit wir zu Simulationszwecken bereits vorhandene Messungen der Herzrate über den Analogausgang ausgegeben haben. Diese Maßnahme ermöglichte uns die effiziente und zuverlässige Weiterentwicklung der anderen Funktionen wie Messung, Filterung, Displayausgabe und Datentransmit, ohne eine tatsächliche Messung an der Person durchzuführen. Für den optimalen Test unseres digitalen Filters haben wir unsere eigene EKG-Aufnahme in ein Array geschrieben, dessen Werte nacheinander in gleichbleibender Frequenz über den DAC-Port ausgegeben werden.

Um den digitalen Filter optimal zu testen, ist es erforderlich, dass die Frequenz der Ausgabe höchstgenau ist. In diesem Fall führte die Auslagerung in den zusätzlichen Task nicht zu den gewünschten Ergebnissen. Es gab geringe Abweichungen der Frequenz, welche das Spektrum der Ausgabe leicht verzerrt haben. Unter diesen Bedingungen war es uns nicht möglich, die Funktionalität des Filters zu testen. Deshalb wurde die Idee der Taskerstellung verworfen und die Analogausgabe durch eigenes Scheduling in der Main-Task implementiert. Diese Maßnahme hat sich als erfolgreich erwiesen.

Implementierung der Messung

Die EKG-Messungen werden mittels des Analog-Digital-Wandlers (ADC) am Pin 33 durchgeführt. Der ADC verfügt über eine Auflösung von 12 Bit, wodurch beim Einlesen der Eingangsspannung ein Integer-Wert zwischen 0 und 4095 ausgegeben wird. Diese Werte repräsentieren proportional eine Spannung von 0 bis 3,3V. Die Anforderung bestand darin, Messungen mit einer Frequenz von 250Hz durchzuführen und sicherzustellen, dass der Zeitpunkt der Ausführung höchste Genauigkeit aufweist, damit die Messung aussagekräftig ist. Dabei sollen die Daten der letzten 30 Sekunden gespeichert werden.

Für die Verwaltung der Daten haben wir eine Klasse namens "readEKG" (siehe Codesegment 3) erstellt, die das Speichern, Filtern, Auswerten und Übertragen der Daten kombiniert. Die Messungen finden im Timer-Interrupt statt, der alle 4ms ausgelöst wird. Der gemessene ADC-Wert wird ohne Zwischenberechnungen direkt hinter die zuletzt geschriebene Position des Buffers geschrieben, der eine Kapazität von 7500 Werten hat. Wenn das Ende des Buffers erreicht ist, wird der Schreibindex

```
class readEKG
{
private:
    uint16_t readValue;
    uint16_t analogBuffer[BUFFERSIZE];
    uint16_t filteredBuffer[BUFFERSIZE];
    uint16_t mWriteIndex;
    uint16_t mFilterIndex;
    uint16_t mReadIndex;
    uint32_t &counter4ms;

    /* Filter */
    double mem0;
    double mem1;
    double mem2;

    /* Calculation of heart rate (variability) */
    void calculateHeartRate();
    bool peakDetected;
    uint16_t heartTiming[2];
    uint16_t heartPeriod[2];
    uint16_t variability;
    float heartRate;
    uint32_t lastTiming;

public:
    readEKG(uint32_t &counter);
    void measure();
    void filter();
    void getValue(uint16_t &pValue);
    uint16_t getReadIndex();
    uint16_t getFilterIndex();
    uint16_t getWriteIndex();
    float getHeartRate();
    uint16_t getHRV();
    void transmitFirst();
    void transmitFirstFiltered();
    void transmitSecond();
    void transmitSecondFiltered();
};
```

Codesegment 3: readEKG-Klasse

zurückgesetzt. Dadurch funktioniert der einfache Buffer wie ein Ringbuffer, dessen alte Werte mit neuen überschrieben werden.

Durch die Durchführung der Messung im Timer-Interrupt konnte eine höchst zufriedenstellende Genauigkeit des Zeitintervalls zwischen den einzelnen Messungen erreicht werden. Genauere Betrachtungen mit einem Oszilloskop ergaben eine Abweichung der Abrufe von ±2,5µs mit vereinzelten Ausschlägen unter 10µs (siehe Abbildung 2). Dies ergibt eine maximale Abweichung von 0,25%.

