Hochschule für Angewandte Wissenschaften Hamburg, Fakultät Life Science

Bachelorarbeit

FreeRTOS Implementierung auf einem SAMD21 zur Steuerung eines Fingermodels mittels EMG-Signal

Vorgelegt von

Bhima Benjamin Nohl

[redacted]

[redacted]

Studiengang: Medizintechnik B.Sc.

Erstprüferin: Prof. Dr. Petra Margaritoff

Zweitprüfer: Dr. Jan Dieckhoff

Hamburg, im Januar 2025

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst habe. Die von mir verwendeten Quellen und Hilfsmittel sind angegeben.

Wörtlich und inhaltlich entnommene Stellen wurden als solche kenntlich gemacht.

Datum: 14.01.2025 Unterschrift:

Danksagung

Ich bedanke mich bei Frau Prof. Dr. Petra Margaritoff und Herrn Dr. Jan Dieckhoff für die Bereitstellung des Themas, sowie für die gute Betreuung und das wertvolle Feedback während der Verfassung dieser Abschlussarbeit.

Besonderer Dank gilt Sneha Dhople.

Abstract

Für die Entwicklung echtzeitfähiger Systeme kann die Open-Source-Software FreeRTOS verwendet werden. Sie ermöglicht die Umsetzung von zeitbasierten Anforderungen in eingebetteten Systemen. Im Rahmen dieser Arbeit wurde das Brick'N'Knowledge-Bio-Feedback-Set genutzt, welches ein Arduino MKR WiFi 1010 Board verwendet. Das Set enthält elektrische Baukomponenten zur Aufnahme und Verarbeitung von Biosignalen. Das Ziel der Arbeit war die Implementierung von FreeRTOS auf dem SAMD21, sowie die Entwicklung eines Projekts, bei dem ein Fingermodel durch eine EMG-Messung gesteuert wird. Dabei wurden die Herausforderungen bei der Ressourcenverwaltung, der Synchronisierung von Tasks und der Einhaltung von Echtzeitanforderungen untersucht und analysiert.

Abkürzungsverzeichnis

CPU	*Central Processing Unit*
PWM	Pulsweitenmodulation
SPI	*Serial Peripheral Interface*
CS	*Chip Select*
SS	*Slave Select*
MISO	*Master-Input/Slave-Input*
MOSI	*Master-Output/Slave-Input*
SCLK	*Serial Clock*
CPOL	*Clock Polarity*
CPHA	*Clock Phase*
API	*Application Programming Interface*
FIFO	*First-In, First-Out*
ISR	*Interrupt Service Routine*
EMG	Elektromyografie
sEMG	*Surface-Electromyogram*
DC	*Direct Current*
CMR	*Common-Mode-Rejection*
CMRR	*Common-Mode-Rejection-Rate*
Ag/AgCl	Silber-Silberchlorid
RTOS	*Real Time OperatingSystem*
EKG	Elektrokardiogramm
EEG	Elektroenzephalogramm
AD-Wandler	Analog-Digital-Wandler
SPS	Samples pro Sekunde

Inhaltsverzeichnis

1 Einleitung

1.1 Zielsetzung und Motivation

Das Ziel dieser Arbeit ist die Implementierung von FreeRTOS auf dem MKR WiFi 1010 in Verbindung mit dem Brick'N'Knowledge-Biofeedback-Set. Es soll ein echtzeitfähiges System aufgebaut werden, das einen Fingermodel mittels EMG-Messung steuert. Es soll eine Grundlage ermöglicht werden, um mit FreeRTOS neue Praktikumsversuche für das Mikroprozessortechnik-Praktikum des Elektroniklabors der HAW Hamburg zu entwickeln. Darüber hinaus bietet sie Ansätze zur Erweiterung des Systems, um zukünftige Projekte wie die Steuerung komplexerer Robotikanwendungen oder die Integration weiterer Biosignale umsetzen zu können.

1.2 Theoretische Grundlagen

In diesem Kapitel wird ein Einblick in Mikrocontroller und deren Kommunikation mit peripheren Geräten gegeben. Da EMG-Messungen eine zentrale Rolle in diesem Projekt spielen, wird der technische und physiologische Hintergrund hinter einer EMG-Messung erläutert, um die Funktionsweise und die Anforderungen an die Signalverarbeitung zu verdeutlichen. Zudem wird die Funktionsweise von FreeRTOS beleuchtet, um Aufschluss über seine Funktionalitäten und Potenziale für die Entwicklung echtzeitfähiger Anwendungen zu geben.

1.2.1 Mikrocontroller

Arduino-Boards mit integrierten Mikrocontrollern bieten eine Plattform für die Entwicklung kompakter Systeme. Ein Arduino-Board besteht aus mehreren Komponenten, die für die technische Spezifikation relevant sind. In Abbildung 1 ist ein Arduino UNO dargestellt. Der Mikrocontroller bildet das Herzstück des Boards. Er ist ein kompakter Integrierter Schaltkreis, der die zentrale Verarbeitungseinheit (CPU) sowie häufig den Arbeits- und Flash-Speicher enthält. Der Mikrocontroller verarbeitet Eingaben basierend auf dem gespeicherten Programm [1].

Ein Quarzoszillator wird verwendet, um dem Prozessor ein Taktsignal bereitzustellen, das die Rate bestimmt, mit der Befehle verarbeitet werden [1]. Viele Arduino-Boards besitzen eine USB-Schnittstelle, über die Code in den Speicher hochgeladen werden kann. Diese Schnittstelle ermöglicht auch Konsolenausgaben, die unter anderem das Debugging des Codes erleichtern. Ein USB-Interface-Chip dient hierbei als Kommunikationsbrücke zwischen

USB-Anschluss und dem Mikrocontroller. Über die digitalen und analogen Pins kann der Mikrocontroller mit externen Systemen kommunizieren. Die analogen Pins können nur als Eingänge genutzt werden, während die digitalen Pins sowohl als Eingänge als auch als Ausgänge verwendet werden können. Die Digitalpins, die mit einer Tilde (~) markiert sind, unterstützen Pulsweitenmodulation (PWM) [2].

Für den Betrieb benötigt das Board eine Spannungsversorgen, diese liegt beim Arduino Uno beispielsweise im Bereich von 6 V bis 20 V liegt [2]. Die Versorgungsspannung kann auf verschiedenen Wegen bereitgestellt werden. Das Pin-Interface besitzt in der Regel einen Pin, um das Board mit Spannung zu versorgen, dies kann mit einen Funktionsgenerator oder einer Batterie erfolgen. Das Board kann auch über den USB-Anschluss mit Spannung versorgt werden. Manche Boards besitzen zusätzlich einen separaten Stromanschluss.

Ein *Reset*-Knopf, der häufig auf dem Board installiert ist, ermöglicht das Zurücksetzen des Mikrocontrollers, indem ein Signal an dessen *Reset*-Pin gesendet wird, was einen Neustart des Programms auslöst. Zusätzlich sind häufig Kontroll-LEDs integriert, die blinken, wenn der Mikrocontroller Daten empfängt (RX) oder sendet (TX) [2]. Darüber hinaus ist auf vielen Boards eine programmierbare LED integriert, die mit einem der Digitalen Pins verbunden ist.

Abbildung 1: Arduino UNO Board Layout [3].

Arduino-Boards besitzen Eingänge und Ausgänge, die genutzt werden können, um eingehende Informationen in Ausgaben zu transformieren. Die Boards sind in verschiedene Familien unterteilt, deren Mikrocontroller auf unterschiedlichen Architekturen basieren. Diese unterscheiden sich unter anderem in ihrer Befehlssatzarchitektur, Speicherorganisation und verfügbaren Peripherieschnittstellen [4].

1.2.2 SPI-Interface

Das *Serial Peripheral Interface* (SPI) ist ein häufig verwendetes Protokoll für die Kommunikation zwischen Peripheriegeräten und Mikrocontrollern. Die Kommunikation basiert auf einem *Master-Slave*-System. Der *Master* ist das steuernde Gerät, das dem *Slave* Anweisungen gibt. Für die Kommunikation zwischen einem *Master* und einem Peripheriegerät sind vier Anschlüsse notwendig. Der *Chip Select* (CS), oder auch *Slave Select* (SS) genannt, schaltet die Leitung für den *Slave* frei, mit dem kommuniziert werden soll. Der *Master-Input/Slave-Output* (MISO) ist die Datenübertragungsleitung vom *Slave* zum *Master*. Umgekehrt ist der *Master-Output/Slave-Input* (MOSI), die Leitung mit der der *Master* Daten an den *Slave* überträgt. Über die Taktleitung (*Serial Clock*, SCLK) sendet der *Master* das Taktsignal, das die Datenübertragung zwischen den Geräten synchronisiert. Es existieren vier verschiedene SPI-Modi, mit dem die Polarität des Taktsignals im Leerlauf, sowie die Flanke, bei der die die Daten abgetastet werden, definiert werden können [5].

In Abbildung 2 ist eine Datenübertragung dargestellt, bei der die *Clock Polarity* (CPOL) und *Clock Phase* (CPHA) auf 0 gesetzt sind. Dies bedeutet, dass die Taktpolarität im Leerlauf auf einer logischen Null liegt. Die Daten werden bei steigender Flanke abgetastet und bei fallender Flanke wird das Bitregister verschoben. Der CS wird in der Regel vor der Übertragung eines Datenpaketes von *HIGH* auf *LOW* gesetzt und nach Abschluss der Übertragung wieder von *LOW* auf *HIGH* gesetzt [5].

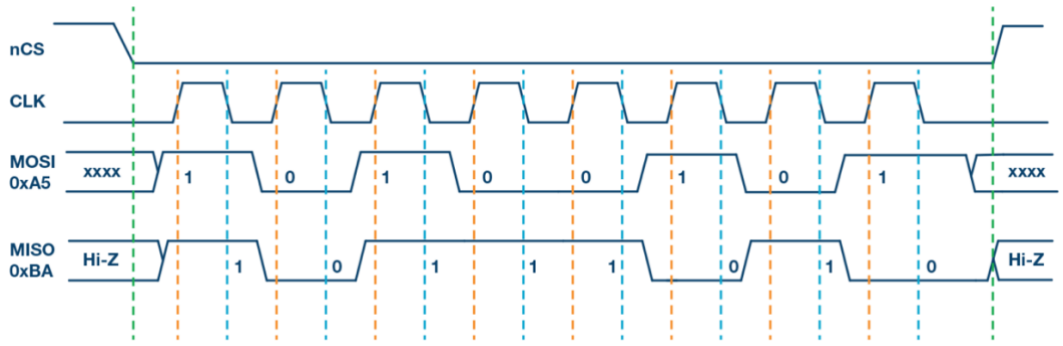

Abbildung 2: SPI-Datenübertragung im Modus CPOL = 0, CPHA =0 [5].

Wie in Abbildung 3 in dargestellt, können mehrere Peripherie-Geräte an dieselben SCLK-, MOSI- und MISO-Leitungen angeschlossen werden. Um die unterschiedlichen Geräte selektieren zu können, benötigt jeder *Slave* seine eigene CS-Leitung, die entsprechend auf *LOW* gesetzt wird, wenn das jeweilige Gerät angesprochen werden soll [5].

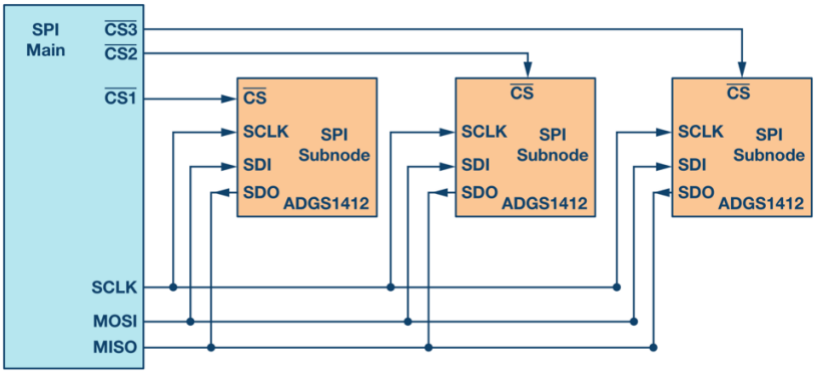

Abbildung 3: SPI-Konfiguration mit mehreren Geräten [5].

1.2.3 FreeRTOS

FreeRTOS wurde speziell für den Einsatz in eingebetteten Systemen entwickelt. Es handelt sich um ein Open-Source-Echtzeitbetriebssystem, das eine einfache Verwaltung von *Tasks* ermöglicht. Mit einer minimalen Speicheranforderung von nur wenigen Kilobytes, ist FreeRTOS für Mikrocontroller mit begrenztem Programmspeicher, wie dem SAMD21, geeignet. Es können Anforderungen an deterministische Reaktionen im Echtzeitsystem erfüllt werden, dadurch dass *Tasks* unabhängig von laufenden Prozessen innerhalb einer vordefinierten Zeit ausgeführt werden können [6].

Das *Application Programming Interface* (API) ermöglicht eine unkomplizierte und entwicklerfreundliche Implementierung von *Tasks*, *Queues*, *Semaphoren* und weiteren Funktionalitäten. *Tasks* können modular entwickelt und unabhängig getestet werden, wodurch die Entwicklung komplexer Echtzeitanwendungen effizienter gestaltet wird. Der Kernel von FreeRTOS ist in der Programmiersprache C geschrieben [6].

1.2.3.1 Tasks

Ein *Task* ist eine unabhängige Code-Einheit, die ähnlich wie ein *Thread* in anderen Betriebsystemen funktioniert. *Tasks* werden durch eine Funktion implementiert. Jeder *Task* muss eine Endlosschleife enthalten, da *Tasks* typischerweise kontinuierlich arbeiten, solange der Scheduler sie nicht beendet oder pausiert. Dies ist auch der Grund warum *Tasks* keine *return-Statements* enthalten dürfen [6].

Die Prioritätssteuerung der *Tasks* ist ein zentrales Konzept in Echtzeitbetriebssystemen. *Tasks* mit höherer Priorität verdrängen *Tasks* mit niedrigerer Priorität und erhalten bevorzugt Rechenzeit (präemptives *Scheduling*). Durch diese Mechanismen lassen sich zeitkritische Aufgaben zuverlässig umsetzen. Insgesamt ermöglicht die API zur *Task*-Erstellung und -

Verwaltung eine flexible und effiziente Nutzung der Systemressourcen, die sich präzise an die Anforderungen der Anwendung anpassen lassen [6].

Ein *Task* kann sich in einem von vier möglichen Zuständen befinden: *Running State*, *Ready State*, *Blocked State* und *Suspended State*. Ein *Task* im *Running State* wird aktuell von der CPU ausgeführt. Pro CPU-Kern kann sich zum selben Zeitpunkt immer nur ein *Task* im *Running State* befinden. *Tasks* im *Ready State* sind bereit zur Ausführung, warten jedoch darauf, dass die CPU verfügbar wird. Der *Scheduler* wählt den *Task* im *Ready State* aus, der die höchste Priorität besitzt und setzt ihn in den *Running State*. Ein *Task* befindet sich im *Blocked State*, wenn er auf ein bestimmtes Ereignis wartet. Beispiele für solche Ereignisse sind das Verstreichen einer definierten Verzögerungszeit (*Delay*), das Freigeben einer Semaphore oder das Eintreffen von Daten in eine *Queue*. Während ein *Task* im *Blocked State* ist, wird er nicht für die CPU-Zuweisung berücksichtigt. Ein Task im *Suspended State* wurde explizit angehalten und bleibt in diesem Zustand, bis dieser fortgesetzt wird. Dieser Zustand wird z.B. genutzt, um nicht benötigte Tasks temporär zu deaktivieren [6].

Jeder *Task* wird mit einer Priorität versehen, die eine Ganzzahl zwischen Null und einem in der Konfigurationsdatei definierten Maximum darstellt. Die Priorität Null stellt die niedrigste Priorität dar und wird in der Regel dem *Idle-Task* zugewiesen, der sicherstellt, dass immer mindestens ein *Task* ausgeführt wird, selbst wenn alle anderen *Tasks* blockiert oder angehalten sind. Seine Hauptaufgabe besteht darin, Kernel-Ressourcen freizugeben, z.B. den *Stack* von *Tasks*, die durch API-Funktionen gelöscht wurden. Zusätzlich kann der *Idle-Task* durch die Implementierung einer *Idle-Hook*-Funktion erweitert werden. Diese Funktion wird bei jeder Ausführung der *Idle-Task* aufgerufen und kann für spezifische Aufgaben genutzt werden, z. B. zum Energiemanagement oder zur Durchführung von Wartungsarbeiten innerhalb der Programmausführung. Durch die Integration der *Idle-Hook*-Funktion in die *Idle-Task* wird zusätzlicher *Overhead* vermieden, der durch die Erstellung eines separaten *Tasks* entstehen würde [6].

In Abbildung 4 ist ein Beispiel mit drei *Tasks* unterschiedlicher Priorität zu sehen. Die *Tasks* niederigerer Priorität werden von *Tasks* höherer Priorität verdrängt. Der *Idle-Task* wird kontinuierlich ausgeführt, falls kein anderer *Task* ausgeführt wird.

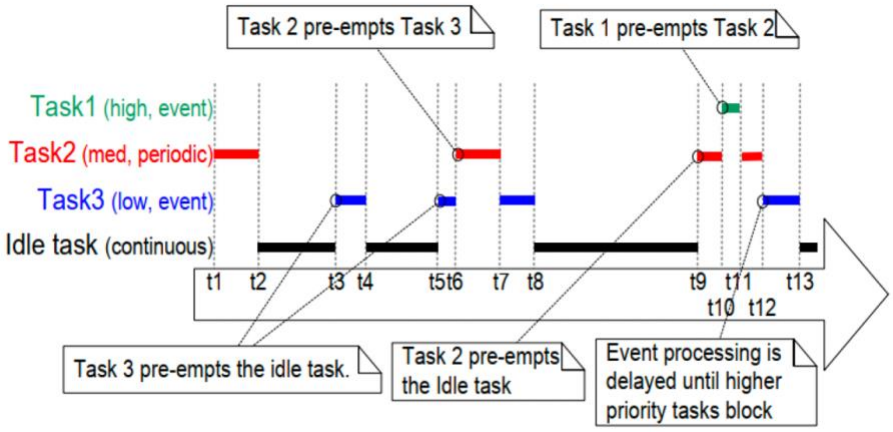

Abbildung 4: Zeitlicher Ablauf von Taskausführungen mit unterschiedlichen Prioritäten und preämptivem Scheduling [6].

Tasks mit derselben Priorität können beliebig oft erstellt werden. Wenn sich mehrere *Tasks* mit gleicher Priorität gleichzeitig im *Ready State* befinden, entscheidet der *Scheduler* mithilfe von *Time Slicing*, welcher Task als Nächstes ausgeführt wird [6] .

Time Slicing ist ein Mechanismus, der die CPU-Rechenzeit gleichmäßig auf Tasks mit derselben Priorität verteilt. Nach Ablauf eines *Time Slices* wird ein *Tick-Interrupt* ausgelöst, wodurch der *Scheduler* zum nächsten *Task* im *Ready State* wechselt. Die Länge des *Time Slice*s ist abhängig von der eingestellten Tickrate. Bei einer Tickrate von 1 kHz beträgt ein *Time Slice* beispielsweise1 ms. Nach jedem *Time Slice* werden die CPU-Register, der *Program Counter* und der *Stack Pointer* des aktuellen Tasks gesichert, und die Werte des neuen *Tasks* werden geladen, sodass dessen Ausführung fortgesetzt werden kann. Dieser Vorgang wird als *Context Switch* bezeichnet [6].

Time Slicing kann deaktiviert werden, indem der *Scheduler* in einen kooperativen Modus versetzt wird. In diesem Modus erfolgt ein *Task*-Wechsel gleicher Priorität nur, wenn der aktuell laufende Task dies explizit erlaubt [6].

1.2.3.2 Queues

Queues (Warteschlangen) bieten eine *thread*-sichere Methode, um Interprozesskommunikation zwischen *Tasks* zu ermöglichen. Die Datenstruktur einer *Queue* basiert auf dem FIFO-Prinzip (*First-In*, *First-Out*). Der sendende *Task* legt eine Variable in den *Queue-Buffer*. Diese Daten können von anderen *Tasks* gelesen werden. Eine *Queue* kann eine begrenzte Anzahl von Variablen aufnehmen, die bei der Erstellung der *Queue* festgelegt wird. Die Daten werden direkt in die *Queue* kopiert, anstatt nur einen *Pointer* auf die Daten zu

speichern. Dies hat den Vorteil, dass die Daten in der *Queue* sicher sind, selbst wenn die ursprüngliche Speicherstelle mit einem neuen Wert überschrieben wird. Beliebig viele *Tasks* können Daten in eine *Queue* schreiben oder Daten aus ihr lesen [6].

1.2.3.3 Interrupts

Wenn ein System auf äußere Signale reagieren soll, kann dies über einen *Interrupt* erfolgen. Dieser kann über eine Netzwerkschnittstelle oder einen einfachen Knopfdruck ausgelöst werden. Die meisten FreeRTOS-API-Funktionen können nicht außerhalb von *Tasks* aufgerufen werden, da diese vorrausetzen, dass der *Scheduler* läuft. Deshalb können diese ebenfalls nicht innerhalb einer *Interrupt Service Routine* (ISR) ausgeführt werden, weshalb FreeRTOS für viele API-Funktionen eine ISR-sichere Variante besitzt [6].

Eine ISR sollte möglichst kurz gehalten werden, während der Hauptteil der auszuführenden Aktion in einem *Task* verarbeitet wird. Dadurch stehen alle FreeRTOS-API-Funktionen uneingeschränkt zur Verfügung, die normalerweise innerhalb einer *Task* verwendet werden können. Um zu gewährleisten, dass der zugehörige *Handler-Task* unmittelbar nach der ISR ausgeführt wird, sollte dieser immer die höchste Priorität besitzen [6].

Um einen *Handler-Task*, nach einer ISR zu aktivieren, können binäre Semaphoren verwendet werden. Der *Handler-Task* wartet in diesem Fall auf eine Semaphore der ISR und wird nach deren Empfang ausgeführt. Ein erstelltes Semaphoren-*Handle* dient als Referenz, wenn die Semaphore gesendet oder übertragen wird [6].

1.2.3.4 Speicherverwaltung

FreeRTOS bietet verschiedene Möglichkeiten den *Heap*-Speicher zu verwalten. Die *Heap*-Größe wird in der Konfigurationsdatei festgelegt. Abhängig von der Anwendung kann ein geeigneter Algorithmus zur Speicherverwaltung gewählt werden. Der einfachste Algorithmus ist der *heap_1*-Algorithmus. Dieser reserviert Speicher für *Tasks* ausschließlich bevor der *Scheduler* gestartet wird. Dieser kann danach nicht mehr freigegeben werden. Ein *Array* wird in kleinere Blöcke aufgeteilt, in denen der *Stack* und der *Task Control Block* (TCB) der *Task* allokiert werden. Da der Speicher nur zu Beginn allokiert wird und keine Allokationen während des Programmverlaufes erfolgen, tritt keine Speicherfragmentierung auf. Diese Methode eignet sich jedoch nur, wenn keine dynamische Speicherallokation während des Programmverlaufes erforderlich ist.

Um dynamischen Speicher nutzen zu können, kann der *heap_4*-Algorithmus verwendet werden. Dieser nutzt den erstgrößten Speicherblock, der groß genug ist, um die angeforderten Daten zu Speichern. Speicherblöcke können freigegeben und neue Daten in freigewordenen Blöcke allokiert werden. Freie, nebeneinanderliegende Blöcke werden zu einem größeren Block zusammengeführt. Dadurch wird Speicherfragmentierung effektiv minimiert [6].

1.2.4 EMG

Zur Steuerung der Muskeln verwendet der Körper elektrische Signale, die durch Ionentransport in den Muskelzellen erzeugt werden. Diese Signale, sogenannte Aktionspotenziale, werden entlang der efferenten motorischen Nervenfasern zur motorischen Endplatte der Muskelfasern geleitet. Dort wird der Neurotransmitter Acetylcholin (ACh) freigesetzt, der an ACh-gesteuerte Kationenkanäle auf der Muskelfasermembran bindet. Die Öffnung dieser Kanäle ermöglicht den Einstrom von Natriumionen in die Muskelfaser, was zu einer lokalen Depolarisation führt. Diese Depolarisation aktiviert spannungsgesteuerte Natriumkanäle und löst ein Aktionspotenzial an der Membran aus [7].