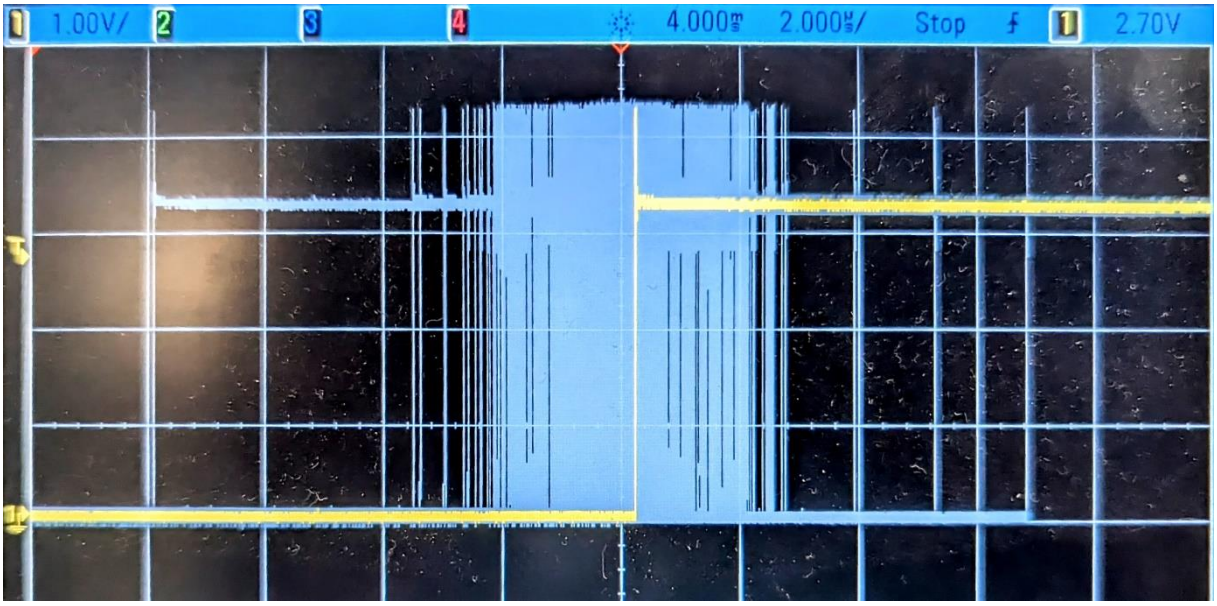


Abbildung 2: Messung des Jitters auf dem Oszilloskop

Display

Auf dem eingebauten Display sollen die Messwerte der EKG-Messung grafisch dargestellt werden. Die Implementierung erfolgt hier mit der Bibliothek „SSD1306Wire.h“, welche alle Funktionen enthält, um Texte und Geometrien auf dem Display auszugeben. Zur besseren Übersicht wurde eine weitere Klasse erstellt, die alle benötigten Attribute enthält.

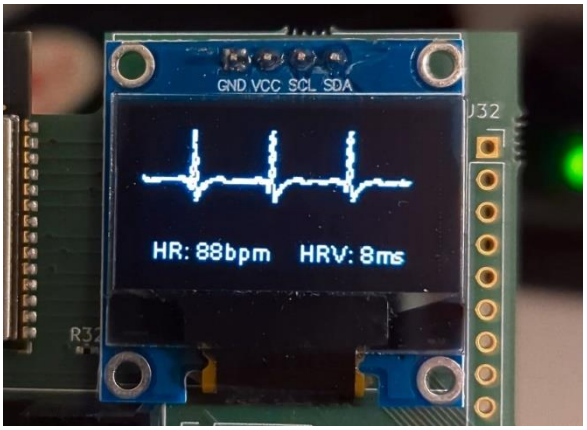


Abbildung 3: Ausgabe der Messung auf dem Display

Der Graph wird alle 32ms zyklisch gezeichnet, was dem menschlichen Auge flüssig erscheint. In jedem Zyklusdurchlauf werden alle neu beschriebenen Daten der EKG-Klasse eingelesen und die y-Position zur Darstellung berechnet. Hierbei wird der 12-Bit-Wert auf die Skala des Displays skaliert. Das Display besitzt eine Auflösung von 128x64 Pixeln, wodurch die y-Position zwischen 0 und 64 liegen muss. Da wir eine skalierbare x-Achse und somit ein variables Zeitfenster implementieren möchten, wird die x-Position des neuen Datums mit einer Gleitkommazahl inkrementiert, die vom gewünschten Zeitfenster abhängt. Sollte sich die aktuelle x-Position nicht bei 0 befinden, wird eine Linie vom letzten gezeichneten Wert zur neuen Position mithilfe der Funktion "drawLine" gezogen. Wenn die Position 128 erreicht, wird das gesamte Display zurückgesetzt.

Darüber hinaus erfolgt die Anzeige der Herzrate und der Herzratenvariabilität auf dem Display. Die Berechnung der Werte wird in einem späteren Kapitel genauer erläutert. Im Allgemeinen ist die Display-Bibliothek so konfiguriert, dass sie Pixel setzt und ausgibt, ohne die vorherigen Pixel zurückzusetzen. Dies ist bei der Texterstellung problematisch, da sich bei wiederholtem Schreiben neue Zeichen überlagern, für eine optimale Graphenausgabe. Um dies zu verhindern, ist es notwendig, die Pixel bei jedem Schreibzyklus im gesamten zu beschreibenden Bereich zurückzusetzen. Dies kann mit einer doppelten For-Schleife gelöst werden, bei der x und y inkrementiert werden.

ESP32 – Schreiben der Werte

Im folgenden Kapitel wird erläutert, wie die Messdaten vom **ESP32** über WLAN übertragen werden.

Die Aufgabenstellung sieht vor, dass die Werte der letzten 30 Sekunden an **MATLAB** übermittelt werden sollen. Um sicherzustellen, dass keine fehlerhaften Zugriffe auftreten, weil ein Datum an derselben Position zur gleichen Zeit geschrieben und gelesen werden möchte, erfolgt die Übertragung in zwei Abschnitten. In jedem Abschnitt werden jeweils 15 Sekunden an Daten verpackt und versendet. Damit dies ermöglicht wird, muss das Scheduling so gewählt werden, dass die Funktionsaufrufe zum Übertragen erst ausgeführt werden, sobald der Index des zuletzt geschriebenen Wertes die gewünschte Position erreicht hat. Um sicherzustellen, dass sich die Funktionsaufrufe abwechseln, wird ein "Handshake" einer booleschen Variablen "flagUDPSend" durchgeführt.

```
/* Transmit Data */
/* UDP send first half of buffer */
if (myEKG.getFilterIndex() >= BUFFERSIZE / 2 && flagUDPSend)
{
  Serial.printf("Send first!\t- %d\n", counter4ms * EKG_SAMPLING_TIME_MS);
  myEKG.transmitFirst();
  myEKG.transmitFirstFiltered();
  flagUDPSend = false;
}

/* UDP send second half of buffer */
if (myEKG.getFilterIndex() < BUFFERSIZE / 2 && !flagUDPSend)
{
  Serial.printf("Send second!\t- %d\n", counter4ms * EKG_SAMPLING_TIME_MS);
  myEKG.transmitSecond();
  myEKG.transmitSecondFiltered();
  flagUDPSend = true;
}
```

Codesegment 4: Implementierung des Schedulings für den Datentransmit

Beim Aufruf der Funktion wird ein Paket zum Senden der Daten mit dem entsprechenden Remote-Port geöffnet. Es ist zu beachten, dass die ungefilterten und gefilterten Daten an zwei verschiedene Ports gesendet werden. Dies erleichtert die Zuweisung der Daten in **MATLAB**. Zudem können in einer UDP-Nachricht lediglich 8 Bit übertragen werden. Daher muss ein 8-Bit-Pointer mittels Typecasting für die gewünschte Position des Buffers deklariert und die Position innerhalb einer Schleife inkrementiert werden, bis sämtliche Daten verschickt wurden. Bei diesem Prozess ist uns ein Fehler unterlaufen, den wir erst beim Betrachten der empfangenen Daten in **MATLAB** entdeckt haben. Uns ist aufgefallen, dass das übertragene Sägezahnsignal genau nach 15 Sekunden einen kleinen Sprung aufweist. Zunächst vermuteten wir, dass der Fehler in der Pointerzuweisung lag, konnten diese Vermutung jedoch nicht bestätigen. Weitere Überlegungen ergaben, dass die Wiederholungen der Schleife nicht ausreichend waren. Die Anzahl der Schleifendurchläufe betrug die Hälfte der Buffergröße von 7500. Hierbei wurde jedoch nicht berücksichtigt, dass durch das Typecasting doppelt so viele Schleifendurchläufe notwendig sind, um alle 16-Bit Werte zu übertragen. Nachdem dieser Fehler behoben wurde, verlief die Übertragung problemlos.

MATLAB – Lesen und schreiben

Im folgenden Kapitel wird die Implementierung der Möglichkeit zur Kommunikation über **MATLAB** diskutiert.

Im Abschnitt "WLAN als Schnittstelle zu **MATLAB**" wird hervorgehoben, dass eine Mitteilung von **MATLAB** an den **ESP32** einmalig während der Konfiguration ausgeführt wird. Dieser Vorgang ist auf die erforderliche lokale IP-Adresse und den UDP-Port des lokalen Laptops zurückzuführen, um eine reibungslose Datenübertragung vom **ESP32** zum **MATLAB** zu gewährleisten. Im Codesegment 2 wird ein Objekt vom Typ "udpport" erstellt, um die Datenübertragung zu konfigurieren. Durch die Verwendung der Funktion "EnableBroadcast" wird das Objekt aktiviert, und es ist nun möglich, Bytes oder Datagramm-Typen zu senden. Mittels der Funktion "write()" wird das UDP-Paket anschließend an den **ESP32** gesendet.