Das Aktionspotenzial breitet sich entlang der Membran aus und führt zur Freisetzung von Kalziumionen aus dem sarkoplasmatischen Retikulum (SR). Die erhöhten Kalziumkonzentrationen im Zytosol fördert die Interaktion zwischen Aktin- und Myosinfilamenten, was die Muskelkontraktion bewirkt. Nach der Kontraktion werden die Kalziumionen aktiv zurück in das SR zurückgepumpt, wodurch der Muskel entspannt, bis ein neues Aktionspotenzial ausgelöst wird [7].

Um die Depolarisation der Muskelaktivität zu messen, kann ein Elektromyografie (EMG) verwendet werden. Hierbei werden typischerweise Elektroden auf der Hautoberfläche angebracht, um die elektrische Aktivität der Muskeln zu erfassen. Ein Oberflächen-EMG-Signal (sEMG) liegt üblicherweise im Mikrovolt- bis niedrigen Millivolt-Bereich [8,9]. Mehrere Faktoren beeinflussen die Signalqualität, darunter der Abstand der Elektroden vom aktiven Muskelareal, Gewebebeschaffenheit (z.B. adipöses Gewebe oder Ödeme), die Intensität der Muskelkontraktion, die Qualität des Haut-Elektroden-Kontakts und die Eigenschaften des eingesetzten Verstärkers [8].

Um Signalvariabilität bei gleichen Messungen zu minimieren, sollte nach Möglichkeit stets derselbe Typ von Elektroden und Verstärker verwendet werden. Die Qualität des Haut-Elektroden-Kontakts spielt eine entscheidende Rolle für das Signal-Rausch-Verhältnis, da sie die Impedanz beeinflusst, welche relativ konstant gehalten werden sollte [9].

Das analoge EMG-Signal wird üblicherweise über einen Differentialverstärker aufgenommen. Typische Bandpassfiltereinstellungen für sEMG liegen zwischen 10–20 Hz (Hochpass) und 500–1000 Hz (Tiefpass) [9]. Der Hochpassfilter eliminiert Artefakte, die typischerweise unterhalb von 10 Hz liegen, während der Tiefpassfilter Signalaliasing und Rauschen reduziert, das durch hohe Frequenzanteile oberhalb der Abtastrate des Verstärkers entstehen könnte und somit das Nyquist-Theorem verletzen würde. Störungen durch Netzfrequenzkomponenten können durch einen Notch-Filter reduziert werden, allerdings besteht das Risiko, dass in diesem Frequenzbereich auch Nutzsignale verloren gehen [9].

Typische Verstärkungswerte liegen im Bereich von 100 bis 10.000 [9][10]. Die wichtigste Verstärkungsphase ist die Vorverstärkungsphase, da sie dem Eingangssignal am nächsten ist. Wichtige Kenngrößen hierbei sind eine hohe Eingangsimpedanz, Unterdrückung von Gleichstromsignalen (DC), hohe Gleichtaktunterdrückung (CMR) und die räumliche Nähe zwischen Signalquelle und Verstärker [10].

Durch eine bipolare Elektrodenanordnung werden Signale, die an beiden Elektroden anliegen, subtrahiert (Differenzbildung). Dies ermöglicht die Unterdrückung von Störsignalen, die an beiden Elektroden gleichermaßen auftreten, wie beispielsweise Netzspannung, Muskelaktivität aus weit entfernten Bereichen oder elektromagnetische Störungen [8]. Das Maß für die Fähigkeit, solche gemeinsamen Störsignale zu unterdrücken, ist das *Common Mode Rejection Ratio* (CMRR) [8]. Ein Beispiel ist der AD7177, der eine CMRR von 120 dB bei 50/60 Hz aufweist [11].

Die Impedanz bei EMG-Messungen auf der Hautoberfläche liegt typischerweise unter 50 kΩ [9]. Eine hohe Eingangsimpedanz des Verstärkers ist wichtig, um die Quelle nicht zu belasten, wodurch der Ladeeffekt minimiert wird und das gemessene Signal weniger verzerrt wird [9].

Eine Referenzelektrode wird eingesetzt werden, um Störströme abzuleiten [12]. Die Erdungselektrode ist dabei in der Regel mit der Masse der Schaltung verbunden und dient als Nullpunkt, um die Potentialdifferenzen der EMG-Elektroden zu messen[13]. Sie wird meist auf der Haut über einem knöchernen Bereich platziert, der wenig Muskelaktivität aufweist, wie zum Beispiel an der Stirn oder am Ellenbogen [14].

EMG-Elektroden besitzen häufig eine mit Gel beschichtete Oberfläche, die als Schnittstelle zwischen Haut und Elektrode dient. Als Elektrolyt im Gel wird häufig Silber-Silberchlorid (Ag/AgCl) verwendet. Dieses Material hat ein geringes Halbzellenpotential von 220 mV und ist nicht polarisiert. Durch Ag/AgCl können Elektronen zwischen Elektrode und Elektrolyt fließen. Bei 10 Hz und gutem Hautkontakt besitzen EMG-Elektroden in der Regel eine Impedanz von ca. 5 kΩ [15].

2 Material und Methoden

2.1 Material

Für die Zusammenschaltung der Schaltkreise wurde das *Brick'R'knowledge Bio Feedback Set* genutzt, da es eine modulare und übersichtliche Implementierung von Mikorocontroller-basierten Anwendungen bietet. Als Echtzeitbetriebssystem (RTOS) kam FreeRTOS zum Einsatz, dessen API-Funktionen eine effiziente Entwicklung von Echtzeitsystemen und eine flexible Aufgabenverteilung ermöglichen. Für die Softwareentwicklung wurden *Visual Studio Code* in Kombination mit *PlatformIO* genutzt, die eine Entwicklungsoberfläche bereitstellen, die speziell für die Anforderungen von eingebetteten Systemen geeignet sind.

2.1.1 Brick'R'knowledge Set

Brick'R'knowledge ist ein Stecksystem, das von der Firma ALLNET GmbH Computersysteme hergestellt wird [16]. Es werden zahlreiche Elektroniksets angeboten, die dem Anwender Elektronik in einem kompakten Format näherbringen sollen. Das Set besteht aus Bausteinen, die zusammengesteckt werden können, um Schaltkreise zu formen. Durch das integrierte Format werden unübersichtliche Schaltungen, die mit vielen Leitungen zusammengebaut werden müssen, in eine kompakte Form gebracht. Ziel von Brick'R'knowledge besteht darin den einfachen Einsatz von Elektronik in Schulen und Hochschulen zu ermöglichen [16].

Das eingesetzte Set ist das *Bio Feedback Set*, das darauf ausgelegt ist, Biosignale zu messen. Mit diesen Biosignalen können beispielsweise Elektrokardiogramme (EKG), Elektromyogramme (EMG) oder Elektroenzephalogramme (EEG) aufgenommen werden.

In Abbildung 5 sind *Bricks* des *Bio-Feedback-Sets* zu erkennen. Das Herzstück des Sets, um diese Signale verarbeiten zu können, ist der MKR WiFi 1010. Dies ist ein Board mit einem Mikrocontroller der SAMD-Familie, dem SAMD21 [17]. Das Board wird auf einen für das Board vorgesehenen Block platziert. Dieser besitzt die nötigen Anschlüsse für die restlichen *Bricks*, um mit ihnen über das SPI- oder I^2C-Interface zu kommunizieren.

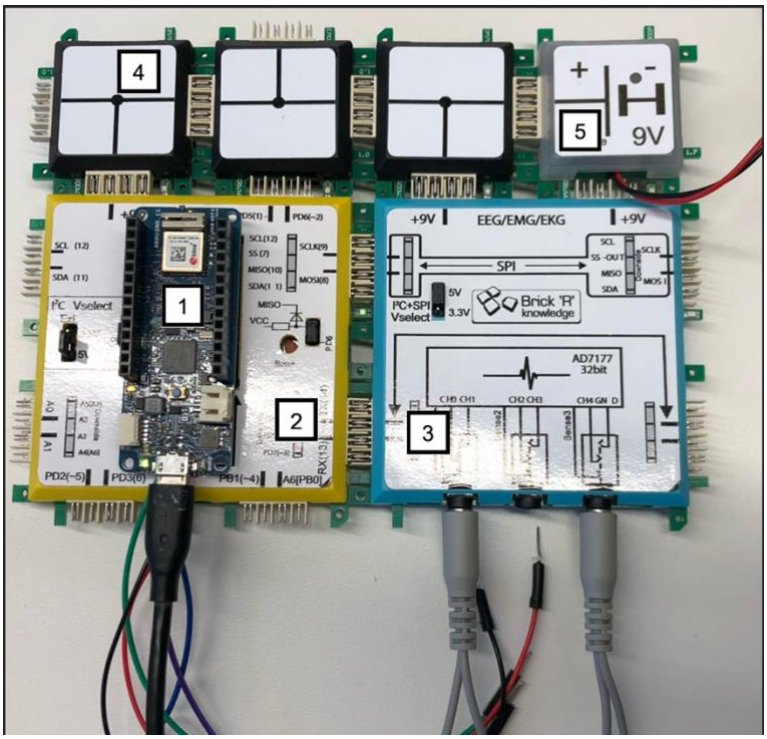

Abbildung 5: Brick'R'knowledge-Set Mikrocontroller- und EEG/EMG/EKG-Brick.

1. *MKR WiFi 1010* 2
2. *Mikrocontroller-Brick*
3. *EEG/EMG/EKG-Brick*
4. *T-Stück*
5. *9V-Brick*

Der EEG/EMG/EKG-*Brick* wird verwendet, um die jeweiligen Biosignale aufzunehmen. Die Signale werden mit einem Analog-Digital-Wandler (AD-Wandler) aufgenommen und anschließend an den Mikrocontroller weitergegeben. Der AD-Wandler besitzt fünf Kanäle, die über drei sogenannte *Sense*-Anschlüsse als Analogeingänge genutzt werden. Über die ersten zwei *Sense*-Eingänge laufen jeweils zwei Kanäle, und der dritte *Sense*-Anschluss besitzt neben einem Kanal auch einen Masseanschluss. Zwei Kabel werden gleichzeitig über einen Klinkenstecker angeschlossen und erhalten dadurch Zugriff auf zwei Kanäle oder auf einen Kanal und den Masseanschluss [18].

Damit komplexere Schaltungen realisiert werden können, ist es möglich, T-*Bricks* zu verwenden, die als einfache Leiterbahnen dienen, die die einzelnen *Bricks* miteinander verbinden. Der Mikrocontroller und EEG/EMG/EKG-*Brick* werden mit einer Spannungsquelle betrieben, wofür ein 9V-*Brick* verwendet wird. An diesem wird eine 9V-Batterie angeschlossen, um die Schaltung mit Spannung zu versorgen.

Weitere *Bricks* sind im Set enthalten, wie ein OLED-*Brick*, der für das Darstellen von Texten und Grafiken verwendet werden kann, ein Pulsoxymeter-*Brick,* der mithilfe zweier Dioden die Sauerstoffsättigung und den Puls messen kann und ein LED-*Brick* mit zwei Leuchtdioden. Das

OLED-Display und der Pulsoxymeter-*Brick* müssen separat mit 9 Volt betrieben werden und kommunizieren über einen I^2C-Bus [18].

2.1.1.1 AD7177-2

Besondere Signifikanz hat der AD-Wandler, der sich im EEG/EMG/EKG Brick befindet. Dieser ist der AD7177-2 und besitzt eine 32-Bit-Auflösung, kann bei 5 SPS (Samples pro Sekunde) effektiv 24,6 Bits oder 32 Bits mit Rauschen liefern, wobei Rauschen durch Störungen im Signal die tatsächliche nutzbare Genauigkeit verringert. Er unterstützt eine Abtastrate von bis zu 10 kSPS [11]. Die Datenübertragung erfolgt mit 8-Bit-Anweisungen und 8–24-Bit-Datenpaketen [11]. Bei Frequenzen von 50–60 Hz liegt die Gleichtaktunterdrückung bei über 120 dB und durch integrierte Buffer-Schaltungen bietet der AD-Wandler eine hohe Eingangsimpedanz (abhängig von der Verstärkung) [11]. Der Wandler benötigt keine weitere Vorverstärkung und liefert das verstärkte Signal direkt an den Mikrocontroller.

Der AD-Wandler besitzt fünf Kanäle (AIN0-4), die beim *Brick* mittels der *Sense*-Eingänge zur Verfügung stehen. Die serielle Schnittstelle wird rechts in Abbildung 6 dargestellt, mit der externe Geräte verbunden werden können. Im Inneren befindet sich die Verschaltung.

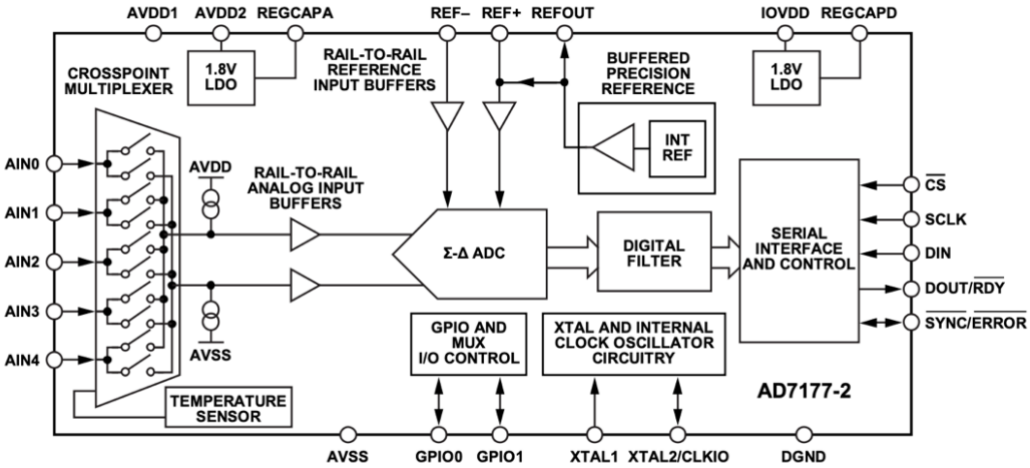

Abbildung 6: AD7177-2 Blockdiagramm des Pin-Layouts und der internen Struktur [11].

Der AD7177-2 besitzt ein konfigurierbares Register, welches über die SPI-Kommunikationsschnittstelle angesprochen werden kann. Die Einstellungsmöglichkeiten können dem Bitregister des Datenblattes entnommen werden [11]. Die Kanäle werden über das *Channel Register* konfiguriert, um beispielsweise als differenzielle oder einzelne Eingänge verwendetet zu werden. Er kann auch als *Multiplexer* mit mehreren Kanälen eingesetzt werden [11]. Die Betriebs- und Schnittstellen-Modi können ebenfalls je nach Anwendung individuell

eingestellt werden. Der *Continuous Conversion Mode* ist zum Beispiel für Echtzeitanwendungen geeignet, da in diesem Modus Daten kontinuierlich erfasst werden [11].

Der Wandler besitzt mehrere Digitalfilter, die in Abbildung 6 in der internen Schaltung zu erkennen sind. Diese können individuell eingestellt werden, wie beispielsweise mit Sinc3- oder Sinc5-Filtern, die niederfrequentes Rauschen unterdrücken. Außerdem können *Offset*- oder Verstärkungs-Einstellungen vorgenommen werden [11].

2.1.1.2 MKR WiFi 1010

Im Set kommt der Arduino MKR WiFi 1010 zum Einsatz. Das Board basiert auf einem SAMD21 Mikrocontroller. Dieser verwendet einen 32-Bit ARM Cortex-M0+ Prozessor mit einer Taktfrequenz von 48 MHz. Das Board besitzt außerdem ein integriertes U-blox NINA-W102 Modul, welches IEEE 802.11 b/g/n sowie Bluetooth 4.2 unterstützt [17]. Dieses kann verwendet werden, um Netzwerkkonnektivität und direkte Gerätekommunikation über Bluetooth zu ermöglichen.

Wie in Abbildung 7 zu sehen ist, besitzt das Board viele Anschlussmöglichkeiten, darunter acht digitale Ein-/Ausgangspins, sieben analoge Pins und dreizehn PWM-Pins. Der Flash-Speicher für den Programmcode beträgt 256 KB und für dynamische Daten stehen 32 KB SRAM zur Verfügung [17].

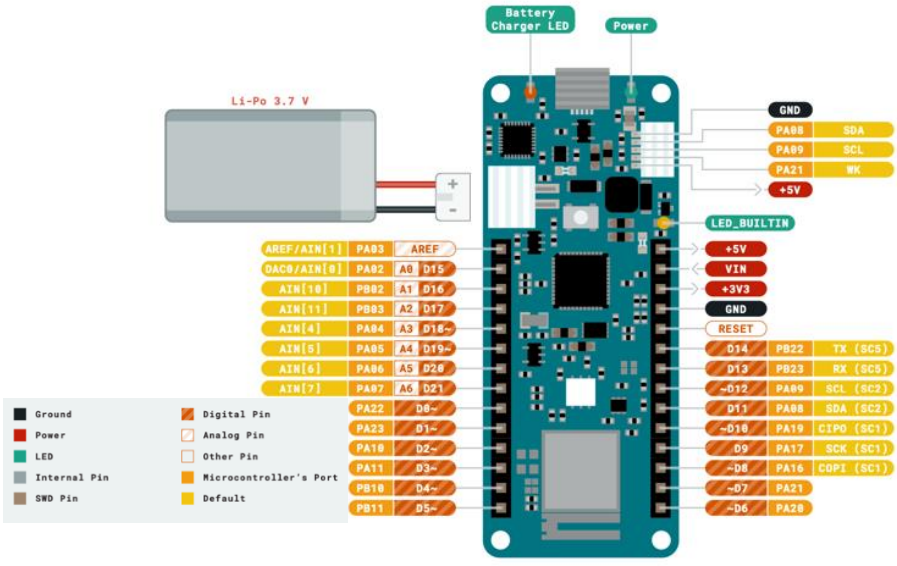

Abbildung 7: MKR WiFi 1010 Pinlayout [19].

Die Energieversorgung kann flexibel über USB (5V), eine Batterie oder einen LiPo-Akku gewährleistet werden. Die Betriebsspannung beträgt 3,3 V, während der USB-Anschluss mit 5 V betrieben wird. SPI, I^2C und USART können als Kommunikationsschnittstellen mit externen Geräten genutzt werden [17].

2.1.1.3 Beispielprogramme

Im Set sind Beispielprogramme enthalten, auf die die Programme dieser Arbeit zum Teil aufbauen. Insbesondere die Ansteuerung der SPI- und I^2C-Schnittstellen des EEG/EMG/EKG-*Bricks* wird aus den Beispielprogrammen übernommen. Die Beispielprogramme zeigen, wie die *Bricks* genutzt werden können, um Biosignale aufzunehmen und diese auf der Konsole oder einem Display auszugeben. Für EKG-, EMG- und EEG-Messungen gibt es viele Beispiele.

Abbildung 8 zeigt, wie ein Beispielprogramm für eine EMG-Messung aufgebaut ist. Dieses nutzt den EEG/EMG/EKG-*Brick* und ein I^2C-OLED-Display. Die Kommunikation der Bricks wird über SPI- oder I^2C-Schnittstellen ermöglicht. In Quellcode 1 ist Code zum setzen des AD-Wandlers dargestellt. So können Einstellung benutzerdefiniert getroffen werden.

Quellcode 1: Code zum setzen des Registers des AD7177-2

```
SPI.transfer(0x01); SPI.transfer16(0x800C);
SPI.transfer(0x02); SPI.transfer16(0x0442);
SPI.transfer(0x06); SPI.transfer16(0x060d);
```

Bei einer EMG-Messung am Arm wird dieses Signal am OLED-Display grafisch dargestellt. Die Signalverarbeitung erfolgt durch den eingesetzten AD7177-2 und den MKR WiFi 1010. Über die Software wird der AD-Wandler eingestellt, wodurch unter anderem die Kanäle ausgewählt und Digitalfilter eingestellt werden, um die Signalqualität zu verbessern. Zusätzlich wird das Signal numerisch auf der Konsole ausgegeben, was eine detailliertere Analyse der erfassten Signale ermöglicht.

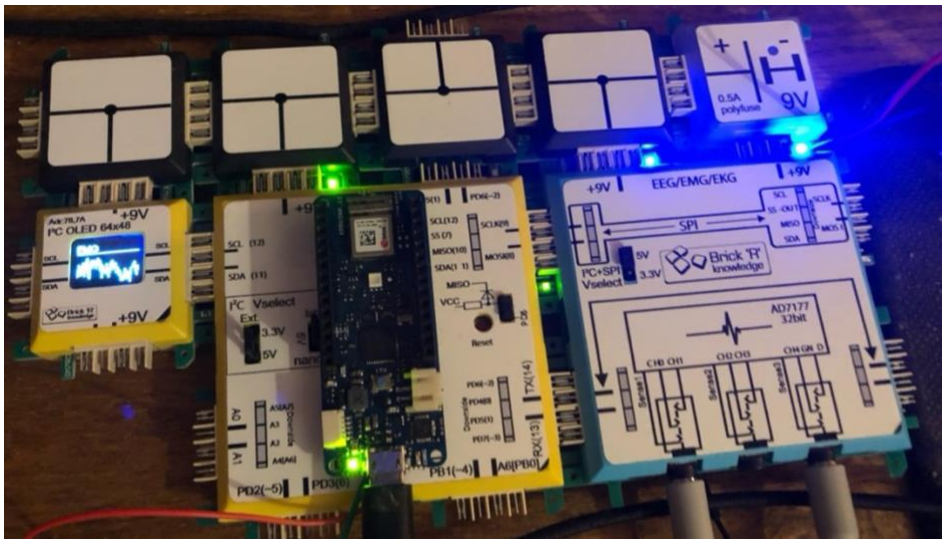

Abbildung 8: EMG-Messung-Aufbau des Beispielcodes mittels EEG/EMG/EKG-Block und OLED-Display.

2.1.2 FreeRTOS (Version 10.2.1)

Es wurden FreeRTOS-Funktionalitäten verwendet, um Programme dieser Arbeit zu entwickeln. Dabei kamen Schlüsselfunktionalitäten wie *Tasks*, *Queues* und *Semaphoren* zum Einsatz. Mit FreeRTOS können Anforderungen an deterministische Reaktionen im Echtzeitsystem erfüllt werden. da *Tasks* unabhängig von laufenden Prozessen innerhalb einer vordefinierten Zeit ausgeführt werden können.

Tasks werden durch eine Funktion implementiert, die dem folgenden Prototyp entspricht:

Quellcode 2: Task-Funktion Prototyp [6]

```
void vATaskFunction( void * pvParameters );
```

Der Parameter *pvParameters* ist der Wert, der dem *Task* übergeben wird. Die Erstellung einer Task erfolgt über die API-Funktion *xTaskCreate()*, die einen Task erstellt und für den *Scheduler* verfügbar macht. Die Funktion hat den folgenden Prototyp:

Quellcode 3: Task-Erstellung Prototyp [6]

```
xTaskCreate( TaskFunction_t pvTaskCode, const char * const pcName,
configSTACK_DEPTH_TYPE usStackDepth, void * pvParameters, UBaseType_t uxPriority,
TaskHandle_t * pxCreatedTask );
```

PvTaskCode ist der *Pointer* (Zeiger) auf die Funktion, die die Logik des *Tasks* enthält. *PcName* ist ein eindeutiger Name des *Tasks*, der vor allem zu *Debugging*-Zwecken dient. Der Name hat keinen Einfluss auf die Ausführung der *Task*. *UsStackDepth* gibt die Größe des zugewiesenen Stacks in Worten an. Die tatsächliche Größe hängt von der Wortbreite des Prozessors ab. Zum Beispiel bei einer Wortbreite von 32 Bit entspricht ein Wert von 128 einer Speicherzuweisung von 512 Bytes. *PvParameters* ist ein *Pointer* auf die Parameter, die dem *Task* übergeben werden sollen. Dieser Parameter ist optional und kann NULL sein. *UxPriority* gibt die Priorität des *Tasks* an, die bestimmt, in welcher Reihenfolge der Scheduler Tasks ausführt. Gültige Werte beginnen bei 0 (niedrigste Priorität). Die maximale Priorität wird in der Konfigurationsdatei des RTOS definiert. Tasks können auch die gleiche Priorität haben. *PxCreatedTask* ist ein *Pointer* auf ein *Handle*, das die neu erstellte Task repräsentiert. Dieses *Handle* ermöglicht die Steuerung des Tasks über andere API-Funktionen wie das Pausieren oder Löschen von Tasks [6].

2.1.2.1 Queues

Der Prototyp um eine *Queue* in FreeRTOS zu erstellen sieht wie folgt aus:

Quellcode 4: Queue-Erstellung Prototyp [6]

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize );
```

UxQueueLength gibt an, wie viele Variablen von der *Queue* zur selben Zeit gehalten werden können. *UxItemSize* ist die Variablengröße in Bytes, die in der *Queue* gespeichert werden kann. Um eine *Queue* erfolgreich zu erstellen, muss ein *Queue-Handle* deklariert werden, das mit einem *xQueueCreate*-Befehl gleichgesetzt wird. Werte können an eine *Queue* gesendet oder von einer *Queue* empfangen werden. Für die jeweiligen Funktionen gibt es unterschiedliche Prototypen.

Quellcode 5: QueueSend-Prototyp [6]

```
BaseType_t xQueueSend( QueueHandle_t xQueue,
const void * pvItemToQueue,
TickType_t xTicksToWait );
```

xQueue ist der *Handle*, mit dem eine *Queue* erstellt wurde. *pvItemToQueue* ist ein *Pointer* auf die Variable, die in die *Queue* gespeichert werden soll. *xTicksToWait* Gibt die Zeit an, die der *Task* warten soll, bis die *Queue* frei wird. Die *QueueReceive*-API-Funktion besitzt einen fast identischen Aufbau.

Quellcode 6: Queue-Receive Prototyp [6]

```
BaseType_t xQueueReceive( QueueHandle_t xQueue,
void * const pvBuffer,
TickType_t xTicksToWait );
```

Der *pvBuffer* ist ein *Pointer* auf den Speicher, in der die Daten kopiert werden. Nur mit diesen drei Prototypen kann die Kommunikation zwischen Tasks ermöglicht werden. Ein *Receive-Task* kann von mehreren Tasks Daten erhalten. Dabei ist die Priorisierung ausschlaggebend für das Verhalten der Tasks. Falls ein *Receive-Task* eine niedrigere Priorität als der *Send-Task* besitzt, würde die *Queue* zuerst volllaufen, bevor ein Wert vom *Receive-Task* erhalten werden kann, da der *Send-Task* im *Blocked State* sein muss. Umgekehrt würde mit höherer Priorität die *Queue* nicht volllaufen können, da der *Receive-Task*, den ersten Wert direkt abgreifen würde [6].

2.1.2.2 Interrupts und Semaphoren

Nachdem ein *Interrupt* ausgelöst, kann innerhalb der ISR eine Semaphore an einen *Handler-Task* gesendet werden. Die binäre Semaphore wird wie folgt erstellt:

Quellcode 7: Semaphore-Erstellung-Funktion [6]

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

Mit der *xSemaphoreGiveFromISR()*-Funktion wird die Semaphore von der ISR übermittelt. Diese hat den folgenden Prototypen:

Quellcode 8: SemaphoreGive-Funktion [6]

```
BaseType_t xSemaphoreGiveFromISR( SemaphoreHandle_t xSemaphore,
BaseType_t *pxHigherPriorityTaskWoken );
```

Der Handle wird referenziert, und wenn die Funktion aufgerufen wird, wird *pxHigherPriorityTaskWoken* auf *pdTRUE* gesetzt. Dies kann mit *portYIELD_FROM_ISR()* verwendet werden, um einen *Context Switch* zu bewirken. Die Semaphore kann dann von dem *Handler-Task* mit der *xSemaphoreTake()*-Funktion angenommen werden.

Quellcode 9: SemaphoreTake-Funktion [6]

```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);
```

Dieser referenziert den *Task-Handle* und wartet bis zu einer definierten Zeit auf die Semaphore. Im Fall eines ISR-*Handlers* sollte dieser auf *portMAX_DELAY* gestellt werden, damit er bis auf unbestimmte Zeit wartet [6].

2.1.3 Entwicklungswerkzeuge

2.1.3.1 Visual Studio Code (Version 1.93.1)

Visual Studio Code (VS Code) ist ein Open-Source-Code-Editor, der von Microsoft entwickelt wurde. Im Gegensatz zu Visual Studio ist VS Code keine integrierte Entwicklungsumgebung (IDE) und verfügt daher nicht über viele der Entwicklungstools, die in einer IDE enthalten sind. Der Editor unterstützt eine Vielzahl von Programmiersprachen und *Frameworks* und eignet sich besonders für die Entwicklung von eingebetteten Systemen.

VS Code bietet Funktionen wie *Syntax-Highlighting*, *Bracket-Matching* und viele weitere nützliche Tools. Durch seine kompakte Grundinstallation und die Möglichkeit ihn modular zu erweitern, ist VS Code flexibel einsetzbar. Spezifische Erweiterungen können installiert werden, um individuelle Entwicklungsanforderungen abzudecken.

2.1.3.2 PlatformIO (Core 6.1.16)

PlattformIO ist ein plattformübergreifendes und architekturunabhängiges Werkzeug für eingebettete Systeme, das speziell auf die Anforderungen von Mikrocontrollern ausgelegt ist [20]. Es wurde als Erweiterung für VS Code verwendet und unterstützt zahlreiche Mikrocontroller-Familien, wie z. B. ARM, AVR, ESP32 oder STM32.

PlatformIO wurde verwendet, um mit der Programmiersprache C++ den Quellcode zu entwickeln. PlatformIO bietet viele Werkzeuge, die die Programmierung erleichtern, darunter Debugger und Testwerkzeuge, die direkt in der Software integriert sind [20]. Die Bibliotheken können mithilfe eines integrierten Suchwerkzeugs gefunden und in Projekte eingebunden werden. Projekte, die mit PlatformIO erstellt werden, enthalten eine *platform.ini*-Datei, über die beispielsweise *Github*-Links für Bibliotheken oder interne Dateipfade eingefügt werden können.

2.1.3.3 FreeRTOS Library

FreeRTOS stellt für viele Mikrocontroller Portierungen zur Verfügung, um die Bibliotheken für die spezifische Hardware kompatibel zu machen. Der Sourcecode kann auf der offiziellen Homepage heruntergeladen werden. Dieser beinhaltet die *Source*- und *Header*-Dateien, die für alle Ports gleich sind und die für den Port relevant sind. Für den SAMD21 existiert kein offizieller Port [21].

2.1.4 AVR Debugger

Im Rahmen dieser Arbeit wurde auch eine Debugging-Schnittstelle für den MKR WiFi 1010 gefertigt. In der Entwicklung von *Embedded*-Systemen können *Debugging*-Instrumente verwendet werden, um *Bugs* im System leichter zu erkennen oder das Systemverhalten besser analysieren zu können. Durch das Setzen von *Breakpoints* wird der Code an einer spezifischen Stelle gestoppt. Dies ermöglicht eine einfachere Analyse des Systemverhaltens. Unter anderem können so Variablen und Speicher während der Ausführung überwacht werden.

Als Debugger wird der Atmel ICE verwendet. Dieser ist kompatibel mit Mikrocontrollern der SAM- und AVR-Familie. Um eine Schnittstelle herzustellen, stehen wie in Abbildung 9 zu sehen, auf der unteren Seite des Boards Interface-Pins zur Verfügung, die mit dem Debugger verbunden werden können. Es wurde eine Verbindung zum Debugger gewährleistet, indem Leitungen direkt auf die Interface-Pins gelötet wurden. Damit wird ermöglicht, dass das Board gleichzeitig mit dem Brick'R'Knwowledge Set verwendet werden kann, da dadurch zwischen Brick und Board ausreichend Spielraum vorhanden ist.

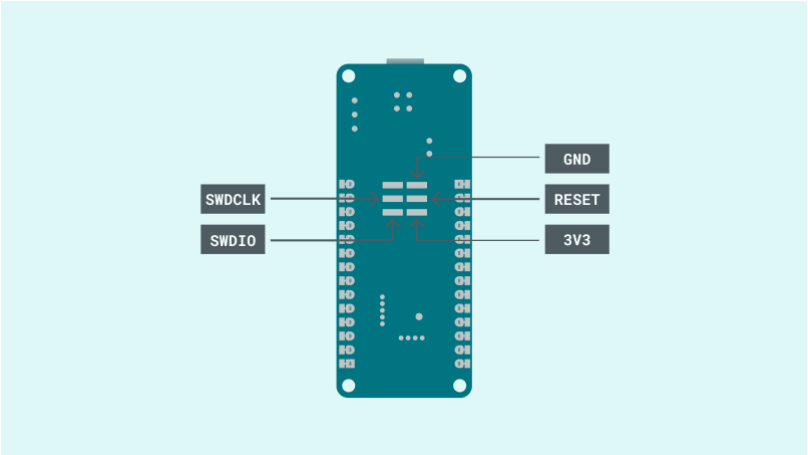

Abbildung 9: MKR WiFi 1010 Debugger-Interface-Pins [22].

2.2 Methoden

2.2.1 Implementierung FreeRTOS auf dem SAMD21

FreeRTOS verfügt über eine große Sammlung von Ports, die für verschiedene Mikrocontroller verwendet werden können. Diese können von der FreeRTOS-Homepage direkt heruntergeladen und genutzt werden. Die Ports besitzen eine vordefinierte Datenstruktur, die wie in Abbildung 10 die eingehalten werden sollte, damit diese funktionieren. FreeRTOS beinhaltet grundlegende Dateien, die von allen Ports geteilt werden. Diese beinhalten die grundlegende FreeRTOS-Funktionen wie den *Scheduler*, *Tasks*, *Queues* usw. Jeder Port benötigt spezifische Dateien für die jeweilige Prozessorarchitektur. Diese beinhalten *Compiler*- und RTOS-Code um FreeRTOS auf der spezifischen Architektur funktionsfähig zu machen.

FreeRTOS
|
➔ Source (FreeRTOS Kernel-Dateien)
|
➔ include (FreeRTOS *Header*-Dateien)
|
➔ Portable (Architekturspezifischer Code)
|
➔ Compiler (Compiler for the Port)
➔ MemMang (Heap-Datei für Speicher-Algorithmus)

Abbildung 10: FreeRTOS Datenstruktur [23].

Für Ports, die noch nicht existieren, gibt es *Templates*, um diese selbst zu programmieren. Da für den SAMD21 kein offizieller Port zur Verfügung steht, müsste ein neuer händisch programmiert werden. Allerdings konnte über das PlatformIO-Suchwerkzeug ein Port gefunden werden, der für den SAMD21 zugeschnitten ist [24].

Dieser Port besitzt alle nötigen FreeRTOS-Funktionen, um die Projekte dieser Arbeit entwickeln zu können. Die Datenstruktur wurde allerdings soweit geändert, dass alle Dateien sich in einem *Source*-Ordner abgelegt sind.

Mit der *Kdiff3*-Applikation wurde verglichen, inwieweit die Dateiein des Ports von den grundlegenden FreeRTOS Dateien abweichen. Im Port wird die FreeRTOS-Kernel Version 10.2.1 verwendet weshalb, dieser mit der selben Grundversion verglichen wurde.

Die Abbildung 11 zeigt, dass viele der Dateien identisch sind. Wenn beide Zeilen in den Spalten A und B grün sind, ist der Inhalt beider Dateien identisch. Wenn eines der Kästchen rot ist, bedeutet dies, dass der Inhalt voneinander abweicht. Wenn ein Kästchen schwarz ist, ist diese Datei in einem der Ports nicht vorhanden. Bei den abweichenden Dateien wurde der Code in der Regel um einige Zeilen ergänzt.

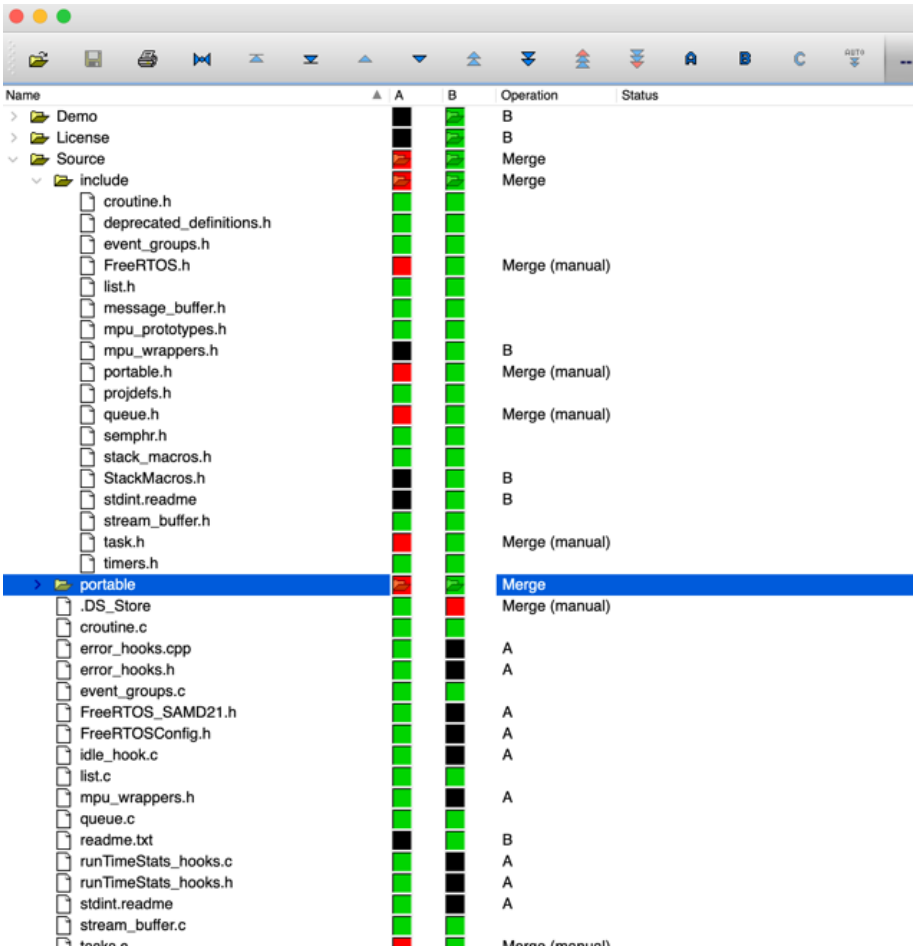

Abbildung 11: SAMD21-Port Diff-Verlgeich mit FreeRTOS Grundinstallation.

Das Ersetzen der Port-Dateien mit Dateien einer neueren FreeRTOS Version (v10.5.1) konnte erfolgreich kompiliert werden und grundlegende Funktionen waren funktionsfähig. Aufgrund der minimalen Abweichungen kann nicht ausgeschlossen werden, dass dadurch *Bugs* entstehen.

2.2.2 Entwicklung von Testprogrammen

Um FreeRTOS auf dem SAMD21 zu testen, wurden mehrere Testprogramme entwickelt. Diese sollen die Funktion von FreeRTOS verdeutlichen und prüfen, wie weit die Hardware für bestimmte Anforderungen belastet werden kann.

2.2.2.1 Entwicklung des Codes

Der erste Test prüft, wie viele *Tasks* erstellt werden können, bevor der *Heap* des Mikrocontrollers voll läuft. In einer *for*-Schleife werden so viele Tasks erstellt, bis der *Heap* des *Controllers* voll ist.

Quellcode 10:Schleife zum Erstellen von beliebig vielen Tasks

```
for(uint8_t i = 0; i < MAX_NUM; i++)
{
        tasknum[i] = i;
        xTaskCreate( vTask,"Math", 128, (void*)&tasknum[i], 1, NULL);
}
```

Jeder *Task* gibt die Größe des *Heaps* aus, diie mit der *xPortGetFreeHeapSize()*-Funktion zurückgegeben wird. Die Funktion gibt keinen Aufschluss über die Fragmentierung des Codes.

Im zweiten Test wird das *Task*-Verhalten geprüft, wenn dieser aufgrund zu hoher Berechnungszeiten nicht mehr ausgeführt werden kann. Die Pins 0 bis 7 werden als digitale Ausgänge verwendet. Beim Stresstest sollen acht *Tasks* diese abwechselnd ansteuern. Jeder Pin bekommt eine Delayzeit von 500 ms und die Priorität wird absteigend vom Pin 7 zugeteilt. Wodurch Pin 7 die höchte Priorität hat und Pin 0 die niedrigste.

Während der Tasks wird eine einfache Berechnung durchgeführt:

Quellcode 11: Schleife zur Belastung des Systems