Um jedoch auch Nachrichten vom **ESP32** an **MATLAB** zu empfangen, ist es erforderlich, ein weiteres Empfangspaket zu erstellen, das in unserem Programm als "uReceiver" bezeichnet wird. Dieses Objekt kann Bytes-Typen empfangen, und verschiedene Eigenschaften des Objekts werden konfiguriert. Dazu gehören insbesondere die IP-Adresse des **ESP32** sowie der UDP-Port.

```
23  while true
24      uReceiverCount = uReceiver.NumBytesAvailable;
25      if  uReceiverCount > 1
26          data(n,:) = read(uReceiver, 1, "uint16");
27          n = n+1;
28          plot(data)
29      end
30  end
```

Codesegment 5: Einlesen der Nachrichten

Um die tatsächlich empfangenen Daten verarbeiten zu können, erfolgt dies in einer while-Schleife, in der die Nachrichten kontinuierlich gelesen werden. Innerhalb diesen Scopes werden die erforderlichen Operationen durchgeführt, um die eingegangenen Daten zu verarbeiten.

Zunächst wird überprüft, ob eine neue Nachricht vorliegt, indem die Funktion **NumBytesAvailable** verwendet wird, um die Anzahl der verfügbaren Bytes zu ermitteln (diese Funktion ist schreibgeschützt). Anschließend werden die empfangenen Daten in einem Array gespeichert. Der Array wird entsprechend der verfügbaren Daten mit Informationen gefüllt.

Hierbei ist zu beachten, dass die Funktion **NumBytesAvailable** uint16-Werte liest, während der **ESP32** nur uint8-Werte schreibt. Es ist wichtig, die Struktur auf der **ESP32**-Seite zu berücksichtigen, insbesondere wie die Nachricht in einem geeigneten Buffer für die Übertragung vorbereitet wird.

Der Array, der zur Speicherung der Daten verwendet wird, hat eine Dimension von 7500 Werten. Diese Anzahl an Daten kann durch folgende Berechnung ermittelt werden:

$$Bufferzeit[s]: 30s$$

$$Auflösung[Hz]: 250\ Hz$$

Alle vier Millisekunden wird ein Wert gelesen

$$Werte = Bufferzeit * Auflösung = 30s * 250Hz = 7500$$

Die Details zum Schreiben und Versenden vom **ESP32** über WLAN an **MATLAB** sind im Kapitel "**ESP32** – Schreiben der Werte" beschrieben.

Realisierung des Filters auf dem ESP32

Um den Code und den eigentlichen Filter in Echtzeit auf dem **ESP32** zu implementieren und zu nutzen, wurde der Filter zuvor in **MATLAB** dimensioniert. Hierbei wurden die aufgezeichneten Daten in einem Spannungs-Zeit-Diagramm dargestellt. Es wurde festgestellt, dass die Frequenz von 50 Hz im aufgezeichneten Signal kaum erkennbar war, was darauf hindeutet, dass die Werte kaum verrauscht waren. Diese Beobachtung ist äußerst positiv und stellt eine gute Einschätzung dar. Aus pädagogischen Gründen und um des Lerneffekts willen haben wir unsere Daten etwas bearbeitet. Wir haben ein künstliches Rauschen hinzugefügt, genauer gesagt ein Rauschen von 50 Hz, das in der blauen Eingangskurve auf Abbildung 4 zu sehen ist.

```
/* Filter Numerator */
const double N0 = 1.0;
const double N1 = -0.6229461048516327092983146940241567790 51;
const double N2 = 1.0;

/* FIlter Denominator */
const double D0 = 1.0;
const double D1 = -0.5530763174366040146878731320612132549 29;
const double D2 = 0.7756795110496130796207125968066975474 36;

/* Filter gain */
const double gain = 0.8878397555248065398103562984033487737 18;
```

Codesegment 6: Koeffizienten des Filters

Um die Auswirkungen des Rauschens besser zu verstehen, haben wir in Anlehnung an den Unterricht und die **MATLAB**-Laborübungen eine Fouriertransformation durchgeführt. Dadurch transformierten wir das Signal aus dem Zeitbereich in den Frequenzbereich und analysierten die enthaltenen Frequenzen. In der blauen Kurve ist deutlich zu erkennen, dass wir einen starken Peak bei 50 Hz haben, der auf das zugefügte Rauschen zurückzuführen ist.

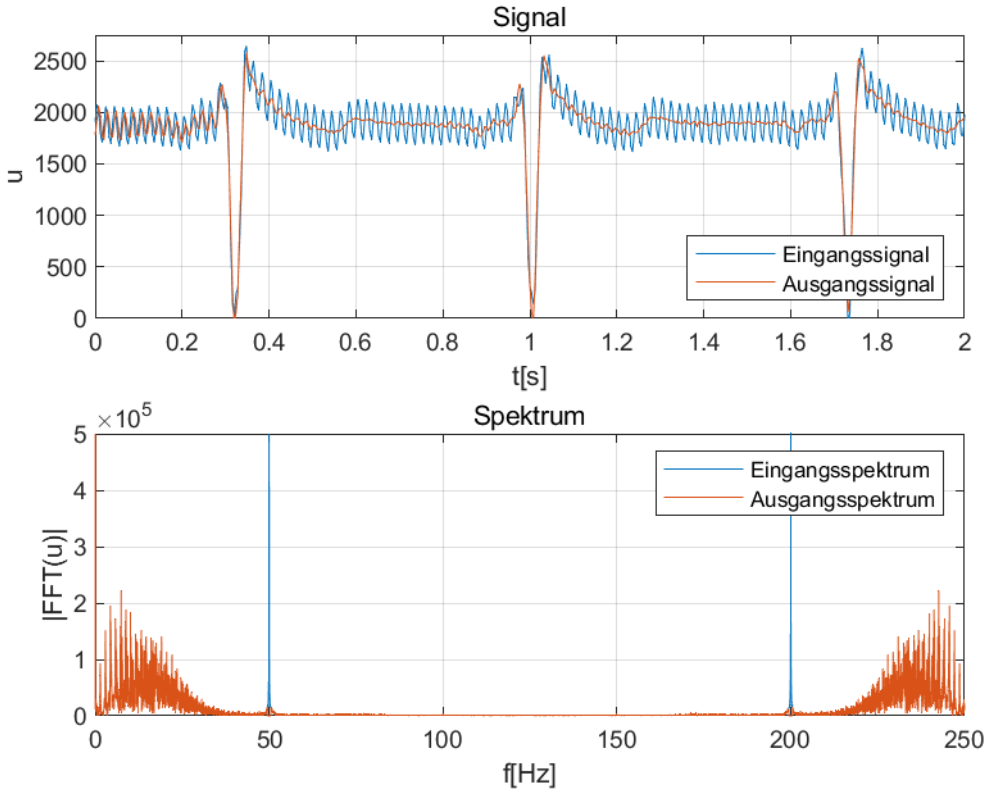


Abbildung 4: Signalverlauf und Spektrum mit hinzugefügtem Rauschen

Nach Analyse des Eingangssignals wird der Filter nun mithilfe des FilterDesigners in **MATLAB** dimensioniert. Am Ende erhalten wir Koeffizienten, mit denen wir die Daten in Echtzeit auf dem **ESP32** filtern und das Rauschen von 50Hz eliminieren können. Beim Einstellen des Filters haben wir durch den Bandstop eine bessere, genauere und stärkere Filterung der Frequenz bei 50Hz erreicht. Man könnte auch eine Lowpass-Einstellung wählen, wodurch das Ziel jedoch schlechter erreichbar wäre und das Rauschen stark verbleiben würde. Außerdem werden die Frequenzen zwischen 45 Hz und 55 Hz mit dem Filter gedämpft. Zudem muss eine Abtastfrequenz von 250Hz angegeben werden, um die Koeffizienten des Filters zu erhalten. Diese können dann in der Logik auf dem **ESP32** implementiert werden. Es ist wichtig, alle Nachkommastellen zu berücksichtigen, da unser Filter in C++ genauer funktioniert, je mehr Nachkommastellen wir nutzen. Der genauere Grund für die Verwendung von mehr Nachkommastellen wird am Ende des Kapitels beschrieben. Das Ergebnis des Filters ist als rote Ausgangskennlinie in der Abbildung 4 zu sehen.

Im folgenden Codeausschnitt (Codesegment 7) wird der Filter auf dem **ESP32** implementiert und die Koeffizienten des Filters angewendet, um die aufgezeichneten Daten zu filtern. Zu beachten ist, dass der Filter aufgrund seines Einschwingverhaltens erst nach dem zweiten Wert beginnt, den Wert zu glätten. Die nachfolgenden Werte werden dann

```
double y = 0.0;

mem0 = analogBuffer[mFilterIndex] - D1 * mem1 - D2 * mem2;
y = gain * (N0 * mem0 + N1 * mem1 + N2 * mem2);

mem2 = mem1;
mem1 = mem0;
```

Codesegment 7: Filterung der Daten

sukzessive gefiltert bzw. geglättet.

Um die Dimensionierung des Filters besser zu verstehen, wird im folgenden Abschnitt die Vorgehensweise der Berechneten Koeffizienten beschrieben. Dabei konzentrieren wir uns auf die Theorie aus dem Unterricht. Die klare Struktur, wie der FilterDesigner von **MATLAB** dies umsetzt wird nicht beschrieben. Es folgt nur das Verständnis von einer Analyse eines diskreten Signals im Bildbereich.