```
for (int i = 0; i < 400000; i++)
{
        stressVar += i * i;
}
```

Die *Tasks* geben in der Konsole aus, wie viele Ticks und wie viele Zyklen der *Task* durchlaufen hat. Die Tick-Anzahl kann mit der Funktion *xTaskGetTickCount()* abgerufen werden. Diese wird

mit *xLastWakeTime* = *xTaskGetTickCount()* initialisiert, sodass zu beginn der *Task* auf Null gesetzt wird. Die verstrichene Zeit kann mit der *micros()*-Funktion ermittelt werden. Wenn dann ein Start- und Endzeitpunkt aufgenommen wird, kann so die verstrichene Zeit berechnet werden.

Der Mikrocontroller arbeitet mit einer Taktfrequenz von 48 MHz, was in Formel 1 dargestellt wird. Die Taktzyklen werden auf der Konsole ausgegeben.

Formel 1: Berechnen der Zyklen

$$Z = t \cdot f$$

$Z: Anzahl\ der\ Zyklen$

$t: Zeit$

$f: Frequenz$

2.2.3 Steuern eines Aktuators per EMG-Messung

Ein Fingermodell wurde entwickelt (Abbildung 12), das mittels eines Servomotors (SAVÖX SC-0251 MG) angetrieben und per EMG-Signal angesteuert wird. Der Servomotor besitzt drei Leitungen. Zwei davon dienen zur Spannungsversorgung mit 5 V (Zuleitung und Masse) und eine Leitung dient zur Steuerung des Motors mittels PWM-Signal. Im Projekt werden mehrere FreeRTOS-Funktionen realisiert, die auf Funktionalität und Echtzeitfähigkeit geprüft werden.

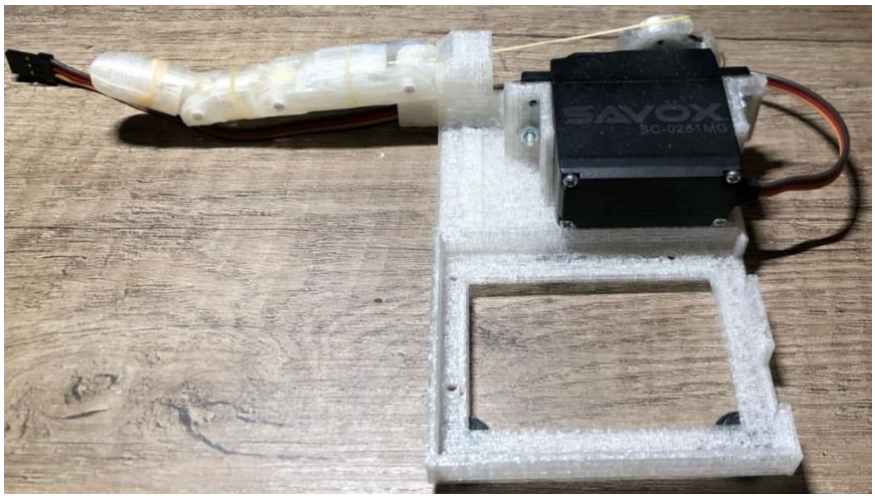

Abbildung 12: Fingermodell angetrieben mittels Servomotors.

In Abbildung 13 wird dargestellt wie das Signal bis zum Fingermodell gelangt. Durch Muskelaktivität wird ein EMG-Signal erzeugt, das verstärkt und gefiltert wird und anschließend per SPI-Kommunikation an den an den Mikrocontroller übergegeben wird. Dieser generiert das PWM-Signal, das an die Steuerleitung des Fingermodells weitergegeben wird.

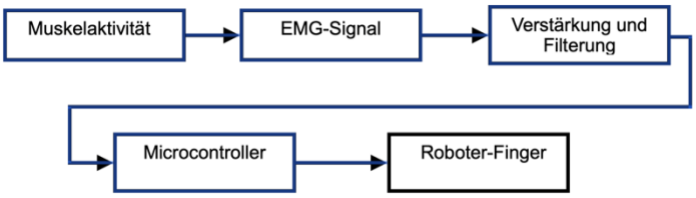

Abbildung 13: Blockdiagramm des Signalflusses von Muskelaktivität bis Aktuierung des Fingermodels.

Der Beispielcode des Brick'N'Knowledge-Sets dient bei der Entwicklung als Grundlage. Vor allem wurde der Code zur SPI-Kommunikation zwischen EMG-Brick und Mikrocontroller genutzt.

In Abbildung 14 ist der Messaufbau zu sehen. Die Schaltung besteht aus dem EEG/EKG/EMG Brick, dem MKR WiFi 1010, der auf dem vorgesehen *Brick* platziert ist, einem Steckbrett, der einen Schlater platziert hat, um den *Interrupt* auszulösen und einem Fingermodell. Ziel des Messaufbaus ist die Messung der Taskabläufe und deren Timings. Der Schalter und Fingermodell werden mit 5 V versorgt und besitzen jeweils eine Steuerleitung, die am MKR WiFi 1010 angeschlossen wird.

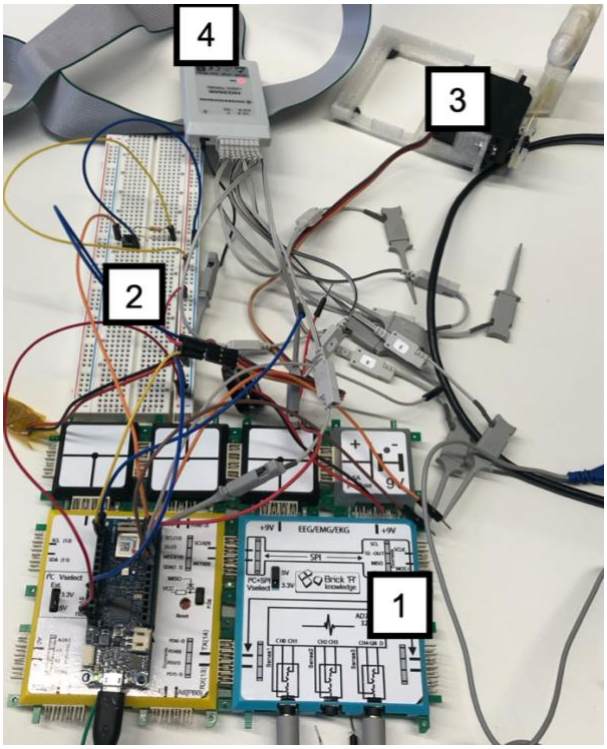

Abbildung 14: Messaufbau der EMG-Messung.

1. *Brick'R'knowledge-Set Aufbau* 2. *Steckbrett mit Button* 3. *Fingermodel*
4. *Digital Probe zur Messung*

Wie in Abbildung 15 dargestellt, werden zwei Elektroden am Arm (*M. Brachioradialis)* in longitudinaler Ausrichtung mit den Muskelfasern platziert. Eine Erdelektrode wird auf der Handrückseite (dorsal) platziert. Es werden Kendall H985G EKG-Elektroden verwendet.

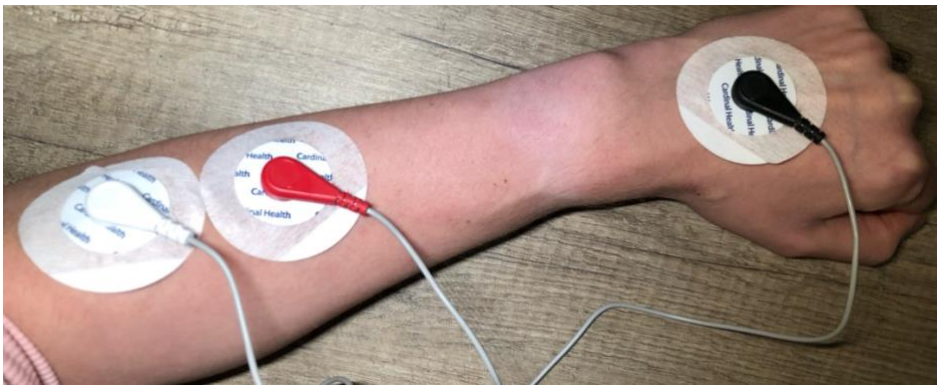

Abbildung 15: Platzierung der EMG-Elektroden.

Die Messung erfolgt alle 20 ms. Bei Muskelkontraktion soll das Fingermodell sich schließen. Es bleibt im geschlossenen Zustand, solange die Kontraktion hoch bleibt. Der PWM-*Task* wartet 300 ms auf die *Queue*, falls diese in diesem Zeitraum keinen neuen Wert erhalten sollte wird der Finger wieder geschlossen und das PWM-Signal wird abgeschaltet.

Ein LED-Task läuft im Hintergrund und bringt die LED auf dem Board zum blinken. Im eingeschalteten Modus soll sie jede Sekunde blinken und beim ausgeschalteten Modus alle 100 ms. Beim Drücken des Buttons wird eine ISR ausgelöst, die ISR übergibt dem *Handler-Task* eine Semaphore übergibt. Dadurch wird der *Task* ausgeführt und die EMG- und PWN-*Tasks* werden deaktiviert, falls sie im Betrieb sind und werden aktiviert, falls sie davor inaktiv waren. Der LED-*Task* wird auf den Takt von entweder 1 Sekunden oder 100 ms Takt gestellt.

In der Abbildung 16 ist der Ablauf einer Messung zu sehen. Der EMG-*Task* wird alle 20 Sekunden ausgeführt. Wird der Schwellenwert überschritten, wird der gemessene Wert in eine *Queue* gegeben, die an den PWM-*Task* weitergegeben wird. Dieser gibt ein Pulsweitenmodudel-Signal (PWM-Signal) auf den Finger, um diesen zu schließen. Der PWM-*Task* wartet bis zu 300 ms auf einen neuen Wert in der *Queue*. Wenn der Schwellenwert bis dahin nicht überschritten wurde, wird das Fingermodell durch ein weiteres PWM-Signal wieder geöffnet und ausgeschaltet. Der LED-*Task* setzt jede Sekunde die interne LED des Boards auf *HIGH* und *LOW*. Die in Abbildung 16 und Abbildung 17 dargestellen *Task*-Abläufe entsprechen nicht der tatsächlichen zeitlichen Skalierung. In der Realität wird der EMG-*Task* mehrmals ausgeführt, bevor der LED-*Task* sich wiederholt.

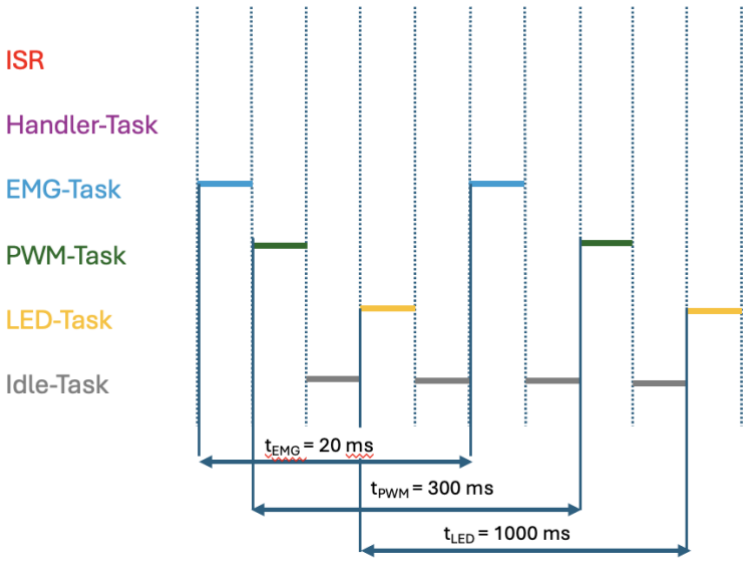

Abbildung 16: Timing- und Ablauf-Diagramm der Tasks.

In Abbildung 17 wird gezeigt, wie der EMG- und PWM-*Task* deaktiviert werden. Nachdem die ISR durch einen Button betätigt wurde, aktiviert dieser den *Handler-Task*, welcher wiederum den EMG- und PWM-*Task* deaktiviert. Die Periode des LED-*Tasks* wird auf 100 ms verkürzt.

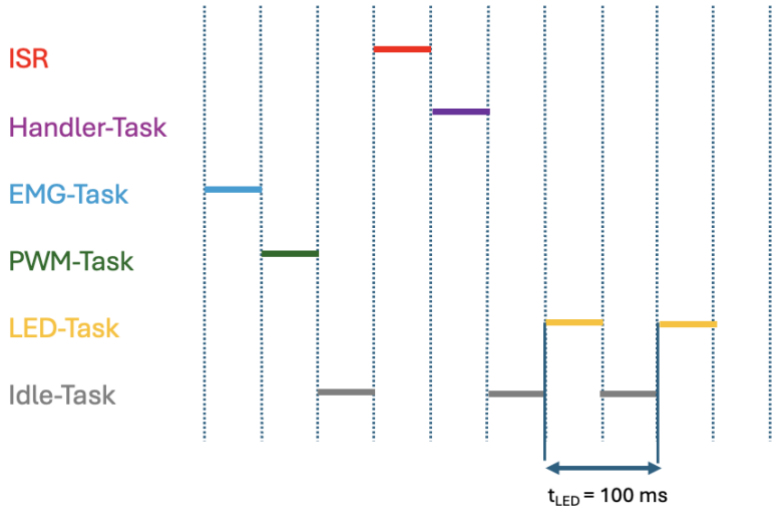

Abbildung 17: Timing- und Ablauf-Diagramm der Tasks nach ISR.

Um die *Task*-Zeiten messen zu können, werden einzelne Pins des MKR WiFi 1010 beim Start der Ausführung auf *HIGH* und nach Ende der Ausführung wieder auf *LOW* gesetzt. So können sowohl die Ausführungs- als auch die Periodenlänge der *Tasks* gemessen werden. Die interne LED des Boards ist parallel zu einem Pin geschaltet, weshalb dieser direkt gemessen werden kann. Dadurch wird nicht die Verarbeitungsdauer des *Tasks* gemessen, sondern nur die Periodenlänge, die dieser für den gesamten *Delay* entweder auf *LOW* oder *HIGH* gesetzt ist. Die Ausgänge werden mit einer *Logic-Probe* und einem Oszilloskop (Rohde & Schwarz HMO722) gemessen. Die Logic Probe wird an den entsprechenden Pins angeschlossen.

3 Ergebnisse

Im Folgenden werden die Ergebnisse der durchgeführten Testprogramme und Messungen zusammengefasst. Diese umfassen die Analyse des Speicher- und *Task*-Verhalten des MKR Wifi 1010 unter Last anhand zweier spezifischer Testprogramme, sowie die Ergebnisse der Messungen des Prototypens und Vollaufbaus inklusive der Steuerung des Fingermodells.

3.1 Messungen der Testprogramme

In der Implementierung der Testprogramme konnte festgestellt werden, dass maximal 20 Tasks erstellt werden konnten, bevor der *Heap*-Speicher des Mikrocontrollers erschöpft war. Die *Heap*-Größe wurde in der Konfigurationsdatei (FreeRTOSConfig.h) auf 14,336 KB festgesetzt. Der *Stack*-Speicher, der beim Erstellen der Task festgelegt wird, beträgt 128 *Words*. Ein *Word* auf einem 32-Bit-Prozessor ist 4 Bytes groß, weshalb ein Speicherbedarf von 512 Bytes vorliegt. Der TCB wird automatisch allokiert. Der genaue Speicherbedarf konnte durch Analyse mit der Funktion *xPortGetFreeHeapSize*() ermittelt werden, dass eine Task einen Gesamtspeicher von 624 Bytes verbraucht. Die Größe des Stacks kann berechnet werden, indem die Differenz zwischen dem Gesamtspeicher, den ein Task verbraucht und dem Speicher der für eine Task explizit allokiert wurde genommen wird. Somit wäre der *Overhead* des TCBs 112 Bytes groß.

Formel 2: Berechnung des TCBs eines Tasks

$$TCBStack = Gesamtspeicher - Taskspeicher$$

$$TCBStack = 624\ Bytes - 512\ Bytes = 112\ Bytes$$

Die Ausgabe des freien *Heaps* nach jeder Task Erstellung wird in Abbildung 18 dargestellt. Nach Erstellung des ersten *Tasks* wird der stehen 13.704 Bytes des *Heaps* zur Verfügung. In Inkrementen nimmt der Speicher pro *Task* um 624 Bytes ab. Nachdem der Scheduler gestartet wird, werden weitere 1.640 Bytes verwendet. Die *xPortGetFreeHeapSize()* gibt erst einen Rückgabewert, wenn Speicherplatz zugewiesen wurde, weshalb der freie *Heap* vor der Erstellung einer *Task* nicht ausgegeben wurde. Durch Rückrechnung kann jedoch darauf geschlossen werden, dass der freie Speicher wahrscheinlich bei 14.328 Bytes liegt. Dies sind 8 Bytes weniger als der zugewiesene Speicher von 14.336 Bytes. Siehe Anhang: Projekt “FreeRTOS_Stresstest_1” für den Quellcode.

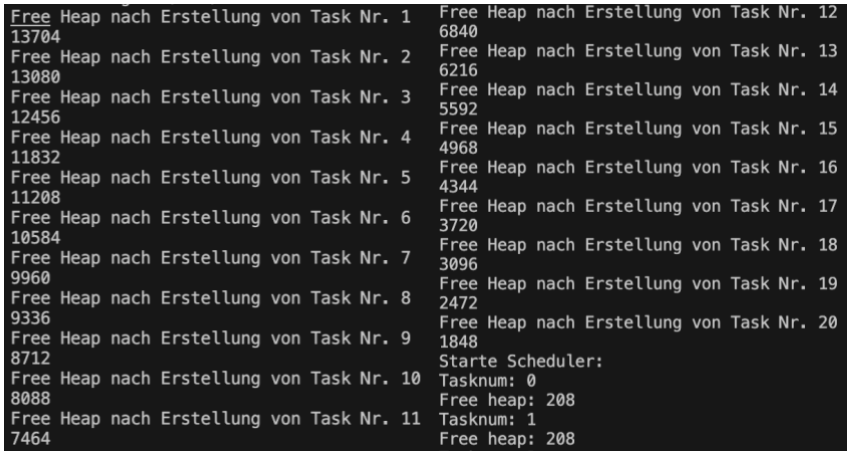

Abbildung 18: Ausgabe des freien Heaps nach Erstellung der Tasks.

Im zweiten Testprogramm konnte festgestellt werden, dass fünf Tasks ausgeführt werden konnten, bevor das Limit erreicht wurde. Die Delay ist auf 500 ms eingestellt. Ein *Task* benötigt ca. 4.456.848 bis 4.459.584 Zyklen zur Berechnung. Die Tickzahl pro *Task* liegt bei 92-93 Ticks. Der MKR WiFi 1010 arbeitet mit 48 MHz, was genau 24 Millionen Zyklen alle 500 ms sind. Fünf *Tasks* durchlaufen ca. 22,29 Millionen Zyklen, was in die *Task*-Frequenz passt.

Ein sechster *Task* konnte alle zwei Zyklen ausgegeben werden, also nachdem alle Tasks, die ausgeführt werden konnten jeweils einmal ausgegeben wurden. Dies konnte er indem die Rechenzeit, die im letzten Zyklus übrig geblieben ist genutzt hat, um die Berechnung zum Teil durchzuführen. Dadurch ergibt sich eine Zyklenanzahl von ca. 49.077.936 für den sechsten Task. Siehe Anhang: Projekt “FreeRTOS_Stresstest_2” für den Quellcode.

Abbildung 19: Konsolenausgabe Testprogramm 2. Pin-Ausgabe, Tickcount und Cyclecount.

3.2 Steuerung des Aktuators per EMG-Messung

Zunächst wurde der Prototyp gemessen. Das Projekt beinhaltet vier *Tasks* und einen *Interrupt*. Der *Measure*-Task führt alle 50 ms eine Messung durch. Dieser Wert wird in eine *Queue* gesendet, die vom PWM-*Task* gelesen wird. Auf Basis des übergebenen Wertes wird ein entsprechendes PWM-Signal generiert, welches wie in Quellcode 12 dargestellt durch eine *map()*-Funktion realisiert wird. Diese generiert ein PWM-Signal proportional zum Wert, den die Funktion erhält.

Quellcode 12: map()-Funktion zur Generierung eines PWM-Signals

```
pwmValue = map(receivedValue, 0, 1023, 0, 255);
analogWrite(PWM_PIN, pwmValue);
```

Für die Messung wurde ein Gleichspannungssignal angelegt. Der LED-Task setzt die interne LED auf dem Board abwechselnd auf *HIGH* und *LOW*. Die Frequenz hängt davon ab abhängig in welchem Zustand sich das Programm befindet. Im *ON*-Modus blinkt die LED einmal pro Sekunde und im *OFF*-Modus alle 100 ms. Um die ISR so kurz wie möglich zu halten, wird ein *Handler-Task* verwendet, der im Falle eines *Interrupt* entweder den *Measure*- und PWM-*Task* aktiviert oder deaktiviert. Der Handler-Task erhält eine Semaphore von der ISR, wodurch er aktiviert wird. Die ISR wird durch eine fallende Flanke an einem Pin ausgelöst. Dies wird durch einen Button realisiert.

Die Messung des Prototyps hat ergeben, dass das Programm mit vier implementierten Tasks Echtzeitfähigkeit besitzt. Der Interrupt konnte erfolgreich betätigt werden. Dieser hat den *Task* für das Messen deaktivieren bzw. aktivieren. Der verwendete Knopf hat geprellt, weshalb eine einsekündige Wartezeit bis zum nächsten Knopfdruck integriert wurde. Das analoge Signal konnte gemessen werden und wurde an den PWM-Task über eine *Queue* übergeben, um auf Basis des gemessenen Wertes ein PWM-Signal zu generieren. Dieses Signal wurde an einem PWM-fähigen Pin ausgegeben. Die Zeiten der *Task*-Abarbeitung, *Task*-Abstände und *Task*-reaktionszeiten wurden über ein Oszilloskop gemessen.

Tabelle 1: Timing-Messungen des Prototypen.

	Messung 1	**Messung 2**	**Messung 3**
Reaktionszeit *Button*-Klick bis Start der ISR	2,88 ms	2,88 ms	2,88 ms
ISR-Länge	8,84 ms	8,84 ms	8,84 ms
Reaktionszeit ISR bis zum *Handler-Task* (*ON-Modus* zu *OFF-Modus*)	11,5 ms	11,5 ms	11,5 ms
Reaktionszeit ISR bis zum *Handler-Task* (*OFF-Modus* zu *ON-Modus)*	20,30 ms	20,30 ms	20,30 ms
Abstand ISR bis zur erster Messung	49,4 ms	49,8 ms	49 ms
***Measure-Task*-Länge**	848 ms	848 ms	848 ms
***Measure-Task* zu *PWM-Task* Abstand**	30,9 ms	30,9 ms	30,9 ms
***Handler*-Task Länge**	1,7 ms	1,7 ms	1,7 ms
PWM-Task Länge	2.66 ms	2.66 ms	2.68 ms
LED -Task Periode *ON*-Modus	1 s	1 s	1 s
LED-Task Periode *OFF*-Modus	100 ms	100 ms	100 ms
Abstand ISR LED (*ON*-Modus zu *OFF*-Modus)	332 ms	176 ms	668 ms
Abstand ISR LED (*OFF*-Modus zu *ON*-Modus)	60 ms	44 ms	16 ms
***Measure*-Task Periode**	50 ms	50 ms	50 ms

Es wurden jeweils drei Messungen durchgeführt. Die zeitlichen Delays der *Tasks* wurden immer eingehalten. In dem Fall, dass ein *Interrupt* ausgelöst wird, befindet sich die LED-*Task* noch im blockierten Zustand des vorherigen Zustandes und kann diesen erst bei der nächsten Ausführung ändern. Das heißt, wenn der LED-*Task* im 1-Sekunden-Takt blinkt, die LED eingeschaltet ist und nach innerhalb des Zyklus 668 ms ein *Interrupt* getätigt, dass die LED noch 332 ms blockiert ist, bis diese in den 100 ms Rhythmus übergehen kann. Siehe Anhang: Projekt “FreeRTOS_Stresstest_3.2” für den Quellcode.

Im Projekt für den Aufbau mit dem Fingermodell wurden ähnliche *Tasks* wie beim Prototypen realisiert. Die Messung erfolgt über den EEG/EMG/EKG-*Brick.* Die Delay-Zeit vom EMG-*Task* wurde auf 20 ms reduziert. Dieser gibt wieder ein Wert in die *Queue*, aber nur falls ein Schwellenwert überschritten wird. Das PWM-Signal wurde modifiziert, um die gewünschte Auslenkung des Servomotors zu generieren. Die Auslenkung ist auch unterschiedlich groß je nach Größe des übergebenden Signals an die *Queue*. Nachdem ein Wert in die *Queue* gegeben wird, wird das PWM-Signal zum Schließen des Fingers generiert. Falls innerhalb von 300 ms kein Wert in die *Queue* gegeben wird, soll der Finger wieder geöffnet werden und anschließend das PWM-Signal auf null gesetzt werden. Die interne LED des Boards wird wieder in den selben Abständen abwechselnd auf HIGH und LOW gesetzt. Und ein *Handler-Task* wird aktiviert, nachdem ein Interrupt ausgelöst wurde.

In Abbildung 20 ist der generelle Verlauf des Programms dargestellt. Zum Start werden Variablen und der EMG-*Brick* Initialisiert. Die Initialisierung des EMG-*Bricks* läuft über das I^2C Interface und braucht in der Regel mehrere Anläufe für die Initialisierung. Der AD-Wandler wird über das SPI-*Interface* eingestellt. Danach läuft ein *Setup* für das PWM-Signal [25], die *Tasks* werden erstellt und der *Scheduler* gestartet. Danach übernimmt der *Taskscheduler*. Der EMG- und LED-*Task* laufen Periodenbasiert, während Handler- und PWM-*Task* ereignisgesteuert ausgeführt werden. Durch Drücken des *Buttons* wird die ISR ausgelöst, die den *Handler* aktiviert und den Status der *Tasks* entsprechend ändert.

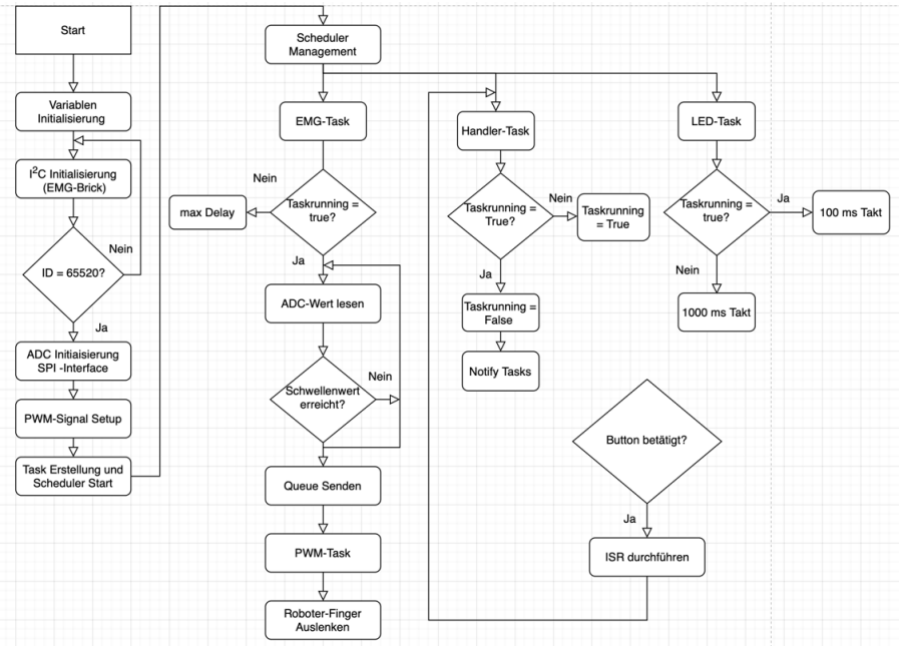

Abbildung 20: Flowchart Diagramm des Programmablaufes

Die Reaktionszeit des Systems auf ein EMG-Signal wurde überprüft. Dazu wurde das EMG-Signal über einen Verstärker (INA114AP) gemessen. In Abbildung 21 ist der Messaufbau zu sehen. An den Eingängen des Verstärkers wurde das differentielle EMG-Signal aufgenommen. Hierzu wurde eine Schaltboard verwendet, auf dem der Verstärker gesetzt wurde.

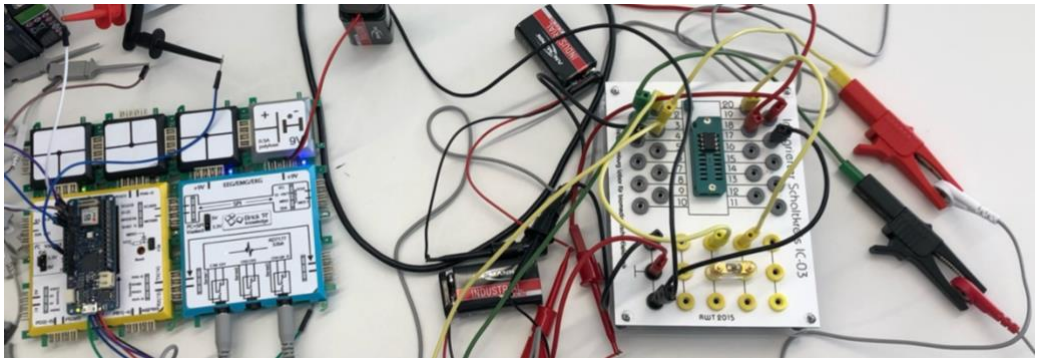

Abbildung 21: Messungsaufbau zur Ermittlung der Reaktionszeit vom EMG-Signal bis zum PWM-Signal.

In Abbildung 22 ist die Messung dargestellt. CH1 zeigt das EMG-Signal, CH2 das erzeugte PWM-Signal. Mit einer Digital-Probe wurden die Messabstände erfasst.

Wird eine ausreichende Differenz zwischen zwei Messungen festgestellt, wird das PWM-Signal ausgelöst. Die Zeit zwischen Messung und PWM-Signal beträgt 23,6 ms. Diese Zeit zeigte nach mehreren Messungen eine geringe Variation.

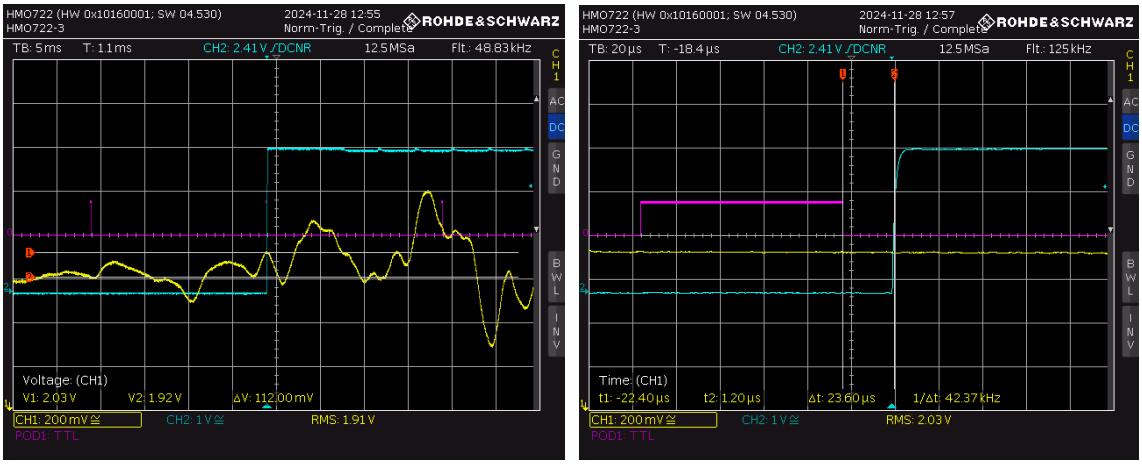

Abbildung 22: EMG-Signal (CH1), PWM-Signal (CH2), EMG-Task (Kanal 0), Messabstände und Reaktionszeit.

Im Rahmen der Vollaufbaumessung wurden die festgelegten *Timings* im Programm überprüft. Diese waren beim EMG-*Task* alle 20 ms, beim PWM-*Task* 300 ms nachdem kein Wert mehr in die *Queue* geschickt wurde und beim LED-Task 1000 ms im *ON*-Zustand und 100 ms *OFF*-Zustand. Zusätzlich wurden die Zeiten gemessen, die jeder *Task* für die Verarbeitung benötigt. Hierfür wurden einzelne Pins des Boards beim Start und Ende der Ausführung angesteuert. Diese sind in Tabelle 2 aufgeführt.

Tabelle 2: Kanal- und Pinlayout der Tasks und des Buttons.

Kanal-Nummer	Zugeordneter Task/Pin
Kanal 0	EMG Task (Pin 0)
Kanal 1	PWM Signal (Pin 1)
Kanal 2	ISR Signal (Pin 3)
Kanal 3	Handler Task (Pin 4)
Kanal 4	LED Task (Pin 6)
Kanal 5	Button Signal (Pin 5)

In Abbildung 23 ist die Reaktion des Systems auf einen Knopfdruck zu sehen (Kanal 5). Die ISR (Kanal 2) wird ausgelöst, woraufhin der der *Handler-Task* (Kanal 3) ausgeführt wird. Dieser schaltet den EMG- und PWM-*Task* wieder ein und ändert den Zustand des LED-*Tasks* (Kanal 4).

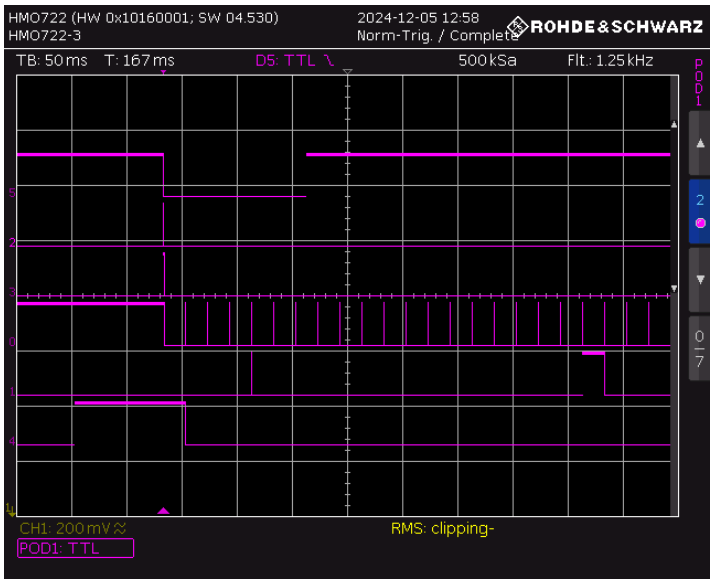

Abbildung 23 Task Ablauf nach Interrupt-Auslösung.

Kanal 5: Button *Kanal 2: ISR* *Kanal 3: Handler-Task* *Kanal 0: EMG-Task*
Kanal 1: PWM-Task *Kanal 4: LED*

Der Ausgang des PWM-*Tasks* wurde verändert, sodass nicht mehr die Bearbeitungsdauer des *Tasks* angezeigt wird, sondern das Umzuschalten erfolgt, sobald die Wartezeit von 300 ms überschritten wurde. Diese Wartezeit ist in Abbildung 24 dargestellt. Zudem werden dadurch auch 15 Zyklen des EMG-*Tasks* dargestellt, die auch nach exakt 300 ms abgeschlossen sind.

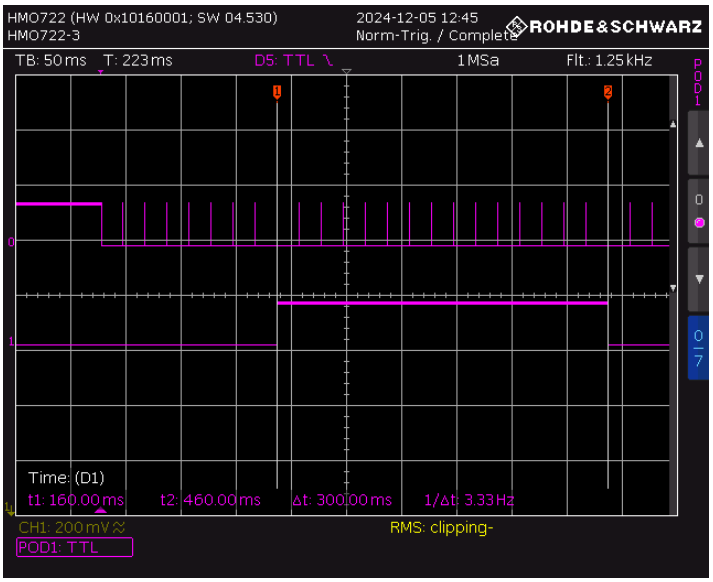

Abbildung 24: PWM-Signal Wartezeit und 10 Perioden der Messung.

Kanal 0: EMG-Task *Kanal 1: PWM-Task*

Tabelle 3 zeigt die Messungen der *Tasks*. Es wurde überprüft, ob die periodischen *Tasks* die festlegten Zeiten einhalten. Außerdem wurde die Ausführungsdauer der einzelnen *Tasks* und der ISR gemessen. Auch die Reaktionszeiten zwischen dem Drücken des *Buttons* und dem Start der ISR und die Zeiten zwischen der Ausführungen einzelner *Tasks* wurden gemessen.

Die festgelegten Timings wurden in allen Messungen eingehalten. Der EMG-*Task* führt alle 20 ms eine Messung durch. Der LED-Task blinkt je nach Zustand alle 100 ms oder jede Sekunde und der PWM-*Task* wartet 300 ms auf einen neuen Wert in der *Queue*. Die einzelnen Bearbeitungszeiten sind in den Messungen auch gleich lang gewesen. Es gab nur in der Zeit zwischen EMG- und PWM-*Task* eine Variation der Länge. Die kürzeste Reaktionszeit war 22 ms lang und die längste 67,6 ms. Ein ähnlicher Befund wurde auch bei der direkten Reaktionsmessung des EMG-Signals festgestellt. Siehe Anhang: Projekt “FreeRTOS_Finger_Test” für den Quellcode.

Tabelle 3: Timing-Messung des Vollaufbau

	Messung 1	**Messung 2**	**Messung 3**
Reaktionszeit *Button*-Klick bis Start der ISR	2,84 ms	2,84 ms	2,84 ms
ISR-Länge (*ON-Modus* zu *OFF-Modus*)	8,84 ms	8,84 ms	8,84 ms
ISR-Länge (*OFF-Modus* zu *ON-Modus*)	8,84 ms	8,84 ms	8,84 ms
Reaktionszeit ISR bis zum *Handler-Task* (*ON-Modus* zu *OFF-Modus*)	11,30 ms	11,30 ms	11,30 ms
Reaktionszeit ISR bis zum *Handler-Task* (*OFF-Modus* zu *ON-Modus)*	11,30 ms	11,30 ms	11,30 ms
EMG-*Task* Periode	20 ms	20 ms	20 ms
EMG-*Task* Ausführungsdauer	94 ms	84 ms	84 ms
Reaktionszeit EMG-*Task* bis PWM-*Task*	22 ms	60 ms	67,6 ms
Handler-*Task* Ausführungsdauer	1,61 ms	1,61 ms	1,61 ms
PWM-*Task* Ausführungsdauer	1,61 ms	1,61 ms	1,61 ms
PWM-*Task Queue* Wartezeit	300 ms	300 ms	300 ms
LED-*Task* Periode *OFF*-Modus	100 ms	100 ms	100 ms
LED-*Task* Periode *ON*-Modus	1 s	1 s	1 s

Es wurde außerdem geprüft, wie feinfühlig das System auf EMG-Signale reagieren kann und das Fingermodell entsprechend ansteuert. Die Empfindlichkeit wurde so eingestellt, so dass der Finger auf eine Fingerbewegung reagieren kann. Dieser Bewegung wurde die kleinste Auslenkung von 14% zugeordnet. Der Auslenkungsverlauf ist in Abbildung 25 dargestellt. Der Auslenkung wird von links nach rechts größer. Der Finger ist bei einer Auslenkung von 13% vollständig geöffnet. Weitere Unterstufungen konnten ebenfalls realisiert werden, sodass ein Gradient von 13-18% der Auslenkung des Servomotors mittels EMG-Messung möglich ist. Die Platzierung der Elektroden spielt dabei eine Rolle, wie gut das System reagieren kann. Diese Werte sind im Code vordefiniert und das System würde an Muskelpartien mit größeren Auslenkungen reagieren, die eine größere Potentialdifferenz vorweisen oder mit kleineren Auslenkungen dort wo die Potentialdifferenz kleiner ist.

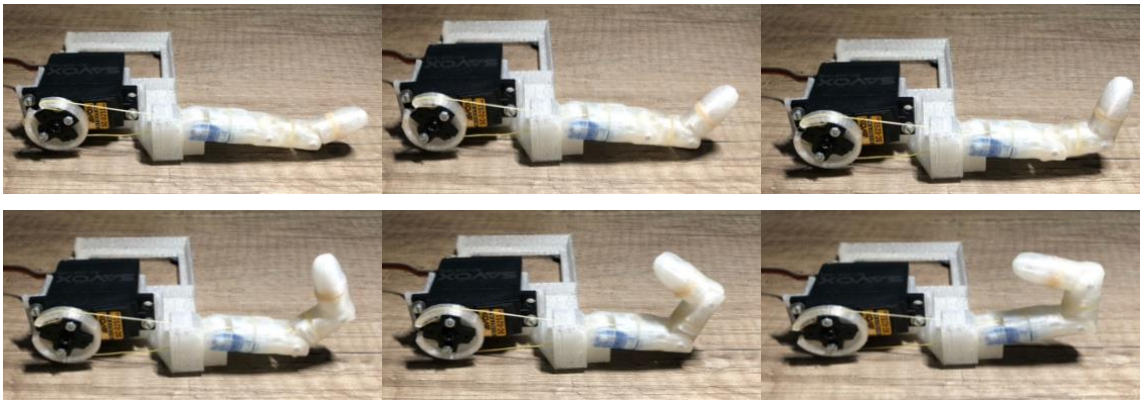

Abbildung 25: Fingermodel Auslenkungsgradient von 13-18%

Für die Bedingung, wann das PWM-Signal generiert werden soll, wurden zwei unterschiedlich Methoden verwendet. In der ersten Methode (Quellcode 13) wurde eine Differenz zwischen zwei Messungen berechnet und das Signal wurde generiert, wenn diese Differenz einen festgelegten Grenzwert überschritt.

Quellcode 13: Erste Mehtode zur Auslösung des EMG-Signals.

```
v1 = values[0] - 0x80000000l;
delta = delta - v1;
sensorValue = delta;
if (abs(delta) > 20000)
{
xQueueSend(sensorQueue, &sensorValue, 10);
}
```

Der Absolutwert des gemessenen Wertes wurde genommen, um auch bei negativen Spannungswerten ausgelöst werden zu können. Der Servomotor konnte somit je nach stärke der Muskelkontraktion mit verschiedenen Öffnungsgraden gesteuert werden. Der *Duty-Cycle* des PWM-Signals lag dabei zwischen 14% und 18%. Die Kontraktionsstärke ist propotional zum *Duty-Cycle*.

In der zweiten Methode wurde der Absolutwert der Spannung genommen, um einen Schwellewert für das PWM-Signal zu definieren. In Quellcode 14 ist der Ansatz dargestellt. Da der Spannungswert einen leichten *Drift* und *Offset* aufwies, wurde zur Ermittlung eines verlässlichen Schwellenwertes eine Mittellung mehrerer Messungen Messungen durchgeführt.

Quellcode 14: Zweite Methode zur Auslösung des PWM-Signals

```
v1 = values[0] - 0x80000000l;
voltage1 = ((float)v1 / (float)(0x80000000) * VREF) * 1000;
offset -= offsetBuffer[filterIndex] / filterSize;
offsetBuffer[filterIndex] = voltage1;
offset += voltage1 / filterSize;
filterIndex = (filterIndex + 1) % filterSize;
delta = voltage1 - offset;
```

Der binäre Wert, wird im Millivolt-Bereich skaliert und aus den zehn aktuellsten Werten wird ein gleitender Mittelwert gebildet. Der Wert wird ebenfalls mit einem vordefineirten Wert verglichen und auf Basis dessen wird das Fingermodell in unterscheidlichen Stärken geschlossen.

Beide Methoden konnten eine gezielte Auslenkung des Fingermodels erzielen, allerdings konnte insgesamt eine präzisere Ansteuerung mit der Methode der Differenzbildung zwischen zwei Werten eine präzisere Ansteuerung des Servomotors erzielt hat. Siehe Anhang: Projekt “FreeRTOS_Roboterfinger_Voltagethreshold” für den Quellcode.

4 Diskussion

Das Ziel dieser Arbeit war die Implementieren von FreeRTOS auf dem SAMD21 sowie die Steuerung eines Fingermodells mithilfe eines EMG-Signals. Die Steuerung eines Fingermodells konnte erfolgreich realisiert werden und die Implementierung konnte mithilfe eines bestehenden Ports umgesetzt werden. Die Entwicklung eines eigens kreierten Ports konnte nicht erfüllt werden.

4.1 Bewertung der Ergebnisse

Das Abschlussprojekt konnte alle vorgegebenen *Timings* einhalten. Nach Bereinigungen von einigen Fehlern, die zum Aufhängen des Programms geführt haben, konnte das Programm verwendet werden ohne abzustürzen. FreeRTOS erwies sich als geeignet um mit dem MKR WiFi 1010 Echtzeitanwendungen zu entwickeln. Im Betrieb mit dem Fingermodell wurden die festgelegten *Delay*-Zeiten eingehalten und verschiedene Funktionalitäten, wie *Tasks*, *Queues* und *Semaphoren* konnten erfolgreich eingesetzt werden.

Im ersten Stresstest zur Überprüfung der Speicherverwaltung wurde festgestellt, wie viel Speicher von *Tasks* und der *Scheduler* beansprucht wird. Außerdem konnte aus den Ergebnissen die Größe des TCBs einer Task extrapoliert werden. Der gesamte freie Speicher ist scheinbar um 8 Bytes geringer als ursprünglich festgelegt. Dies könnte auf die interne Speicherverwaltung von FreeRTOS sein zurückzuführen sein. Die *xPortGetFreeHeapSize()*-Funktion kann nur verwendet werden, wenn durch eine andere Funktion Speicher zugewiesen wurde. Das FreeRTOS-Handbuch liefert jedoch keine Erklärung für dieses Verhalten. Der *Scheduler* selbst hat 1640 Bytes beansprucht. Es gibt die Möglichkeit, dass FreeRTOS beim Start des *Schedulers* einen *Timer*- und *Idle-Task* erstellt, diese beanspruchen eine minimale *Stack*-Größe, welche in der Konfigurations-Datei auf 600 Bytes festgelegt ist. Zusammen mit der internen Datenstruktur des *Schedulers* könnte dies die gemessene Speichergröße erklären Größe erklären.

Im zweiten Stresstest, der das Verhalten des Systems unter Volllast untersuchen sollte, konnte festgestellt werden, dass nicht alle *Tasks* ausgeführt wurden. Dies entspricht den Erwartungen an das Verhaltens von FreeRTOS, da der *Scheduler Tasks* höherer Priorität bevorzugt und *Tasks* niedrigerer Priorität eingeschränkt oder gar nicht ausführt, wenn die Systemlast zu hoch ist.

Die Ergebnisse zeigen, dass fünf *Tasks* vollständig innerhalb der vorgegebenen 500-ms-Periode ausgeführt werden konnten. Die durchschnittliche Anzahl der Zyklen lag bei ca.

4.456.848 bis 4.459.584, was eine Konstanz in der Taskausführungszeit belegt. Die Tickzahl von 92 bis 93 Ticks bestätigt ebenfalls, dass das System stabil arbeitet. Der sechste *Task* hingegen konnte nur nach mehreren *Task*-Zyklen ausgegeben werden. Seine Zyklenzahl von 49.077.936 deutet darauf hin, dass er erst nach Abschluss von priorisierteren *Tasks* vom *Scheduler* berücksichtigt wurde. Dies geschah durch Nutzung der verbleibenden Rechenzeit, bevor der erste *Task* mit der höchsten Priorität erneut ausgegeben werden musste. Diese Beobachtungen zeigen, dass der *Scheduler* flexibel mit Restkapazitäten umgehen kann und auch *Tasks* mit niedriger Priorität nach längeren Zyklen ausgeführt werden können.

In den Tests wurden einige der Hardwarelimits in Kombination mit FreeRTOS deutlich. Um Synchronität und die Einhaltung von Timings zu gewährleisten, ist es der Realisierung von Tasks vorteilhaft, dass diese keine zu lange Ausführungszeiten kurz zu halten.

Die Messergebnisse des Prototypen und des Vollaufbaus zeigen, dass das System unter den gegebenen Bedingungen robust und echtzeitfähig arbeitet. Im Prototypen wurden die wesentlichen Funktionen erfolgreich implementiert und getestet. Die Timings der periodischen *Tasks* wurden eingehalten, sowie die korrekte Ausführung der eventbasierten *Tasks* nach Interrupt-Auslösung. Die gemessenen Reaktionszeiten und Ausführungsdauer der einzelnen *Tasks* belegen, dass die Priorisierung und Synchronisation der Tasks wie vorgesehen funktioniert.

Im Vollaufbau wurde das System erfolgreich für die Anwendung mit einem Fingermodell angepasst. Die Messungen zeigen, dass die EMG-Daten zuverlässig erfolgt und das PWM-Signal präzise auf die Muskelaktivität reagiert. Die gemessenen Timings weisen eine hohe Präzision bei der Ansteuerung des Modells auf. Das System reagiert schnell auf Muskelaktivitäten und kann durch die Erkennung feiner Unterschiede in der Muskelkontraktion verschiedene Auslenkungen des Servomotors bewirken.

4.2 Implementierung und Entwicklung

Für den SAMD21 existiert kein offizieller Port. Der Versuch, den Port des SAMD20, der zur gleichen Prozessorfamilie gehört, auf den SAMD21 zu integrieren, war nicht erfolgreich. Stattdessen wurde ein eines Drittanbieter-Port verwendet, der Änderungen im *Source*-Code beinhaltet. Dies birgt Unsicherheiten hinsichtlich der Fehlerfreiheit des Ports und der Kompatibilität mit zukünftigen FreeRTOS-Versionen. Es kann nicht ausgeschlossen werden, dass der Drittanbieter-Port reibungslos ohne *Bugs* funktioniert. Dadurch kann im speziellen

nicht sichergestellt werden, dass neuste Versionen von FreeRTOS auf dem Port übertragen werden können.

Der EEG/EMG/EKG-*Brick* des Brick'R'knowledge-Sets bereitete während der Entwicklung Probleme, da er häufig von der Software nicht erkannt wurde. Dadurch kam es während der Entwicklung der Programme zu Engpässen. Der Austausch des *Bricks* durch ein Modell aus einem anderen Set konnte das Problem beheben. Das Set soll in regulären Anwendung mit einer 9V-Batterie betrieben werden. Diese werden allerdings nach einigen Stunden Nutzung entladen, bis diese nicht mehr nutzbar sind und ausgewechselt werden müssen. Außerdem ist die 9V-Batterie der simultanen Nutzung des Vollaufbaus überlastet und kann somit für diese Anwendung nicht verwendet werden. Für die Realisierung wurde daher eine Powerbank verwendet, die nicht überlastet wird und wiederaufladbar ist, wodurch die Arbeit mit dem Kit ökonomischer wird. Eine Nutzung mit einem Funktionsgenerator war aus Sicherheitstechnischen Gründen nicht möglich.

In den Testprogrammen traten Synchronisationsprobleme auf, die dazu führten, dass Tasks gelegentllich mit falschen Prioritäten ausgeführt wurden. Dieses Problem konnte durch die Implementierung eines Befehls behoben werden, der sicherstellt, dass auf die Initialisierung der USB-Schnittstelle gewartet wird, wie in Quellcode 15 aufgezeigt.

Quellcode 15: Befehl zum Warten der USB-Schnittstelle

```
while(Serial!);
```

Außerdem zeigte die Betätigung des Buttons, dass der EMG-*Task* Ausgaben proportional zur Zeit tätigte, die seit seiner Deaktivierung vergangen war. Dieses Verhalten ist auf die Verwendung der Funktion *vTaskDelayUntil()* (Quellcode 16) im EMG-*Task* zurückzuführen. Diese Funktion erzeugt eine periodische Delay und sorgt dafür, dass die zeitliche Ausführung desTasks synchron zu einem festgelegten Startzeitpunkt bleibt.

Quellcode 16: Delay-Zeit des EMG-Tasks