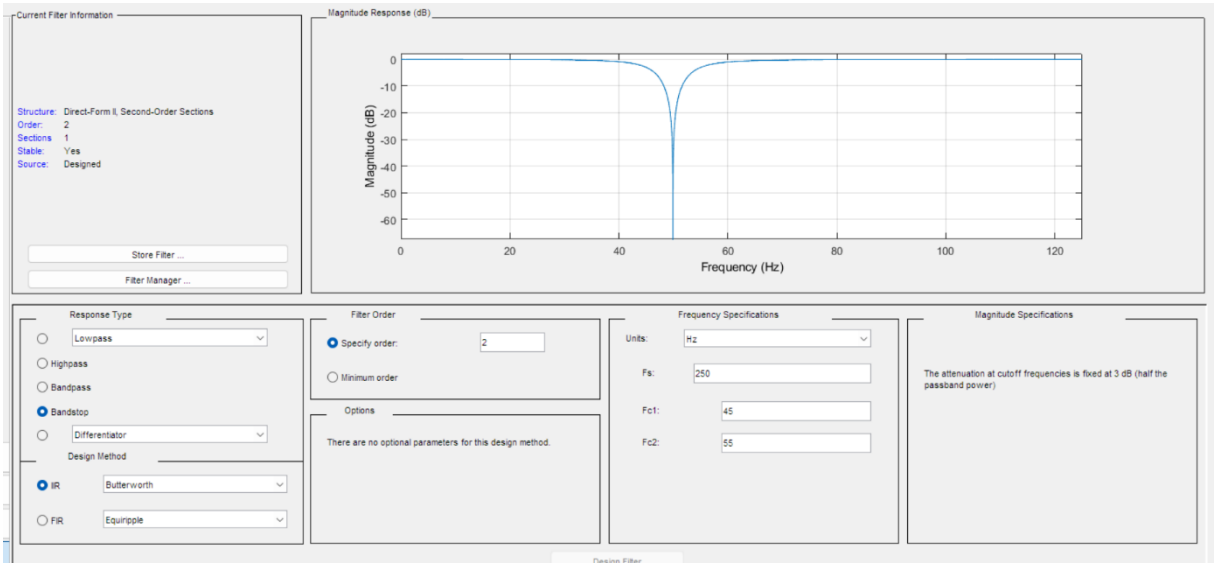


Abbildung 5: Der Filterdesigner von MATLAB

Da wir in unserem Fall alle vier Millisekunden einen Wert abtasten, erhalten wir eine zeitdiskrete Anwendung bzw. ein zeitdiskretes Signal. Mit Hilfe einer z-Transformation vom Zeitbereich in den Bildbereich kann das zeitdiskrete System beschrieben werden. Diese Transformation ist äquivalent zur Laplace-Transformation und erfolgt durch Substitution von z, wobei z eine komplexe Zahl ist. Dadurch wird das abgetastete kausale Signal in den Bildbereich transformiert und die Interpretation der Transformation erfolgt auf der komplexen Ebene. Die z-Transformation erlaubt eine Beurteilung der Stabilität anhand der berechneten Pole und Nullstellen. Die erhaltene Übertragungsfunktion kann weiter faktorisiert werden. Diese Faktoren können nun angepasst werden, um den gewünschten Filter zu setzen. Der Abstand vom Einheitskreis gibt die Stärke der Extremwerte an.

Generell kann gesagt werden, dass dieser Prozess auch in der Anwendung FilterDesigner durchlaufen wird. Auf diese Weise werden die Koeffizienten schnell und effizient berechnet und stehen für die Implementierung zur Verfügung.

Herz

Herzrate – Erklärt und implementiert

Die Herzrate, auch als Pulsfrequenz bekannt, gibt die Anzahl der Herzschläge pro Minute an und ist ein essenzieller Indikator für die Aktivität des Herz-Kreislauf-Systems. Der menschliche Puls ist nicht konstant, sondern variiert je nach körperlicher Aktivität, emotionaler Zustände und anderen Einflussfaktoren. Während Ruhephasen oder Schlaf kann die Herzrate langsamer sein, während sie bei körperlicher Anstrengung oder emotionaler Erregung zunimmt. Die **Herzrate** wird oft als Maß für die Belastung des Kreislaufsystems verwendet. Ein erhöhter Puls kann auf eine gesteigerte Anstrengung

oder emotionale Aufregung hinweisen, während ein niedriger Puls in Ruhephasen als normal betrachtet wird.

Es ist wichtig zu beachten, dass die Herzrate individuell variieren kann und von verschiedenen Faktoren beeinflusst wird, darunter das Alter, die Fitness und der Gesundheitszustand einer Person. Die Messung der Herzrate ist eine einfache, aber aussagekräftige Methode, um den allgemeinen Zustand des Herz-Kreislauf-Systems zu überwachen.