```
vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(20));
```

Die Funktion *vTaskDelayUntil()* verwendet die Variable *xLastWakeTime*, die mit der Tickzahl initialisiert wird, die seit dem Start des *Schedulers* vergangen ist [6]. Diese Variable wird mit

dem *Tickcount* gleichgesetzt seitdem der *Taskscheduler* gestartet wurde. Im Vergleich zur *vTaskDelay()*-Funktion, berücksichtigt *vTaskDelayUntil()* die Ausführungszeit des vorherigen Codes und gewährleistet so eine konstante Periodizität. Beim Deaktivieren des EMG-*Tasks* durch den *Button* und dessen anschließender Reaktivierung trat folgendes Problem auf: Der EMG-*Task* wurde ohne *Delay* solange ausgeführt, bis dieser die vergangene Zeit zwischen Deaktivierung und Reaktivierung aufgeholt hat. Dadurch, dass die Tick-Anzahl in die Variable *xLastWakeTime* gespeichert wird, trat der Fehler auf, dass wenn der EMG-*Task* durch den Button wieder aktiviert wird, kein *Delay* ausgelöst, bis der *Task* die vergangene Zeit zwischen beiden Buttonbetätigungen aufgeholt hat. Beispielsweise wurden bei einer einsekündigen Deaktivierungszeit des *Tasks* 50 aufeinanderfolgende Iterationen ausgeführt, ohne einen neuen *Delay* auszulösen. Um dieses Problem zu beheben, wurde die Variable xLastWakeTime mit der aktuellen Tickzahl gleichgesetzt. Dies bewirkt, dass die Variable nicht mehr im Verzug ist und dass dadurch nach der Reaktivierung *Delay* ausgelöst wird

Die Ansteuerung des PWM-Signal durch Differenzbildung von zwei Messwerten bietet eine einfache und effektive Realiserung auf Programmebene. Spannungsverschiebungen während der Messung haben in der Methodik keine Auswirkung darauf, ob eine Auslenkung des Fingermodells erzielt wird. Bei langanhaltenden Muskelkontrakationen großer Muskelgruppen kann es verstärkt zu Fluktuationen im EMG-Signal kommen. Dies ist darauf zurückzuführen, dass bei gleichzeitiger Aktivierung mehrerer Motoreinheiten die Signale miteinander interagieren und so zu Fluktuationen im Endsignal führen können [26]. Probleme könnten jedoch durch unterschiedlichen Gegebenheitenheiten von Elektroden, Hautoberfläche und Elektrodenplatzierung sein. Eine Methode um unterschiedlichen Faktoren bei der Messung entgegenzuwirken, könnte eine Normierung des Signals beim Start des Programmes sein. Dort könnte der Schwellenwert je nach Signal skaliert werden. So könnten Signale mit hohen oder niedriegen Differenzen eine ähnliche Systemreaktion hervorrufen. Ein weiteres theoretisches Problem bei der Differenzbildung ist, dass bei hoher Kontraktion des Muskels die Differenz zwischen zwei Messwerten dennoch zu klein ist, um eine Auslenkung zu erzeugen und somit, das Fingermodel sich nicht schließt. Dies wäre unerwünschtes Verhalten der Schaltung. In der Praxis konnte bei anhaltender hoher Kontraktion dennoch in den meisten Fällen eine Auslenkung erzielt werden. Die Variabilität zwischen zwei Messungen ist hoch genug, um das Fingermodel in den meisten Fällen geschlossen zu halten.

Der EMG-*Task* könnte durch eine häufigere Ausführung eine genauere Messung ermöglichen. Aktuell wird der *Task* alle 20 ms, ausgeführt was einer Frequenz von 50 Hz entspricht. Die wichtigsten Frequenzanteile eines EMG-Signals liegen im Bereich zwischen 50-150 Hz, weshalb eine Messrate von 200 Hz bzw. 5 ms realisiert werden sollte. Allerdings konnte das System bei dieser hohen Messfrequent das Fingermodell nciht erfolgreich auslenken. Trotz

einer Bearbeitungszeit von ca. 84 ms des EMG-Tasks, war das System nicht in der Lage dazu mit einer höheren Frequenz zu arbeiten.

4.3 Ausblick

Um eine reibungslose Entwicklung mit dem SAMD21 in Verbindung mit FreeRTOS zu ermöglichen, sollte idealerweise ein Port von Grund auf erstellt werden, der mit neuen FreeRTOS-Source-Dateien reibungslos aktualisiert werden kann. FreeRTOS bietet eine generelle Anleitung für die Portierung auf neue Mikrocontroller, allerdings sind gute Vorkenntnisse für eine erfolgreiche Portierung Empfehlenswert. Die Port-Dateien für den SAMD21 sind vorhanden, jedoch gab es Probleme beim Kompilieren. Diese Fehlermeldungen konnten in Umfang dieser Arbeit nicht gelöst werden.

Der Aufbau könnte um weitere *Tasks* erweitert werden, um zusätzliche Funktionen zu integrieren. Darunter könnten Modi-Einstellungen gemacht werden, um beispielsweise die Empfindlichkeit des Motors zu ändern. Darunter wären Anwendungen möglich, die die Steuerung eines Roboterarms ermöglichen, der dazu in der Lage ist Objekte zu greifen. In diesem Zusammenhang könnte ein Ansatz verfolgt werden, bei dem sich die Hand zangenartig schließt, um Objekte zu halten, oder ein Arm der eine Rotationsachse besitzt. Diese Anwendungen könnten auch über eine EMG-Messung gesteuert werden.

Die entwickelte Software kann auch auf anderen Mikrocontrollern verwendet werden. Wird ein Mikrocontroller mit einem offiziellen Port besitzt verwendet, kann auch sichergestellt werden, dass dieser Port gründlich getestet wurde und keine Fehler enthält. PlatformIO bietet eine Vielzahl von Mikrocontrollern, für die Quellcode entwickelt werden kann. In der ini-Datei muss lediglich die Baudrate auf das entsprechende System angepasst werden, andernfalls sollte der Quelltext Systemübergreifend kompatibel sein.

Für die Anwendung könnten Methoden Entwickelt werden, welche eine präzisere Ansteuerung des Fingermodels ermöglichen. Die EMG-Messung könnte auch mithilfe einer anderen Muskelpartie ausgeweitet werden. Es könnten verbesserte Verstärkungsschaltungen aufgebaut werden, die eine optimierte EMG-Messung ermöglichen.

Es existieren derzeit keine standardisierten Methoden zur Steuerung von Robotik mittels EMG-Signalen [27]. Die Auswahl der Methode hängt von den Anforderungen an die Steuerungspräzision und Echtzeitfähigkeit ab. Eine häufig verwendete Methode ist unter anderem die, wie auch in der Arbeit verwendete, Schwellenwertmethode. Dort werden Amplituden- oder Frequenzgrenzen festgelegt, welche als Schwellenwerte dienen, um eine

Bewegung auszulösen. Diese Methode findet am häufigsten verwenden, da diese ressourcenschonend und einfach umsetzbar ist [27]. Darüber hinaus kommen zunehmend statistische und maschinelle Lernmethoden zum Einsatz. Dies ermöglicht Mustererkennung innerhalb des EMG-Signals, um Robotik-Komponenten steuern können [27,28].

In zukünftigen Projekten könnte untersucht werden, welche der in der Literatur beschriebenen Methoden eine optimierte Steuerung des Fingermodells oder ähnlicher Robotik-Anwendungen ermöglichen.

5 Literaturverzeichnis

[1] Gridling G, Weiss B. Introduction to Microcontrollers. 1.4. Vienna University of Technology; 2007.

[2] Arduino S.r.l. Arduino® UNO R3 Product Reference Manual 2024.

[3] Vasudhendra Badami. A tour of the Arduino UNO board. Hackerearth 2016. https://www.hackerearth.com/blog/developers/a-tour-of-the-arduino-uno-board/ (accessed January 13, 2025).

[4] Kamal Raj. Microcontrollers : architecture, programming, interfacing and system design. International Journal of Soft Computing and Engineering (IJSCE) 2012;Volume-2:861.

[5] Dhaker P. Introduction to SPI Interface. Analog Dialogue 2018;52:1–5.

[6] Barry R. Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide Release Version - 1.1.0. Amazon.com, Inc; 2023.

[7] Gash MC, Kandle PF, Murray I V., Varacallo MA. Physiology, Muscle Contraction 2023.

[8] Ahmad I, Dhanbad S, Dey UK. A Review of EMG recording technique. International Journal of Engineering Science and Technology (IJEST) 2012;Vol. 4:530–9.

[9] Day S. Important Factors in Surface EMG Measurement. Bortec Biomedical Ltd Publishers 2002:1–17.

[10] Wang J, Tang L, E Bronlund J. Surface EMG Signal Amplification and Filtering. Int J Comput Appl 2013;82:15–22. https://doi.org/10.5120/14079-2073.

[11] Analog Devices. AD7177-2 32-Bit, 10 kSPS, Sigma-Delta ADC with 100 µs Settling and True Rail-to-Rail Buffers 2016:60.

[12] Appold F. Elektrophysiologische Messungen im Physikunterricht-EKG, EMG, EOG Schriftliche Hausarbeit für die erste Staatsprüfung für ein Lehramt an Gymnasien. Julius-Maximilians-Universität Würzburg, 2011.

[13] Zahak M. Signal Acquisition Using Surface EMG and Circuit Design Considerations for Robotic Prosthesis. Computational Intelligence in Electromyography Analysis - A Perspective on Current Applications and Future Challenges, InTech; 2012. https://doi.org/10.5772/52556.

[14] Konrad P. EMG-FIBEL Eine praxisorientierte Einführung in die kinesiologische Elektromyographie Noraxon INC. USA. Version 1.1. Noraxon INC, USA; 2011.

[15] S. Lee JK. Biopotential Electrode Sensors in ECG/EEG/EMG Systems. Analog Devices 2008:1–2.

[16] Brick'R'knowledge. Impressum. Brick'R'knowledge 2025. https://www.brickrknowledge.com (accessed January 13, 2025).

[17] Arduino S.r.l. Arduino® MKR WiFi 1010 Product Reference Manual. Arduino S.r.l.; 2025.

[18] ALLNET® GmbH Computersysteme. Brick'R'knowledge Bio Feedback Set Anleitung Rev. 1.0. Rev. 10.0. Germering: ALLNET® GmbH Computersysteme; 2019.

[19] Arduino S.r.l. MKR WiFi 1010 Pinout. Arduino 2020:1–3. https://docs.arduino.cc/resources/pinouts/ABX00023-full-pinout.pdf (accessed January 13, 2025).

[20] Kravets I. What is PlatformIO? GitHub 2023. https://github.com/platformio/platformio-docs/blob/develop/what-is-platformio.rst (accessed January 14, 2025).

[21] FreeRTOS. Supported Devices. Https://WwwFreertosOrg/Documentation/02-Kernel/03-Supported-Devices/00-Supported-Devices 2025. https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/00-Supported-devices (accessed January 13, 2025).

[22] José Bagur. Debugging SAM-Based Arduino® Boards with Atmel-ICE. Arduino 2024. https://docs.arduino.cc/tutorials/mkr-wifi-1010/atmel-ice/ (accessed January 13, 2025).

[23] FreeRTOS. Source Organization. FreeRTOS 2025. https://www.freertos.org/Documentation/02-Kernel/06-Coding-guidelines/01-Source-code-organization (accessed January 13, 2025).

[24] BriscoeTech. Arduino-FreeRTOS-SAMD21. GitHub 2019. https://github.com/BriscoeTech/Arduino-FreeRTOS-SAMD21 (accessed January 13, 2025).

[25] MartinL. Changing Arduino Zero PWM Frequency. Arduino Forum 2015. https://forum.arduino.cc/t/changing-arduino-zero-pwm-frequency/334231/3 (accessed January 13, 2025).

[26] Raez MBI, Hussain MS, Mohd-Yasin F. Techniques of EMG signal analysis: detection, processing, classification and applications. Biol Proced Online 2006;8:11–35. https://doi.org/10.1251/bpo115.

[27] Carvalho CR, Fernández JM, del-Ama AJ, Oliveira Barroso F, Moreno JC. Review of electromyography onset detection methods for real-time control of robotic exoskeletons. J Neuroeng Rehabil 2023;20. https://doi.org/10.1186/s12984-023-01268-8.

[28] R.A.R.C. Gopura DSVBJMPGTSSJ. Electrodiagnosis in New Frontiers of Clinical Research - Recent trends in EMG-Based control methods for assistive robots. IntechOpen; 2013.

6 Anhang

~/Documents/PlatformIO/Projects/Finger_Setting_Test/platformio.ini Printed on 7.1.2025, 2:02

```
1 ;platformio.ini
2 ;Konfig-Datei, ist für die verwendeten Projekte für die
  Tests und Messungen gleich
3
4 [env:mkrwifi1010]
5 platform = atmelsam
6 board = mkrwifi1010
7 framework = arduino
8 lib_deps = briscoetech/FreeRTOS_SAMD21@^2.3.0 ;Einbindung
  FreeRTOS-Bibliothek
9 monitor_speed = 115200 ;Baudrate
```

1

```
1
2  //Stresstest Kapitel 2.2.2.1 und 3.1
3  #include <FreeRTOS_SAMD21.h>
4  // Include FreeRTOS library
5  #include <Arduino.h>
6
7  //Task Nummer und Stack Größe
8  #define MAX_NUM 20
9  #define STACK_SIZE 128
10
11 // Task-Funktion
12 void vTask(void* pvParameters) {
13 uint8_t taskNum = *((uint8_t*)pvParameters);
14  for(;;)
15  {
16 Serial.print("Tasknum: ");
17 Serial.println(taskNum);
18
19 Serial.print("Free heap: ");
20 Serial.println(xPortGetFreeHeapSize()); //Return
   verfügbaren Heapsize
21
22 vTaskDelay(3000);
23  }
24
25 }
26
27 void setup() {
28     // Serial Initialisieren
29     while (!Serial);
30     Serial.begin(115200);
31     uint8_t tasknum[MAX_NUM];
32
33
34
35 //Schleife um Tasks zu erstellen
36  for(uint8_t i = 0; i < MAX_NUM; i++)
```

```
37  {
38          tasknum[i] = i;
39
40          xTaskCreate( vTask,"Heap_Task", STACK_SIZE,
        (void*)&tasknum[i], 1, NULL);
41          Serial.print("Free Heap nach Erstellung von Task ");
42          Serial.print("Nr. ");
43          Serial.println(i+1);
44          Serial.println(xPortGetFreeHeapSize());
45
46  }
47      Serial.println("Starte Scheduler:");
48      // Start des FreeRTOS scheduler
49      vTaskStartScheduler();
50
51
52  }
53
54
55  void loop() {
56      // Hier nichts zu tun, alles wird vom Scheduler erledigt
57  }
```

```
1
2  //Stresstest 2 Kapitel 2.2.2.1 und 3.1
3  #include <FreeRTOS_SAMD21.h>
4  // Include FreeRTOS library
5  #include <Arduino.h>
6
7  //Eingestellte Delayzeiten und Pinnummern
8  #define PIN_NUM 8
9  #define DELAY 500
10
11
12 const int pin[PIN_NUM] = {0,1,2,3,4,5,6,7}; //PIN 0-7 als
   Ausgänge
13 const int delay_time[PIN_NUM] ={10,10,10,10,10,10,10,10}; //
   gewünschte Delayzeiten
14
15 // TaskHandle_t pin_Handle; //PIN Handle hier nicht nötig
16
17 // Task Funktion
18 void vTaskPinControl(void *pvParameters);
19
20 void setup() {
21     // Initialize Serial
22     Serial.begin(115200);
23     while (!Serial);  // Auf Serial warten
24
25
26 for(uint8_t i = 0; i < 8; i++)
27 {
28 pinMode(pin[i], OUTPUT);
29 xTaskCreate(vTaskPinControl, "PinTask", 128,
   (void*)&pin[i], pin[i+1], NULL);
30 }
31     // Start des FreeRTOS scheduler
32     vTaskStartScheduler();
33
34
```

```
35
36 }
37
38 void loop() {
39     // Hier nichts zu tun, alles wird vom Scheduler erledigt
40 }
41
42 // Task für PIN Toggle
43 void vTaskPinControl(void *pvParameters) {
44 uint8_t  pin_num = *((uint8_t*)pvParameters);
45 TickType_t xLastWakeTime;
46 uint32_t startTime = 0; // Startzeit messen
47 uint32_t endTime = 0; // Endzeit messen
48 uint32_t elapsedTime = 0;
49 // Initialisiere den letzten Aufrufzeitpunkt
50 xLastWakeTime = xTaskGetTickCount();
51 for(;;){
52
53   startTime = micros(); // Startzeit messen
54    volatile uint32_t stressVar = 0; //volatile variable
55         for (int i = 0; i < 400000; i++) {
56             stressVar += i * i;  // Berechnung, um die CPU
       zu belasten
57         }
58
59         digitalWrite(pin[pin_num],
     digitalRead(pin[pin_num])^ 1);
60         Serial.print("Task on Pin ");
61         Serial.print(pin_num);
62         Serial.println(" toggling.");
63         Serial.print("Current Tick Count: ");
64         Serial.println(xTaskGetTickCount()); // Print
     aktuellen tick count
65
66     endTime = micros(); // Endzeit messen
67     elapsedTime = endTime - startTime;
68     //Serial.print(elapsedTime);
```

```
69      //Serial.println(" µs");
70
71          // Berechne Zyklen
72          uint32_t cycles = elapsedTime * 48; // Taktzyklen
        bei 48 MHz
73          Serial.print("Cycles: ");
74          Serial.println(cycles);
75          Serial.println("");
76          if (pin_num == 1)
77          {
78              vTaskSuspendAll();
79          }
80    vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(DELAY));
81   }
82  }
```

```cpp
//Prototypenprojekt Kapitel 3.2
#include <FreeRTOS_SAMD21.h>
#include <Arduino.h>

//Messpins
#define BUTTON_PIN 7
#define LED_PIN 6
#define PWM_PIN 5
#define SENSOR_PIN A0

#define PWM_MEASURE_PIN 0
#define SENSOR_MEASURE_PIN 1
#define HANDLER_MEASURE_PIN 2
#define LED_MEASURE_PIN 3
#define ISR_MEASURE_PIN 4

#define DEBOUNCE_DELAY 0 // 1000 ms debounce delay
volatile unsigned long lastDebounceTime = 0;


// Global state variable um task ON/OFF-Modus zu prüfen
volatile bool tasksRunning = true;

// Task handles
TaskHandle_t interrupt_LEDHandle;
TaskHandle_t measureTaskHandle;
TaskHandle_t pwmTaskHandle;
TaskHandle_t handlerTaskHandle;

// Semaphore handle
SemaphoreHandle_t taskControlSemaphore;

// Queue für Datentransfer zwischen tasks
QueueHandle_t sensorQueue;
volatile BaseType_t xShouldSuspend = pdFALSE;
volatile BaseType_t xShouldResume = pdFALSE;

```

```
38
39 // ISR die die Semaphore an den handlerTask übergibt
40 void buttonISR() {
41     digitalWrite(ISR_MEASURE_PIN,1);
42     unsigned long currentTime = millis();
43     if ((currentTime - lastDebounceTime) > DEBOUNCE_DELAY) {
44         lastDebounceTime =
       currentTime;                                // Update last
       debounce time
45
46     BaseType_t xHigherPriorityTaskWoken = pdFALSE;
47     xSemaphoreGiveFromISR(taskControlSemaphore,
     &xHigherPriorityTaskWoken);
48     portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
49
50     }
51     digitalWrite(ISR_MEASURE_PIN,0);
52 }
53
54 // ISR Handler
55 void handlerTask(void *pvParameters) {
56
57     for (;;) {
58         digitalWrite(HANDLER_MEASURE_PIN, 1);
59         // Warte auf Semaphore von ISR
60         if (xSemaphoreTake(taskControlSemaphore,
       portMAX_DELAY) == pdTRUE) {
61             tasksRunning = !tasksRunning;
62
63             if (tasksRunning) {
64             // Serial.println("Resuming tasks...");
65             xTaskNotifyGive(measureTaskHandle);
66             xTaskNotifyGive(pwmTaskHandle);
67
68              digitalWrite(HANDLER_MEASURE_PIN, 0);
69             }
70             else
```

```
71              {
72                  //Serial.println("Suspending tasks...");
73
74                  digitalWrite(HANDLER_MEASURE_PIN, 0);
75
76              }
77
78          }
79      }
80  }
81
82
83
84  void interrupt_LED(void *pvParameters) {
85
86          TickType_t xLastWakeTime;
87  // Initialisiere den letzten Aufrufzeitpunkt
88  xLastWakeTime = xTaskGetTickCount();
89  for(;;){
90
91    digitalWrite(LED_MEASURE_PIN,
     digitalRead(LED_MEASURE_PIN)^ 1);
92
93    digitalWrite(LED_PIN, digitalRead(LED_PIN)^ 1);
94  if (tasksRunning == false)
95  {
96
97    vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(100));
98
99
100 }
101 else if (tasksRunning == true)
102 {
103     vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(1000));
104
105
106
```

```
107 }
108  }
109 
110 }
111 
112 void pwmTask(void *pvParameters) {
113     int receivedValue;
114 
115     for (;;) {
116 
117 
118         digitalWrite(PWM_MEASURE_PIN,
    digitalRead(PWM_MEASURE_PIN)^ 1);
119         // Warte auf notification vom handlerTask to um
    fortzusetzen fals tasks suspended sind
120         if(!tasksRunning)
121         {
122         ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
123         }
124         // Warte um Messwert vom measureTask zu erhalten
125         if (xQueueReceive(sensorQueue, &receivedValue,
    portMAX_DELAY) == pdPASS)
126         {
127 
128             //  PWM-Signal basierend auf receivedValue
     generieren
129             int pwmValue = map(receivedValue, 0, 1023, 0,
     255);
130             analogWrite(PWM_PIN, pwmValue);
131 
132         // Debugging output
133         //Serial.print("PWM adjusted to: ");
134         // Serial.println(pwmValue);
135 
136         }
137          digitalWrite(PWM_MEASURE_PIN,
    digitalRead(PWM_MEASURE_PIN)^ 1);
```

```
138
139     }
140
141 }
142
143 // Task um analoges SIgnal zu messen
144 void measureTask(void *pvParameters) {
145     int sensorValue;
146     TickType_t xLastWakeTime;
147     xLastWakeTime = xTaskGetTickCount();
148     for (;;) {
149
150         digitalWrite(SENSOR_MEASURE_PIN,1); //Measure Output
151         // Warte auf notification vom handlerTask to um
    fortzusetzen fals tasks suspended sind
152         if(!tasksRunning)
153         {
154         ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
155         xLastWakeTime = xTaskGetTickCount(); //Reset
    Tickcount damit kein Backlog von vTaskDelayUntil()
    entsteht
156         }
157
158
159         // Sensorwert lesen
160         sensorValue = analogRead(SENSOR_PIN);
161
162         // Sensorwert an PWM-task senden
163         xQueueSend(sensorQueue, &sensorValue, 0);
164
165         // Debugging output
166         //Serial.print("Sensor value: ");
167        // Serial.println(sensorValue);
168
169         digitalWrite(SENSOR_MEASURE_PIN,0);
170
171         // Wait 50 ms
```

```
172
173         vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(50));
174     }
175 }
176
177
178 void setup() {
179     Serial.begin(115200);
180     delay(100);
181     // Set up  pin
182     pinMode(BUTTON_PIN, INPUT_PULLUP);
183     pinMode(ISR_MEASURE_PIN, OUTPUT);
184     pinMode(LED_MEASURE_PIN, OUTPUT);
185     pinMode(PWM_MEASURE_PIN, OUTPUT);
186     pinMode(HANDLER_MEASURE_PIN, OUTPUT);
187     pinMode(SENSOR_MEASURE_PIN, OUTPUT);
188
189
190    //Queue für Datentransfer
191     sensorQueue = xQueueCreate(10, sizeof(int));
192     if (sensorQueue == NULL) {
193         Serial.println("Queue creation failed!");
194         while (1); // Stop here if queue creation failed
195     }
196
197     // Initialisiere die semaphore
198     taskControlSemaphore = xSemaphoreCreateBinary();
199
200
201
202     // Taskkreierung
203     xTaskCreate(measureTask, "Measuretask", 128, NULL, 3,
    &measureTaskHandle);
204     xTaskCreate(pwmTask, "pwmTask", 128, NULL, 2,
    &pwmTaskHandle);
205     xTaskCreate(interrupt_LED, "interrupt_LED", 128, NULL,
    1, &interrupt_LEDHandle);
```

```
206     xTaskCreate(handlerTask, "HandlerTask", 128, NULL, 4,
    &handlerTaskHandle);
207 
208     //ISR wenn Button triggered
209     attachInterrupt(digitalPinToInterrupt(BUTTON_PIN),
    buttonISR, FALLING);
210 
211     // Start des FreeRTOS scheduler
212     vTaskStartScheduler();
213 }
214 
215 void loop() {
216    // Hier nichts zu tun, alles wird vom Scheduler erledigt
217 }
```

```cpp
//Kapitel 2.2.3 und 3.2 Vollaufbaumessung
#include <FreeRTOS_SAMD21.h>
// Include FreeRTOS library

#include <Arduino.h>
#include <Wire.h> // Definitionen laden fuer I2C
#include <SPI.h>
#include <avr/pgmspace.h> // weitere Definitionen



//Setup for Brick'N'Knowledge System Literaturverweis [16]

#ifdef ARDUINO_SAMD_MKRWIFI1010
#define __AVR_SAMD21__   1

#endif


// Alle Adressen bei den 8574xx bricks:
#define i2cIO9534_0 (0x40>>1) // Trick um elegant mit Bytes
#define i2cIO9534_1 (0x42>>1) // zu arbeiten statt in 7 Bit
#define i2cIO9534_2 (0x44>>1) // das letzte Bit ist das R/W
#define i2cIO9534_3 (0x46>>1) // dass von der Arduino
#define i2cIO9534_4 (0x48>>1) // Bibliothek dazugefuegt wird
#define i2cIO9534_5 (0x4A>>1) // wir definieren hier alle
#define i2cIO9534_6 (0x4C>>1) // Bereiche die man mit den
#define i2cIO9534_7 (0x4E>>1) // Bricks einstellen kann,.

#define i2cIO9534A_0 (0x70>>1) // Die Serie PCF8474AT
#define i2cIO9534A_1 (0x72>>1) // beginnt bei Adresse
#define i2cIO9534A_2 (0x74>>1) // 0x70 = 70 sedezimal
#define i2cIO9534A_3 (0x76>>1)  // 01110000 binaer
#define i2cIO9534A_4 (0x78>>1) // oder intern 0111000
#define i2cIO9534A_5 (0x7A>>1) // dabei 0111xxx
#define i2cIO9534A_6 (0x7C>>1) // mit x fuer die
#define i2cIO9534A_7 (0x7E>>1) // Dilschalterposition
```

```
38
39 #define myi2cIOadr i2cIO9534_0 // HIER PASSENDE ADRESSE
   EINTRAGEN
40
41 #define SSPB 10 // Pin 10 = SS  - --- PB2  1<<2 als Maske
42
43 // SPI: SCLK MOSI MISO und SS Brick=PB2 D10 fest !!
44 // MEGA Pin 51=MOSI Pin 52=SCK und Pin50=MISO °!!!
45 // mega:
46 // NANO: MOSI:PB3 D11, MISO: PB4 D12, -SS:PB2:D10
47 // UNO: MOSI: PB3 D11, MISO: PB4 D12, -SS:PB2:D10
48 // ACHUNG: MISO ist auch STatus DATA READY !!!
49 //
50 // PORTB 4  (7 6 5 4 -- 3 2 1 0)
51 // BEI MKR PD6
52
53 #if defined(__AVR_ATmega2560__)  // Arduino MEGA2560
54 #define MISOPIN 50
55 #warning ("INFO : compiles for AT MEGA2560")
56 #elif defined(__AVR_SAMD21__)
57 #warning ("INFO : compiles for MKR WIFI1010")
58 #define MISOPIN 2  // READ ONLY MKR ueber Diode...
59 #else // NANO
60 #warning ("INFO : compiles rest: also NANO...")
61 #define MISOPIN 12
62 #endif
63
64 // #define MISOPIN 50
65
66 void enabless()
67 {
68   //digitalWrite(SSPB, LOW);
69   #if defined(__AVR_SAMD21__)
70     digitalWrite(7, LOW);
71   #else
72    PORTB &= 0xfb; // Default = high 7 6 5 4 3 2low 1 0 PB2
    low
```

```
73    #endif
74    delayMicroseconds(10);
75  }
76  void disabless()
77  {
78    delayMicroseconds(10);
79    #if defined(__AVR_SAMD21__)
80      digitalWrite(7, HIGH);
81    #else
82      PORTB |= 1 << 2; // Default = high bit2 PB2
83    #endif
84    // digitalWrite(SSPB, HIGH);
85    delayMicroseconds(10);
86    //
87  }
88
89  //AD 7177-x
90  // Register
91  // SS:-------_______________----------
92
93  //
94  unsigned short ad71_readid()
95  {
96    unsigned char ch1;
97    unsigned short val1=0;
98    enabless();
99    ch1 = SPI.transfer(0x47); // COmm: -wen r-w ra5..9  r-1=1
     read 0=writ
100   val1 = SPI.transfer16(0); // ID
101   disabless();
102   return(val1);
103 }
104
105 //
106
107 //
108 #define NOCHAN 1
```