Im weiteren Verlauf wird die Echtzeit-Berechnung des Herzrate beschrieben.

Gängige Praktiken, wie das Ausführen eines Interrupts nach Detektion einer Flanke, sind in diesem Fall nicht geeignet, um die Zeit zwischen zwei Peaks bei einem analogen Signal zu messen. Stattdessen haben wir kontinuierlich die gefilterten Werte abgerufen und mit einem Threshold von 800 verglichen. Es werden drei Bedingungen abgefragt. Erstens, ob der Wert unter dem Schwellenwert liegt; zweitens, ob er kleiner als der vorherige Wert ist; und drittens, ob bereits zuvor beide Bedingungen erfüllt wurden. Letzte Bedingung wird mithilfe eines Flags gesteuert, das mit einer steigenden Flanke zurückgesetzt wird. Wenn alle drei Bedingungen erfüllt sind, wird der aktuelle Zeitpunkt gespeichert und die Differenz zum zuletzt erfassten Peak berechnet.

Der beschriebene Algorithmus wurde zunächst in **MATLAB** mit Hilfe einer gespeicherten Messung getestet und optimiert. Dabei hat sich der Schwellenwert von 800 bewährt. In Abbildung 6 lassen sich die Zeitpunkte erkennen, zu welchen ein Peak detektiert wurde. Allerdings gibt es das Problem, dass unsaubere Daten auftreten können. In den Daten, welche zur Simulation herangezogen wurden, trat eine minimale Abweichung mit einem kurzen Sprung nach oben bei einer R-Zacke auf. Dies wurde vom Algorithmus als steigende Flanke erkannt, wodurch unmittelbar eine zweite Spitze fälschlicherweise detektiert wurde. Um ähnliche Vorfälle künftig zu verhindern, werden für die nächsten 200 Millisekunden keine weiteren Spitzen mehr gemessen. Dadurch konnte das Problem behoben werden.

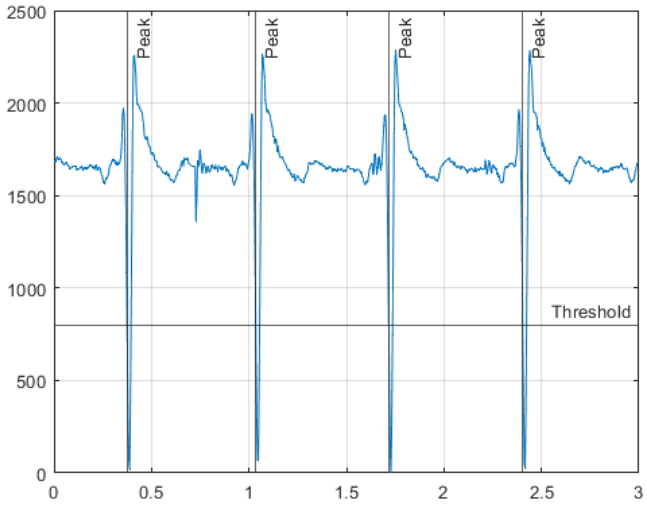


Abbildung 6: Darstellung der berechneten Peaks in MATLAB

Der Algorithmus wurde auf dem **ESP32** implementiert. Anhand der gemessenen Differenzen lässt sich die Herzrate in bpm mit einfachen Umformungen berechnen:

$$heartrate = \frac{60 * 1000}{\Delta heartTiming}$$

Herzratenvariabilität – Erklärt und implementiert

Die Herzratenvariabilität (HRV) misst den zeitlichen Abstand zwischen zwei aufeinanderfolgenden Herzschlägen und gibt die Variation dieses Zeitabstands in Millisekunden an. Das menschliche Herz schlägt nicht in einem konstanten Rhythmus,

sondern variiert zwischen langsameren und schnelleren Schlägen. Diese Variabilität hängt stark von der aktuellen Körperposition und den ausgeführten Aktivitäten ab. Im vorherigen Abschnitt haben wir die Herzrate besprochen, die die Belastung des Kreislaufsystems beschreibt, also wie stark der Kreislauf beansprucht wird. Im Gegensatz dazu beschäftigt sich die Herzratenvariabilität damit, wie das Herz-Kreislauf-System auf diese Belastung reagiert. Ähnlich wie die Herzrate unterliegt auch die Herzratenvariabilität inneren und äußeren Einflussfaktoren. Ein Beispiel für einen inneren Einflussfaktor sind Hormone. Aufgrund diverser Einflussfaktoren kann kein exakter Normwert festgelegt werden. Kriterien wie Alter, Geschlecht und körperliche Verfassung können den Messwert beeinflussen.