```
109
110 void ad71_init()
111 {
112   enabless();
113   // Register setzen...
114   SPI.transfer(0x01); SPI.transfer16(0x800C);  //
    ADCMODE ! ref en, mode continouus, clock Crystal : 100 00
    000  0 000 11 00
115   SPI.transfer(0x02); SPI.transfer16(0x0442);  //
    INTERFACE MOD data_stat(6)en w32 en --- 000 0   0 10 0
    0 1   0 0  00 1 0  old 0x0042
116   SPI.transfer(0x06); SPI.transfer16(0x060d);  // GPIO 000
    0  0 11 0   0 0 00 1101
117   // Channel 0
118   SPI.transfer(0x10); SPI.transfer16(0x8001);  // CH0
    enable set0 ain0 ain1 ... 10 00 00 00 000 0 0001
119   // Channel 1
120   SPI.transfer(0x11); SPI.transfer16(0x0043);  // CH1
    enable set0 ain2 ain3 ... 10 00 00 00 010 0 0011
121   // Channel 2
122   SPI.transfer(0x12); SPI.transfer16(0x0096);  // CH2
    enable set0 ain4 ref- ... 10 00 00 00 100 1 0110
123   // Channel 3  optional
124   SPI.transfer(0x13); SPI.transfer16(0x0232);  // CH3
    enable set0 smp+ tmp- ... 10 00 00 10 001 1 0010
125   // alle gleiches setup daher setup 0
126   // SETup registers.
127   SPI.transfer(0x20); SPI.transfer16(0x1300);  // SET0 :
    000 1  0 0 1 1  0 0 00 0000
128   SPI.transfer(0x21); SPI.transfer16(0x1300);  // SET1 :
    000 1  0 0 1 1  0 0 00 0000
129   SPI.transfer(0x22); SPI.transfer16(0x1300);  // SET2 :
    000 1  0 0 1 1  0 0 00 0000
130   SPI.transfer(0x23); SPI.transfer16(0x1300);  // SET3 :
    000 1  0 0 1 1  0 0 00 0000
131   // Filter Config gehoeren zu setup reg
132   SPI.transfer(0x28); SPI.transfer16(0x050e);  // FLT0 : 0
```

```
    000 0 101   x 00 0 1010 sps (00111 10k 01000 5k...)
133   SPI.transfer(0x29); SPI.transfer16(0x050a);  // FLT1 : 0
    000 0 101   x 00 0 1010 sps (00111 10k 01000 5k...)
134   SPI.transfer(0x2A); SPI.transfer16(0x050a);  // FLT2 : 0
    000 0 101   x 00 0 1010 sps (00111 10k 01000 5k...)
135   SPI.transfer(0x2B); SPI.transfer16(0x050a);  // FLT3 : 0
    000 0 101   x 00 0 1010 sps (00111 10k 01000 5k...)
136   // OFFSET und GAIN auf factory lassen...
137   disabless();
138
139 }
140
141 unsigned long values[4];
142
143
144
145 void ad71_reset()
146 {
147   enabless();
148   SPI.transfer(0xff);
149   SPI.transfer(0xff);
150   SPI.transfer(0xff);
151   SPI.transfer(0xff);
152   SPI.transfer(0xff);
153   SPI.transfer(0xff);
154   SPI.transfer(0xff);
155   SPI.transfer(0xff);
156   disabless();
157   delayMicroseconds(550);
158 }
159
160
161 void ad71_dataread()
162 {
163   // Wenn ready incl status 4 byts 32 bit
164   // STATUS
165   unsigned char sts;
```

```
166   unsigned char ch1;
167   unsigned long val1=0;
168   unsigned long val2=0;
169   unsigned int idx;
170   static int first = 0;
171   int val=0;
172   int i=0;
173   for (i=0; i<NOCHAN;i++) {
174     // SYNC:
175     // NICHT RDY Abfragen,,,
176   #ifdef XXXXXX
177     enabless();
178     do {
179
180       sts = SPI.transfer(0x40);
181       sts = SPI.transfer(0x00);
182        delayMicroseconds(10);
183        // Serial.print(sts);
184        // Serial.println("S wt:");
185
186     } while (((sts & 0x80)== 0x80) && (first == 1));
187     first = 1;
188     disabless();
189 #endif
190
191     // DATA RDY
192     delayMicroseconds(10);
193     //
194     enabless();
195     delayMicroseconds(4); // DRDY valid...
196     // SINGLE MODE SPI.transfer(0x01);
    SPI.transfer16(0x801C);  // ADCMODE ! ref en, mode SINGLE,
    clock Crystal : 100 00 000  0 000 11 00
197     //
198     // Sonst abfragen hier immer...
199     do {
200       val = digitalRead(MISOPIN); // MISOPIN
```

```
201     } while (val == HIGH);
202 
203     //
204     ch1 = SPI.transfer(0x44); // COmm:  data read
205 
206     //
207     val1 = SPI.transfer16(0); // data <
208     val2 = SPI.transfer16(0); // data
209     //
210 
211      //
212     delayMicroseconds(5); // DRDY valid...
213     disableless();
214     //
215     val1 = val2 | (val1 << 16); // 4 bit status
216     idx = val2 & 0x3; // CHANNEL FLG  bit 3210
217     values[idx] = val1 & 0xfffffff0;
218 
219   }
220 }
221 
222 //Setup Brick'N'Knowledge end
223 
224 
225 
226 
227 //Pin Number Defines
228 #define BUTTON_PIN 5
229 #define LED_PIN 6
230 #define PWM_PIN 7
231 #define SENSOR_PIN A0
232 
233 //Measure pins
234 #define SENSOR_MEASURE_PIN 0
235 #define PWM_MEASURE_PIN 1
236 #define ISR_MEASURE_PIN 3
237 #define HANDLER_MEASURE_PIN 4
```

```
238 //#define LED_MEASURE_PIN 5
239
240 #define STACK_SIZE 200
241 #define VREF 2.5  // Reference Spannung in Volt
242
243 #define DEBOUNCE_DELAY 1000 // 1000 ms debounce delay für
    den Button
244 volatile unsigned long lastDebounceTime = 0;
245
246 // Global state variable um task ON/OFF-Modus zu prüfen
247 volatile bool tasksRunning = true;
248
249 // Task handles
250 TaskHandle_t interrupt_LEDHandle;
251 TaskHandle_t ReadEMGHandle;
252 TaskHandle_t pwmTaskHandle;
253 TaskHandle_t handlerTaskHandle;
254
255
256
257 // Queue um Daten zwischen tasks zu senden
258 QueueHandle_t sensorQueue;
259 volatile BaseType_t xShouldSuspend = pdFALSE;
260 volatile BaseType_t xShouldResume = pdFALSE;
261
262 // Semaphore handle
263 SemaphoreHandle_t taskControlSemaphore;
264
265 // Prototypen der Task-Funktionen
266 void buttonISR(); //ISR Triggered durch Button Press
267 void vHandlerTask(void *pvParameters); //Handler nach ISR
268 void vTaskReadEMG(void *pvParameters); // Liest Value des
    EMG
269 void vControl_LED(void *pvParameters); //LED für visuellen
    Feedback on Mode Switch ON/OFF
270 void vPWMTask(void *pvParameters); // PWM erhält wert von
    ReadEMG
```

```
271
272
273
274
275 void setup() {
276   // Setze die Pins auf Output
277
278     // Set up button pin
279     pinMode(BUTTON_PIN, INPUT_PULLUP);
280
281     //Set up measure Pins
282     pinMode(ISR_MEASURE_PIN, OUTPUT);
283     //pinMode(LED_MEASURE_PIN, OUTPUT);
284     pinMode(PWM_MEASURE_PIN, OUTPUT);
285     pinMode(HANDLER_MEASURE_PIN, OUTPUT);
286     pinMode(SENSOR_MEASURE_PIN, OUTPUT);
287     pinMode(PWM_PIN, OUTPUT);
288
289      delay(100);
290
291   while (!Serial); //Wait for Serial
292
293   Serial.begin(115200);
294   Serial.println("Starting EEG:");
295
296   #if defined(__AVR_SAMD21__)
297    pinMode(MISOPIN, INPUT); // MISO !!
298    pinMode(7, OUTPUT);
299   #else
300    pinMode(12, INPUT); // MISO !!
301    pinMode(SSPB, OUTPUT);
302   #endif
303   disabless();
304
305
306
307   Wire.begin(); // I2C aktivieren !
```

```
308   Wire.beginTransmission(myi2cIOadr);// Startvorgang I2C
    Adresse
309   Wire.write(0x03) ; // IO Ports auf 01010101 abwechseln
310   Wire.write(0xF0) ;  // 1=input
311   Wire.endTransmission(); // Stop Kondition setzen bei I2C
312   Wire.beginTransmission(myi2cIOadr);// Startvorgang I2C
    Adresse
313   Wire.write(0x01) ; // IO Ports auf 01010101 abwechseln
314   Wire.write(0x0f) ;  // led an !!
315   Wire.endTransmission(); // Stop Kondition setzen bei I2C
316   // INIT
317   SPI.begin();
318   SPI.beginTransaction(SPISettings(7000000, MSBFIRST,
    SPI_MODE3));
319   // NICHT VERWENDEN:
320   //SPI.setDataMode(SPI_MODE3);  // sonst immer 1     clk
    ---____----   rising edhe = spi mode 3 krit mkr´
321   //SPI.setClockDivider(8);
322
323   Serial.println("Searching EEG:");
324   unsigned short id=ad71_readid(); // 0x4fdx
325
326   while ((id & 0xfff0)!= 0x4fd0) {
327      Serial.print("ERROR id=");Serial.print(id);
328      Serial.println(" EEG Brick not found");
329      Wire.beginTransmission(myi2cIOadr);// Startvorgang I2C
     Adresse
330      Wire.write(0x01) ; // IO Ports auf 01010101 abwechseln
331      Wire.write(0x07) ;  // led an !!
332      Wire.endTransmission(); // Stop Kondition setzen bei
     I2C
333      delay(100);
334      Wire.beginTransmission(myi2cIOadr);// Startvorgang I2C
     Adresse
335      Wire.write(0x01) ; // IO Ports auf 01010101 abwechseln
336      Wire.write(0x0f) ;  // led an !!
337      Wire.endTransmission(); // Stop Kondition setzen bei
```

```
     I2C
338     delay(100);
339     id=ad71_readid(); // 0x4fdx
340   }
341
342   Serial.print("id=");Serial.println(id);
343   // OK Init dr Register
344   ad71_reset();
345   ad71_init();
346
347    //Queue für Datentransfer
348     sensorQueue = xQueueCreate(10, sizeof(int));
349
350
351     if (sensorQueue == NULL) {
352         Serial.println("Queue creation failed!");
353         while (1);
354     }
355
356     // Initialize die Semaphore
357     taskControlSemaphore = xSemaphoreCreateBinary();
358
359
360   //PWM Set-up Literaturverweis [24]
361     REG_GCLK_GENDIV = GCLK_GENDIV_DIV(1) |          //
     Divide the 48MHz clock source by divisor 1: 48MHz/1=48MHz
362                       GCLK_GENDIV_ID(4);            // Select
           Generic Clock (GCLK) 4
363    while (GCLK->STATUS.bit.SYNCBUSY);              // Wait
    for synchronization
364
365    REG_GCLK_GENCTRL = GCLK_GENCTRL_IDC |           // Set
    the duty cycle to 50/50 HIGH/LOW
366                       GCLK_GENCTRL_GENEN |         // Enable
           GCLK4
367                       GCLK_GENCTRL_SRC_DFLL48M |   // Set
           the 48MHz clock source
```

```
368                       GCLK_GENCTRL_ID(4);           // Select
        GCLK4
369   while (GCLK->STATUS.bit.SYNCBUSY);              // Wait
for synchronization
370
371   // Enable the port multiplexer for the digital pin D7
372   PORT-
>Group[g_APinDescription[7].ulPort].PINCFG[g_APinDescriptio
n[7].ulPin].bit.PMUXEN = 1;
373
374
375   // Connect the TCC0 timer to digital output D7 - port
pins are paired odd PMUO and even PMUXE
376   // F & E specify the timers: TCC0, TCC1 and TCC2
377   PORT-
>Group[g_APinDescription[6].ulPort].PMUX[g_APinDescription[
6].ulPin >> 1].reg = PORT_PMUX_PMUXO_F;
378
379   // Feed GCLK4 to TCC0 and TCC1
380   REG_GCLK_CLKCTRL = GCLK_CLKCTRL_CLKEN |         // Enable
GCLK4 to TCC0 and TCC1
381                       GCLK_CLKCTRL_GEN_GCLK4 |     // Select
        GCLK4
382                       GCLK_CLKCTRL_ID_TCC0_TCC1;   // Feed
        GCLK4 to TCC0 and TCC1
383   while (GCLK->STATUS.bit.SYNCBUSY);              // Wait
for synchronization
384
385   // Dual slope PWM operation: timers countinuously count
up to PER register value then down 0
386   REG_TCC0_WAVE |= TCC_WAVE_POL(0xF) |         // Reverse
the output polarity on all TCC0 outputs
387                     TCC_WAVE_WAVEGEN_DSBOTH;    // Setup
       dual slope PWM on TCC0
388   while (TCC0->SYNCBUSY.bit.WAVE);               // Wait
for synchronization
389
```

```
390   // Each timer counts up to a maximum or TOP value set by
      the PER register,
391   // this determines the frequency of the PWM operation:
392   REG_TCC0_PER = 239999;          // Set the frequency of
      the PWM on TCC0 to 250kHz
393   while (TCC0->SYNCBUSY.bit.PER);               // Wait
      for synchronization
394
395   // Set the PWM signal to output 50% duty cycle
396   REG_TCC0_CC3 = (239999 * 18) / 100;        // TCC0 CC3 -
      on D7
397   while (TCC0->SYNCBUSY.bit.CC3);               // Wait
      for synchronization
398
399
400    REG_TCC0_CC3 = (239999 * 13) / 100;             // TCC0
      CC1 - 17% duty cycle on D5
401    while (TCC0->SYNCBUSY.bit.CC3);
402   // Divide the 48MHz signal by 1 giving 48MHz (20.83ns)
      TCC0 timer tick and enable the outputs
403   REG_TCC0_CTRLA |= TCC_CTRLA_PRESCALER_DIV1 |    // Divide
      GCLK4 by 1
404                     TCC_CTRLA_ENABLE;             // Enable
             the TCC0 output
405   while (TCC0->SYNCBUSY.bit.ENABLE);              // Wait
      for synchronization
406
407   //Code Source https://forum.arduino.cc/t/changing-arduino-
      zero-pwm-frequency/334231/3
408
409
410
411   // Erstelle die FreeRTOS Tasks
412   xTaskCreate(vTaskReadEMG, "ReadEMG", STACK_SIZE, NULL, 3,
      &ReadEMGHandle);
413   xTaskCreate(vPWMTask, "pwmTask", STACK_SIZE, NULL, 2,
      &pwmTaskHandle);
```

```
414   xTaskCreate(vControl_LED, "interrupt_LED", STACK_SIZE,
    NULL, 1, &interrupt_LEDHandle);
415   xTaskCreate(vHandlerTask, "HandlerTask", STACK_SIZE,
    NULL, 4, &handlerTaskHandle);
416
417
418   //ISR wenn Button triggered
419   attachInterrupt(digitalPinToInterrupt(BUTTON_PIN),
    buttonISR, FALLING);
420
421   // Starte den Scheduler
422   vTaskStartScheduler();
423 }
424
425 void loop() {
426   // loop bleibt leer, da FreeRTOS den Scheduler benutzt
427 }
428
429
430 void vTaskReadEMG(void *pvParameters) {
431   TickType_t xLastWakeTime;
432   int32_t sensorValue = 0;
433   float voltage1 = 0.0f;
434   xLastWakeTime = xTaskGetTickCount();
435   long v1 = 0; // Adjust for two's complement
436   long delta = 0;
437
438   for (;;) {
439
440         digitalWrite(SENSOR_MEASURE_PIN,1); //Measure Output
441
442         // Warte auf Notification vom Handler-Task
443
444         if(!tasksRunning)
445         {
446         ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
447         xLastWakeTime = xTaskGetTickCount(); //Reset
```

```
      Tickcount damit kein Backlog von vTaskDelayUntil()
      entsteht
448         }
449
450     delta = values[0] - 0x80000000l;
451     ad71_dataread();  // EMG-Daten einlesen
452
453     // Read values from CH0 and CH1
454     v1 = values[0] - 0x80000000l;  // Adjust for two's
    complement
455     voltage1 = ((double)v1 / (double)(0x80000000) * VREF) *
    1000;  // Skalierung von -VREF to +VREF in mV
456     // v2 = values[1] - 0x80000000l;
457     sensorValue = v1;
458
459
460
461
462     delta = delta - v1; //delta zwischen letzten Einlesen
    und aktuellem
463
464
465     if (abs(delta) > 40000)
466     {
467        xQueueSend(sensorQueue, &sensorValue, 10);
468
469     }
470
471      //Serial.print("CH0 Voltage: ");
     Serial.print(voltage1, 6); Serial.println("mV ");
472 /*
473     //Debug Outputs
474     if (counter > 50)
475     {
476     Serial.println("Measure Task ");
477     Serial.print("CH0 Voltage: "); Serial.print(voltage1,
    6); Serial.println("mV "); // 6 decimal places
```

```
478     Serial.print("CH0: "); Serial.println(v1);
479     Serial.print("delta"); Serial.println(delta);
480    // Serial.print("CH1: "); Serial.println(v2);
481     counter = 0;
482 
483     }
484 counter++;
485 */
486 
487 
488  digitalWrite(SENSOR_MEASURE_PIN,0); //Measure Output
489     // Delay for 20ms before the next read
490 vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(20));
491 
492 
493   }
494 
495 }
496 
497 // ISR die die Semaphore an den handlerTask übergibt
498 void buttonISR() {
499     digitalWrite(ISR_MEASURE_PIN,1);
500 
501     unsigned long currentTime = millis();
502     if ((currentTime - lastDebounceTime) > DEBOUNCE_DELAY) {
503 
504         lastDebounceTime =
    currentTime;                              // Update last
    debounce time
505 
506     BaseType_t xHigherPriorityTaskWoken = pdFALSE;
507     xSemaphoreGiveFromISR(taskControlSemaphore,
   &xHigherPriorityTaskWoken);
508     portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
509 
510     }
511 
```

```
512     digitalWrite(ISR_MEASURE_PIN,0);
513
514
515 }
516
517 // ISR Handler
518 void vHandlerTask(void *pvParameters) {
519
520     for (;;) {
521
522          // Warte auf Semaphore von ISR
523         if (xSemaphoreTake(taskControlSemaphore,
    portMAX_DELAY) == pdTRUE) {
524             digitalWrite(HANDLER_MEASURE_PIN, 1);
525             tasksRunning = !tasksRunning;
526
527             if (tasksRunning) {
528                 Serial.println("Resuming tasks...");
529             xTaskNotifyGive(ReadEMGHandle);
530             xTaskNotifyGive(pwmTaskHandle);
531
532              digitalWrite(HANDLER_MEASURE_PIN, 0);
533             }
534             else
535             {
536                 Serial.println("Suspending tasks...");
537
538                 digitalWrite(HANDLER_MEASURE_PIN, 0);
539
540             }
541
542
543         }
544
545
546
547     }
```

```
548 }
549
550 void vControl_LED(void *pvParameters) {
551
552 int32_t receivedValue;
553 TickType_t xLastWakeTime;
554 const TickType_t xTimeout = pdMS_TO_TICKS(300);
555 // Initialisiere den letzten Aufrufzeitpunkt
556 xLastWakeTime = xTaskGetTickCount();
557 for(;;){
558  //digitalWrite(LED_MEASURE_PIN,1);
559
560
561
562
563 if (tasksRunning == false  )
564 {
565   digitalWrite(LED_PIN, digitalRead(LED_PIN)^ 1);
566   vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(100));
567
568   //digitalWrite(LED_MEASURE_PIN,0);
569 }
570 else if (tasksRunning == true)
571 {
572     digitalWrite(LED_PIN, digitalRead(LED_PIN)^ 1);
573     vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(1000));
574
575    // digitalWrite(LED_MEASURE_PIN,0);
576 }
577
578
579  }
580
581 }
582
583 void vPWMTask(void *pvParameters) {
584     int32_t receivedValue;
```

```
585     TickType_t xLastWakeTime = xTaskGetTickCount();
586     const TickType_t xTimeout = pdMS_TO_TICKS(300); //
    Delay um auf die zu warten Queue
587     bool tasksRunning = true;
588     bool isFingerOpened = false;
589 
590     for (;;) {
591 
592       digitalWrite(PWM_MEASURE_PIN,1); //Measure Output
593 
594         // Warte auf notification vom handlerTask to um
      fortzusetzen fals tasks suspended sind
595         if (!tasksRunning) {
596             ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
597 
598         }
599 
600         // Warte um Messwert vom measureTask zu erhalten
601         if (xQueueReceive(sensorQueue, &receivedValue,
      xTimeout) == pdPASS) {
602 
603             // PWM um Finger zu schließen
604             REG_TCC0_CC3 = (239999 * 18) / 100;
605             isFingerOpened = false;  // Set duty cycle auf
        13%
606 
607             //Serial.println("schließen...");
608 
609             digitalWrite(PWM_MEASURE_PIN,1); //Measure
        Output
610 
611 
612         }
613         else if (!isFingerOpened) //wird nur ausgeführt
      nachdem der Finger geschlossen wurde
614         {
615           digitalWrite(PWM_MEASURE_PIN,0);
```

```
616
617
618                 REG_TCC0_CC3 = (239999 * 13) / 100;
619                 //Serial.println("Öffnen...");
620
621                 vTaskDelay(pdMS_TO_TICKS(20)); //Delay im
        Finger öffnen zu lassen
622
623
624
625               // PWM-Signal ausschalten
626               REG_TCC0_CC3 = 0;
627               Serial.println("Finger im Ruhezustand (PWM
       0).");
628
629                 isFingerOpened = true;
630
631                 digitalWrite(PWM_MEASURE_PIN,1); //Measure
        Output
632             }
633
634         }
635 }
636
637
638
639
640
641
```

```
1  //Kapitel 3.2 Schwellenwert Methode
2  //Finger Aktuierung Methode über gleitenden Mittelwert
3  #include <FreeRTOS_SAMD21.h>
4  // Include FreeRTOS library
5
6  #include <Arduino.h>
7  #include <Wire.h> // Definitionen laden fuer I2C
8  #include <SPI.h>
9  #include <avr/pgmspace.h> // weitere Definitionen
10
11
12
13 //Setup für Brick'N'Knowledge System Literaturverweis [16]
14
15 #ifdef ARDUINO_SAMD_MKRWIFI1010
16 #define __AVR_SAMD21__   1
17
18 #endif
19
20
21 // Alle Adressen bei den 8574xx bricks:
22 #define i2cIO9534_0 (0x40>>1) // Trick um elegant mit Bytes
23 #define i2cIO9534_1 (0x42>>1) // zu arbeiten statt in 7 Bit
24 #define i2cIO9534_2 (0x44>>1) // das letzte Bit ist das R/W
25 #define i2cIO9534_3 (0x46>>1) // dass von der Arduino
26 #define i2cIO9534_4 (0x48>>1) // Bibliothek dazugefuegt wird
27 #define i2cIO9534_5 (0x4A>>1) // wir definieren hier alle
28 #define i2cIO9534_6 (0x4C>>1) // Bereiche die man mit den
29 #define i2cIO9534_7 (0x4E>>1) // Bricks einstellen kann,.
30
31 #define i2cIO9534A_0 (0x70>>1) // Die Serie PCF8474AT
32 #define i2cIO9534A_1 (0x72>>1) // beginnt bei Adresse
33 #define i2cIO9534A_2 (0x74>>1) // 0x70 = 70 sedezimal
34 #define i2cIO9534A_3 (0x76>>1)  // 01110000 binaer
35 #define i2cIO9534A_4 (0x78>>1) // oder intern 0111000
36 #define i2cIO9534A_5 (0x7A>>1) // dabei 0111xxx
37 #define i2cIO9534A_6 (0x7C>>1) // mit x fuer die
```

```
38 #define i2cIO9534A_7 (0x7E>>1) // Dilschalterposition
39
40 #define myi2cIOadr i2cIO9534_0 // HIER PASSENDE ADRESSE
   EINTRAGEN
41
42 #define SSPB 10 // Pin 10 = SS  - --- PB2  1<<2 als Maske
43
44 // SPI: SCLK MOSI MISO und SS Brick=PB2 D10 fest !!
45 // MEGA Pin 51=MOSI Pin 52=SCK und Pin50=MISO °!!!
46 // mega:
47 // NANO: MOSI:PB3 D11, MISO: PB4 D12, -SS:PB2:D10
48 // UNO: MOSI: PB3 D11, MISO: PB4 D12, -SS:PB2:D10
49 // ACHUNG: MISO ist auch STatus DATA READY !!!
50 //
51 // PORTB 4  (7 6 5 4 -- 3 2 1 0)
52 // BEI MKR PD6
53
54 #if defined(__AVR_ATmega2560__)  // Arduino MEGA2560
55 #define MISOPIN 50
56 #warning ("INFO : compiles for AT MEGA2560")
57 #elif defined(__AVR_SAMD21__)
58 #warning ("INFO : compiles for MKR WIFI1010")
59 #define MISOPIN 2  // READ ONLY MKR ueber Diode...
60 #else // NANO
61 #warning ("INFO : compiles rest: also NANO...")
62 #define MISOPIN 12
63 #endif
64
65 // #define MISOPIN 50
66
67 void enabless()
68 {
69   //digitalWrite(SSPB, LOW);
70   #if defined(__AVR_SAMD21__)
71     digitalWrite(7, LOW);
72   #else
73    PORTB &= 0xfb; // Default = high 7 6 5 4 3 2low 1 0 PB2
```