Für die Berechnung des HRV-Wertes müssen die letzten beiden gemessenen RR-Differenzen gespeichert werden. Der Betrag der Differenz zwischen diesen beiden Werten entspricht dem HRV-Wert. In Codesegment 8 ist die Implementierung zur Berechnung der Herzfrequenz und der Herzfrequenzvariabilität dargestellt.

```
void readEKG::calculateHeartRate()
{
    if (!peakDetected)
    {
        if ((counter4ms - heartTiming[0]) > 200 / EKG_SAMPLING_TIME_MS)
        {
            heartTiming[1] = counter4ms;
            heartPeriod[1] = (heartTiming[1] - heartTiming[0]) * EKG_SAMPLING_TIME_MS;
            heartRate = 60000.0 / float(heartPeriod[1]);
            variability = abs(heartPeriod[1] - heartPeriod[0]);

            heartPeriod[0] = heartPeriod[1];
            heartTiming[0] = heartTiming[1];
            Serial.printf("HR: %d\n", heartRate);
        }
    }
}
```

Codesegment 8: Berechnung der Herzrate und der Herzratenvariabilität

Ergebnis

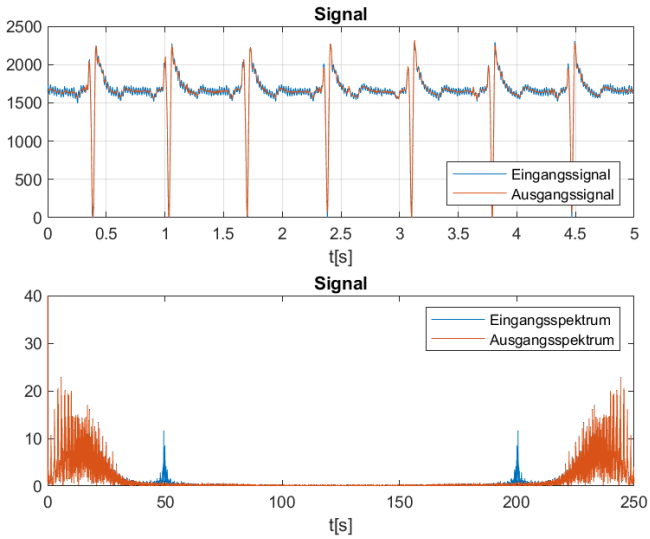
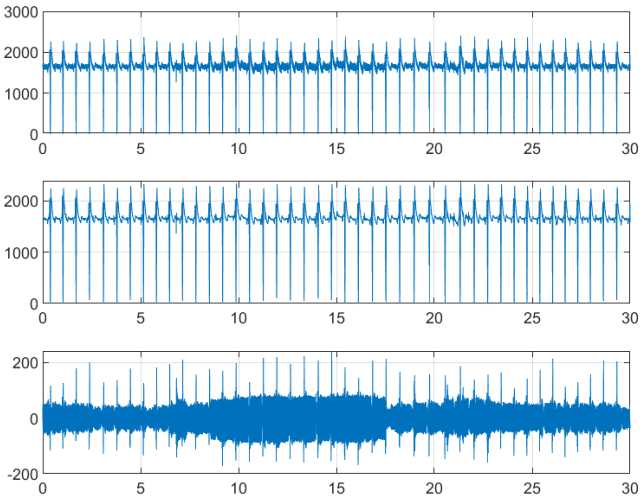


Abbildung 7: Finale Messung und Filterung

Folgende Abbildung (Abbildung 7) zeigt die Messungen auf dem **ESP32** inklusive der erfolgreichen Filterung bei der Frequenz von 50Hz. Die nächste Abbildung (Abbildung 8) stellt das ungefilterte und gefilterte Signal getrennt dar. In der dritten Zeile ist die Differenz beider Signale dargestellt. Es ist zu erkennen, dass ein kontinuierliches Rauschen eliminiert werden



konnte, jedoch gibt es Ausschläge bei den R-Zacken, welche aber keinen Einfluss auf das Ergebnis haben.

Abbildung 8: Differenz des gefilterten und ungefilterten Signals

Abbildungsverzeichnis

Abbildungen

Codesegmente

Tabellen

Literaturverzeichnis

- *ESP32 Useful Wi-Fi Library Functions (Arduino IDE)*. (o. J.). Random Nerd Tutorials. Abgerufen 14. November 2023, von https://randomnerdtutorials.com/esp32-useful-wi-fi-functions-arduino/
- Franke-Gricksch, N. (2017, Oktober 28). Berechnung des HRV-Werts RMSSD. Herzratenvariabilität. https://xn--hrv-herzratenvariabilitt-dcc.de/2017/10/berechnung-des-hrv-werts-rmssd/
- Skripte und Aufgaben der Veranstaltung Informationstechnik bei Prof. Dr.-Ing. Baas