```
 low
  #endif
  delayMicroseconds(10);
}
void disabless()
{
  delayMicroseconds(10);
  #if defined(__AVR_SAMD21__)
    digitalWrite(7, HIGH);
  #else
    PORTB |= 1 << 2; // Default = high bit2 PB2
  #endif
  // digitalWrite(SSPB, HIGH);
  delayMicroseconds(10);
  //
}

//AD 7177-x
// Register
// SS:-------_______________----------

//
unsigned short ad71_readid()
{
  unsigned char ch1;
  unsigned short val1=0;
  enabless();
  ch1 = SPI.transfer(0x47); // COmm: -wen r-w ra5..9  r-1=1
 read 0=writ
  val1 = SPI.transfer16(0); // ID
  disabless();
  return(val1);
}

//

//
```

```
109 #define NOCHAN 1
110
111 void ad71_init()
112 {
113   enabless();
114   // Register setzen...
115   SPI.transfer(0x01); SPI.transfer16(0x800C);  //
    ADCMODE ! ref en, mode continouus, clock Crystal : 100 00
    000  0 000 11 00
116   SPI.transfer(0x02); SPI.transfer16(0x0442);  //
    INTERFACE MOD data_stat(6)en w32 en --- 000 0   0 10 0
    0 1   0 0  00 1 0  old 0x0042
117   SPI.transfer(0x06); SPI.transfer16(0x060d);  // GPIO 000
    0  0 11 0   0 0 00 1101
118   // Channel 0
119   SPI.transfer(0x10); SPI.transfer16(0x8001);  // CH0
    enable set0 ain0 ain1 ... 10 00 00 00 000 0 0001
120   // Channel 1
121   SPI.transfer(0x11); SPI.transfer16(0x0043);  // CH1
    enable set0 ain2 ain3 ... 10 00 00 00 010 0 0011
122   // Channel 2
123   SPI.transfer(0x12); SPI.transfer16(0x0096);  // CH2
    enable set0 ain4 ref- ... 10 00 00 00 100 1 0110
124   // Channel 3  optional
125   SPI.transfer(0x13); SPI.transfer16(0x0232);  // CH3
    enable set0 smp+ tmp- ... 10 00 00 10 001 1 0010
126   // alle gleiches setup daher setup 0
127   // SETup registers.
128   SPI.transfer(0x20); SPI.transfer16(0x1300);  // SET0 :
    000 1  0 0 1 1  0 0 00 0000
129   SPI.transfer(0x21); SPI.transfer16(0x1300);  // SET1 :
    000 1  0 0 1 1  0 0 00 0000
130   SPI.transfer(0x22); SPI.transfer16(0x1300);  // SET2 :
    000 1  0 0 1 1  0 0 00 0000
131   SPI.transfer(0x23); SPI.transfer16(0x1300);  // SET3 :
    000 1  0 0 1 1  0 0 00 0000
132   // Filter Config gehoeren zu setup reg
```

```
133     SPI.transfer(0x28); SPI.transfer16(0x050e);  // FLT0 : 0
    000 0 101   x 00 0 1010 sps (00111 10k 01000 5k...)
134     SPI.transfer(0x29); SPI.transfer16(0x050a);  // FLT1 : 0
    000 0 101   x 00 0 1010 sps (00111 10k 01000 5k...)
135     SPI.transfer(0x2A); SPI.transfer16(0x050a);  // FLT2 : 0
    000 0 101   x 00 0 1010 sps (00111 10k 01000 5k...)
136     SPI.transfer(0x2B); SPI.transfer16(0x050a);  // FLT3 : 0
    000 0 101   x 00 0 1010 sps (00111 10k 01000 5k...)
137     // OFFSET und GAIN auf factory lassen...
138     disabless();
139
140 }
141
142 unsigned long values[4];
143
144
145
146
147 void ad71_reset()
148 {
149   enabless();
150   SPI.transfer(0xff);
151   SPI.transfer(0xff);
152   SPI.transfer(0xff);
153   SPI.transfer(0xff);
154   SPI.transfer(0xff);
155   SPI.transfer(0xff);
156   SPI.transfer(0xff);
157   SPI.transfer(0xff);
158   disabless();
159   delayMicroseconds(550);
160 }
161
162
163 void ad71_dataread()
164 {
165   // Wenn ready incl status 4 byts 32 bit
```

```
166   // STATUS
167   unsigned char sts;
168   unsigned char ch1;
169   unsigned long val1=0;
170   unsigned long val2=0;
171   unsigned int idx;
172   static int first = 0;
173   int val=0;
174   int i=0;
175   for (i=0; i<NOCHAN;i++) {
176     // SYNC:
177     // NICHT RDY Abfragen,,,
178   #ifdef XXXXXX
179     enabless();
180     do {
181
182       sts = SPI.transfer(0x40);
183       sts = SPI.transfer(0x00);
184        delayMicroseconds(10);
185        // Serial.print(sts);
186        // Serial.println("S wt:");
187
188     } while (((sts & 0x80)== 0x80) && (first == 1));
189     first = 1;
190     disabless();
191 #endif
192
193     // DATA RDY
194     delayMicroseconds(10);
195     //
196     enabless();
197     delayMicroseconds(4); // DRDY valid...
198     // SINGLE MODE SPI.transfer(0x01);
    SPI.transfer16(0x801C);  // ADCMODE ! ref en, mode SINGLE,
    clock Crystal : 100 00 000  0 000 11 00
199     //
200     // Sonst abfragen hier immer...
```

```
201     do {
202       val = digitalRead(MISOPIN); // MISOPIN
203     } while (val == HIGH);
204 
205     //
206     ch1 = SPI.transfer(0x44); // COmm:  data read
207 
208     //
209     val1 = SPI.transfer16(0); // data <
210     val2 = SPI.transfer16(0); // data
211     //
212 
213      //
214     delayMicroseconds(5); // DRDY valid...
215     disabless();
216     //
217     val1 = val2 | (val1 << 16); // 4 bit status
218     idx = val2 & 0x3; // CHANNEL FLG  bit 3210
219     values[idx] = val1 & 0xfffffff0;
220 
221   }
222 }
223 
224 //Setup Brick'N'Knowledge end
225 
226 
227 
228 
229 //Pin Number Defines
230 #define BUTTON_PIN 5
231 #define LED_PIN 6
232 #define PWM_PIN 7
233 #define SENSOR_PIN A0
234 
235 //Measure pins
236 #define SENSOR_MEASURE_PIN 0
237 #define PWM_MEASURE_PIN 1
```

```
238 #define ISR_MEASURE_PIN 3
239 #define HANDLER_MEASURE_PIN 4
240 //#define LED_MEASURE_PIN 5
241
242
243 #define STACK_SIZE 256
244 #define VREF 2.5  // Reference voltage in volts
245
246 #define DEBOUNCE_DELAY 1000 // 1000 ms debounce delay to
    reduce button bounce
247 volatile unsigned long lastDebounceTime = 0;
248
249 // Global state variable um task ON/OFF-Modus zu prüfen
250 volatile bool tasksRunning = true;
251
252 // Task handles
253 TaskHandle_t interrupt_LEDHandle;
254 TaskHandle_t ReadEMGHandle;
255 TaskHandle_t pwmTaskHandle;
256 TaskHandle_t handlerTaskHandle;
257
258
259
260 // Queue um Daten zwischen tasks zu senden
261 QueueHandle_t sensorQueue;
262 volatile BaseType_t xShouldSuspend = pdFALSE;
263 volatile BaseType_t xShouldResume = pdFALSE;
264
265 // Semaphore handle
266 SemaphoreHandle_t taskControlSemaphore;
267
268 // Prototypen der Task-Funktionen
269 void buttonISR(); //ISR Triggered durch Button Press
270 void vHandlerTask(void *pvParameters); //Handler nach ISR
271 void vTaskReadEMG(void *pvParameters); // Liest Value des
    EMG
272 void vControl_LED(void *pvParameters); //LED für visuellen
```

```
   Feedback on Mode Switch ON/OFF
273 void vPWMTask(void *pvParameters); // PWM erhält wert von
   ReadEMG
274
275
276
277 void setup() {
278   // Setze die Pins auf Output
279
280     // Set up button pin
281     pinMode(BUTTON_PIN, INPUT_PULLUP);
282
283     //Set up measure Ppns
284     pinMode(ISR_MEASURE_PIN, OUTPUT);
285     //pinMode(LED_MEASURE_PIN, OUTPUT);
286     pinMode(PWM_MEASURE_PIN, OUTPUT);
287     pinMode(HANDLER_MEASURE_PIN, OUTPUT);
288     pinMode(SENSOR_MEASURE_PIN, OUTPUT);
289     pinMode(PWM_PIN, OUTPUT);
290
291      delay(100);
292
293   while (!Serial); //Wait for Serial for proper Sync
294
295 // put your setup code here, to run once:
296   Serial.begin(115200);
297   Serial.println("Starting EEG:");
298
299   #if defined(__AVR_SAMD21__)
300    pinMode(MISOPIN, INPUT); // MISO !!
301    pinMode(7, OUTPUT);
302   #else
303    pinMode(12, INPUT); // MISO !!
304    pinMode(SSPB, OUTPUT);
305   #endif
306   disableSS();
307
```

```
308
309
310   Wire.begin(); // I2C aktivieren !
311   Wire.beginTransmission(myi2cIOadr);// Startvorgang I2C
    Adresse
312   Wire.write(0x03) ; // IO Ports auf 01010101 abwechseln
313   Wire.write(0xF0) ;  // 1=input
314   Wire.endTransmission(); // Stop Kondition setzen bei I2C
315   Wire.beginTransmission(myi2cIOadr);// Startvorgang I2C
    Adresse
316   Wire.write(0x01) ; // IO Ports auf 01010101 abwechseln
317   Wire.write(0x0f) ;  // led an !!
318   Wire.endTransmission(); // Stop Kondition setzen bei I2C
319   // INIT
320   SPI.begin();
321   SPI.beginTransaction(SPISettings(7000000, MSBFIRST,
    SPI_MODE3));
322   // NICHT VERWENDEN:
323   //SPI.setDataMode(SPI_MODE3);  // sonst immer 1     clk
    ---___----   rising edhe = spi mode 3 krit mkr´
324   //SPI.setClockDivider(8);
325
326   Serial.println("Searching EEG:");
327   unsigned short id=ad71_readid(); // 0x4fdx
328
329   while ((id & 0xfff0)!= 0x4fd0) {
330      Serial.print("ERROR id=");Serial.print(id);
331      Serial.println(" EEG Brick not found");
332      Wire.beginTransmission(myi2cIOadr);// Startvorgang I2C
     Adresse
333      Wire.write(0x01) ; // IO Ports auf 01010101 abwechseln
334      Wire.write(0x07) ;  // led an !!
335      Wire.endTransmission(); // Stop Kondition setzen bei
     I2C
336      delay(100);
337      Wire.beginTransmission(myi2cIOadr);// Startvorgang I2C
     Adresse
```

```
338       Wire.write(0x01) ; // IO Ports auf 01010101 abwechseln
339       Wire.write(0x0f) ;  // led an !!
340       Wire.endTransmission(); // Stop Kondition setzen bei
     I2C
341       delay(100);
342       id=ad71_readid(); // 0x4fdx
343    }
344
345    Serial.print("id=");Serial.println(id);
346    // OK Init dr Register
347    ad71_reset();
348    ad71_init();
349
350     //Queue für Datentransfer
351      sensorQueue = xQueueCreate(10, sizeof(int));
352      if (sensorQueue == NULL) {
353          Serial.println("Queue creation failed!");
354          while (1); // Stop here if queue creation failed
355      }
356
357      // Initialize the semaphore
358      taskControlSemaphore = xSemaphoreCreateBinary();
359
360
361   //PWM Set-up Literaturverweis [24]
362      REG_GCLK_GENDIV = GCLK_GENDIV_DIV(1) |          //
     Divide the 48MHz clock source by divisor 1: 48MHz/1=48MHz
363                        GCLK_GENDIV_ID(4);            // Select
           Generic Clock (GCLK) 4
364     while (GCLK->STATUS.bit.SYNCBUSY);              // Wait
    for synchronization
365
366     REG_GCLK_GENCTRL = GCLK_GENCTRL_IDC |           // Set
    the duty cycle to 50/50 HIGH/LOW
367                        GCLK_GENCTRL_GENEN |         // Enable
           GCLK4
368                        GCLK_GENCTRL_SRC_DFLL48M |   // Set
```

```
            the 48MHz clock source
369                     GCLK_GENCTRL_ID(4);           // Select
            GCLK4
370   while (GCLK->STATUS.bit.SYNCBUSY);              // Wait
    for synchronization
371
372   // Enable the port multiplexer for the digital pin D7
373   PORT-
    >Group[g_APinDescription[7].ulPort].PINCFG[g_APinDescriptio
    n[7].ulPin].bit.PMUXEN = 1;
374
375
376   // Connect the TCC0 timer to digital output D7 - port
    pins are paired odd PMUO and even PMUXE
377   // F & E specify the timers: TCC0, TCC1 and TCC2
378   PORT-
    >Group[g_APinDescription[6].ulPort].PMUX[g_APinDescription[
    6].ulPin >> 1].reg = PORT_PMUX_PMUXO_F;
379
380   // Feed GCLK4 to TCC0 and TCC1
381   REG_GCLK_CLKCTRL = GCLK_CLKCTRL_CLKEN |         // Enable
    GCLK4 to TCC0 and TCC1
382                     GCLK_CLKCTRL_GEN_GCLK4 |     // Select
            GCLK4
383                     GCLK_CLKCTRL_ID_TCC0_TCC1;   // Feed
            GCLK4 to TCC0 and TCC1
384   while (GCLK->STATUS.bit.SYNCBUSY);              // Wait
    for synchronization
385
386   // Dual slope PWM operation: timers countinuously count
    up to PER register value then down 0
387   REG_TCC0_WAVE |= TCC_WAVE_POL(0xF) |         // Reverse
    the output polarity on all TCC0 outputs
388                     TCC_WAVE_WAVEGEN_DSBOTH;    // Setup
            dual slope PWM on TCC0
389   while (TCC0->SYNCBUSY.bit.WAVE);               // Wait
    for synchronization
```

```
390
391   // Each timer counts up to a maximum or TOP value set by
      the PER register,
392   // this determines the frequency of the PWM operation:
393   REG_TCC0_PER = 239999;          // Set the frequency of
      the PWM on TCC0 to 250kHz
394   while (TCC0->SYNCBUSY.bit.PER);                // Wait
      for synchronization
395
396   // Set the PWM signal to output 50% duty cycle
397   REG_TCC0_CC3 = (239999 * 18) / 100;        // TCC0 CC3 -
      on D7
398   while (TCC0->SYNCBUSY.bit.CC3);                // Wait
      for synchronization
399
400
401    REG_TCC0_CC3 = (239999 * 13) / 100;             // TCC0
      CC1 - 17% duty cycle on D5
402    while (TCC0->SYNCBUSY.bit.CC3);
403   // Divide the 48MHz signal by 1 giving 48MHz (20.83ns)
      TCC0 timer tick and enable the outputs
404   REG_TCC0_CTRLA |= TCC_CTRLA_PRESCALER_DIV1 |    // Divide
      GCLK4 by 1
405                     TCC_CTRLA_ENABLE;             // Enable
             the TCC0 output
406   while (TCC0->SYNCBUSY.bit.ENABLE);              // Wait
      for synchronization
407
408   //Code Source https://forum.arduino.cc/t/changing-arduino-
      zero-pwm-frequency/334231/3
409
410
411
412   // Erstelle die FreeRTOS Tasks
413   xTaskCreate(vTaskReadEMG, "ReadEMG", STACK_SIZE, NULL, 3,
      &ReadEMGHandle);
414   xTaskCreate(vPWMTask, "pwmTask", STACK_SIZE, NULL, 2,
```

```
    &pwmTaskHandle);
415   xTaskCreate(vControl_LED, "interrupt_LED", STACK_SIZE,
    NULL, 1, &interrupt_LEDHandle);
416   xTaskCreate(vHandlerTask, "HandlerTask", STACK_SIZE,
    NULL, 4, &handlerTaskHandle);
417   xTaskCreate(vIdleTask, "Idle Task", STACK_SIZE, NULL, 0,
    NULL);
418
419
420   //ISR when Button is Triggered
421   attachInterrupt(digitalPinToInterrupt(BUTTON_PIN),
    buttonISR, FALLING);
422
423   // Starte den Scheduler
424   vTaskStartScheduler();
425 }
426
427 void loop() {
428   // loop bleibt leer, da FreeRTOS den Scheduler benutzt
429 }
430
431
432
433
434 void vTaskReadEMG(void *pvParameters) {
435     float offset = 0.0f; // Dynamischer Offset
436     const float threshold = 0.05f; // Schwellenwert in mV
437     const int filterSize = 10; // Größe des gleitenden
     Mittelwerts
438     float offsetBuffer[filterSize] = {0.0f}; // Puffer für
     den gleitenden Mittelwert
439     int filterIndex = 0; // Index für den Ringpuffer
440     float movingAverage = 0.0f; // Gleitender Mittelwert
441
442     TickType_t xLastWakeTime;
443     float sensorValue = 0;
444     float voltage1 = 0.0f;
```

```
445     xLastWakeTime = xTaskGetTickCount();
446     long v1 = 0;
447     float delta = 0.0f;
448     int counter = 0;
449 
450     for (;;) {
451         // Warten, falls Tasks pausiert sind
452         if (!tasksRunning) {
453             ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
454             xLastWakeTime = xTaskGetTickCount(); //
     Rücksetzen von TickCount
455         }
456 
457         // EMG-Daten einlesen
458         ad71_dataread();
459         v1 = values[0] - 0x80000000l;  // Anpassung für
    Zweierkomplement
460         voltage1 = ((float)v1 / (float)(0x80000000) * VREF)
    * 1000;  // Skalierung auf mV
461 
462         // Offset kontinuierlich anpassen
463         offset -= offsetBuffer[filterIndex] /
    filterSize; // Alten Wert entfernen
464         offsetBuffer[filterIndex] = voltage1; // Neuen Wert
    hinzufügen
465         offset += voltage1 / filterSize; // Neuen Wert zum
    Offset hinzufügen
466         filterIndex = (filterIndex + 1) % filterSize; //
    Index rotieren
467 
468         // Differenz zwischen gemessenem Wert und Offset
469         delta = voltage1 - offset;
470 
471         // Vergleich mit Schwellenwert<
472         if (fabs(delta) > threshold) {
473             sensorValue = delta;
474 
```

```
475             xQueueSend(sensorQueue, &sensorValue, 10);
476         }
477
478         //Debug-Ausgaben
479         if (counter == 20) {
480              Serial.println("Measure Task");
481             Serial.print("CH0 Voltage: ");
        Serial.print(voltage1, 6); Serial.println("mV ");
482             Serial.print("Offset: ");
        Serial.println(offset, 6);
483             Serial.print("Delta (Voltage - Offset): ");
        Serial.println(delta, 6);
484             counter = 0;
485         }
486         counter++;
487
488         digitalWrite(SENSOR_MEASURE_PIN, 0); // Output
      toggeln
489         vTaskDelayUntil(&xLastWakeTime,
      pdMS_TO_TICKS(20)); // Verzögerung
490      }
491 }
492
493    void vPWMTask(void *pvParameters) {
494     float receivedValue;
495     TickType_t xLastWakeTime = xTaskGetTickCount();
496     const TickType_t xTimeout = pdMS_TO_TICKS(300); //
    Delay um auf die zu warten Queue
497     bool tasksRunning = true;
498     bool isFingerOpened = false;
499
500     for (;;) {
501
502
503
504         // Warte auf notification vom handlerTask to um
      fortzusetzen fals tasks suspended sind
```

```
505         if (!tasksRunning) {
506             ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
507 
508         }
509 
510         // Warte um Messwert vom measureTask zu erhalten
511         if (xQueueReceive(sensorQueue, &receivedValue,
    xTimeout) == pdPASS) {
512 
513             if (fabs(receivedValue) > 0.1)
514             {
515              // PWM um Finger zu schließen
516             REG_TCC0_CC3 = (239999 * 18) / 100;
517             isFingerOpened = false;  // Set duty cycle to
      13%
518             }
519             /*
520             else if (receivedValue > 0.1)
521             {
522            // PWM um Finger zu schließen
523             REG_TCC0_CC3 = (239999 * 17) / 100;
524             isFingerOpened = false;  // Set duty cycle to
      13%
525             }
526             */
527            /*
528                 else if (fabs(receivedValue) > 0.15)
529             {
530 
531           // PWM um Finger zu schließen
532             REG_TCC0_CC3 = (239999 * 16) / 100;
533             isFingerOpened = false;  // Set duty cycle to
      13%
534             }
535            */
536             else if (fabs(receivedValue) > 0.07)
537             {
```

```
538             // PWM um Finger zu schließen
539              REG_TCC0_CC3 = (239999 * 15) / 100;
540              isFingerOpened = false;  // Set duty cycle to
        15%
541              }
542
543               else if (fabs(receivedValue) > 0.05)
544              {
545              // PWM um Finger zu schließen
546              REG_TCC0_CC3 = (239999 * 14) / 100;
547              isFingerOpened = false;  // Set duty cycle to
        14%
548              }
549              //Serial.println("Closing...");
550
551
552
553
554          }
555          else if (!isFingerOpened) //wird nur ausgeführt
      nachdem der Finger geschlossen wurde
556          {
557
558
559
560              REG_TCC0_CC3 = (239999 * 13) / 100;
561              //Serial.println("Öffnen...");
562
563              vTaskDelay(pdMS_TO_TICKS(20)); //Delay im
        Finger öffnen zu lassen
564
565
566
567                // PWM-Signal ausschalten
568              REG_TCC0_CC3 = 0;
569              Serial.println("Finger at rest (PWM 0).");
570
```

```
571                 isFingerOpened = true;
572 
573 
574             }
575 
576         }
577 }
578 
579 
580 
581 // ISR die die Semaphore an den handlerTask übergibt
582 void buttonISR() {
583     digitalWrite(ISR_MEASURE_PIN,1);
584 
585     unsigned long currentTime = millis();
586     if ((currentTime - lastDebounceTime) > DEBOUNCE_DELAY) {
587 
588         lastDebounceTime =
        currentTime;                                // Update last
        debounce time
589 
590     BaseType_t xHigherPriorityTaskWoken = pdFALSE;
591     xSemaphoreGiveFromISR(taskControlSemaphore,
      &xHigherPriorityTaskWoken);
592     portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
593 
594     }
595 
596     digitalWrite(ISR_MEASURE_PIN,0);
597 
598 
599 }
600 
601 // ISR Handler
602 void vHandlerTask(void *pvParameters) {
603 
604     for (;;) {
```

```
605
606         // Warte auf Semaphore von ISR
607         if (xSemaphoreTake(taskControlSemaphore,
    portMAX_DELAY) == pdTRUE) {
608             digitalWrite(HANDLER_MEASURE_PIN, 1);
609             tasksRunning = !tasksRunning;
610
611             if (tasksRunning) {
612                 Serial.println("Resuming tasks...");
613             xTaskNotifyGive(ReadEMGHandle);
614             xTaskNotifyGive(pwmTaskHandle);
615
616              digitalWrite(HANDLER_MEASURE_PIN, 0);
617             }
618             else
619             {
620                 Serial.println("Suspending tasks...");
621
622                 digitalWrite(HANDLER_MEASURE_PIN, 0);
623
624             }
625
626
627         }
628
629
630
631     }
632 }
633
634 void vControl_LED(void *pvParameters) {
635
636 int32_t receivedValue;
637 TickType_t xLastWakeTime;
638 const TickType_t xTimeout = pdMS_TO_TICKS(300);
639 // Initialisiere den letzten Aufrufzeitpunkt
640 xLastWakeTime = xTaskGetTickCount();
```

```
641 for(;;){
642  //digitalWrite(LED_MEASURE_PIN,1);
643
644
645
646
647 if (tasksRunning == false  )
648 {
649   digitalWrite(LED_PIN, digitalRead(LED_PIN)^ 1);
650   vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(100));
651
652   //digitalWrite(LED_MEASURE_PIN,0);
653 }
654 else if (tasksRunning == true)
655 {
656     digitalWrite(LED_PIN, digitalRead(LED_PIN)^ 1);
657     vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(1000));
658
659    // digitalWrite(LED_MEASURE_PIN,0);
660 }
661
662
663  }
664
665 }
666
667
668
669
```