

# FreeRTOS auf dem ATmega2560

Projektarbeit

von

Herrn Nooraldeen Yamk

im Studiengang Elektrotechnik/Informationstechnik  
mit Schwerpunkt Automatisierungstechnik

an der HAWK Hochschule für angewandte Wissenschaft und Kunst  
Hildesheim / Holzminden / Göttingen  
Fakultät Ingenieurwissenschaften und Gesundheit in Göttingen



Februar 2024

# Abbildungsverzeichnis

---

Abbildung 1: Speicherarten [13, S. 144] .....	3
Abbildung 2: Funktionsprinzip des Stacks [20] .....	4
Abbildung 3: Speicherbereiche in C [23] .....	4
Abbildung 4: Context switching [12] .....	5
Abbildung 5: Aufbau eines SPI-Systems [9] .....	5
Abbildung 6: SPI-Modi [24, S. 10] .....	6
Abbildung 7: Free-RTOS Quelldateien .....	8
Abbildung 8: CTC Mode, Timing Diagram [26, S. 146] .....	12
Abbildung 9: CTC-Mode-Bits [26, S. 145] .....	12
Abbildung 10: Prescaler-Bits [26, S. 157] .....	12
Abbildung 11: TCCR1A Register [26, S. 154] .....	12
Abbildung 12: TCCR1B Register [26, S. 156] .....	12
Abbildung 13: TIMSK Register [22, S. 82] .....	13
Abbildung 14: TIMSK1 Register [26, S. 161] .....	13
Abbildung 15: Tasks-Zustände [1] .....	15
Abbildung 16: Queues-Funktionsweise [1, S. 105] .....	17
Abbildung 17: Mutexe-Funktionsweise [1, S. 245] .....	19
Abbildung 18: Prescaler-Bits [26, S. 198] .....	20
Abbildung 19: SPSR-Register [26, S. 198] .....	20
Abbildung 20: SPCR-Register [26, S. 197] .....	20
Abbildung 21: SPI-Schreibprotokoll [30, S. 30] .....	20
Abbildung 22: SPI-Lese-Protokoll [30, S. 29] .....	21

# Listingverzeichnis

---

Lisitng 1: portSAVE_CONTEXT() - Anpassungen .....	9
Lisitng 2: portSTORE_CONTEXT() - Anpassungen .....	10
Lisitng 3: pxPortInitialiseStack() - Anpassungen .....	11
Lisitng 4: Konfiguration der Funktion prvSetupTimerInterrupt() .....	13
Lisitng 5: Task-Prototyp [1, S. 46].....	14
Lisitng 6: xTaskCreate() Functoin [vgl. 1, S. 49] .....	14
Lisitng 7: vTaskDelay()- Function [1, S. 68] .....	16
Lisitng 8: vTaskDelayUntil()- Function [1, S. 72].....	16
Lisitng 9: xQueueCreate()-Function [1, S. 109] .....	18
Lisitng 10: xQueueSendToBack()-Function [1, 110].....	18
Lisitng 11: xQueueRecive()-Function [1, S. 113].....	18
Lisitng 12: xSemaphoreCreateMutex()-Function [1, S. 246] .....	18
Lisitng 13: xSemaphoreTake()-Function [1, S. 196] .....	18
Lisitng 14: xSemaphoreGive()-Function [1, S. 197].....	18
Lisitng 15: Initialisierung der SPI-Schnittstelle .....	20
Lisitng 16: SPI_MasterTransmit() - Function .....	21
Lisitng 17: SPI_MasterReceive() - Function .....	21
Lisitng 18: read_Pressure() - Function .....	21
Lisitng 19: read_Temperature() - Function .....	21
Lisitng 20: Verwendung von xTaskCreate() - Function .....	22
Lisitng 21: Erstellung von vTask1 .....	22
Lisitng 22: Erstellung von vTask2 .....	22
Lisitng 23: Verwendung von xSemaphoreCreateMutex() - Function.....	22
Lisitng 24: Mutex - vTask1 .....	22
Lisitng 25: Mutex - vTask2 .....	22
Lisitng 26: Verwendung von xQueueCreate() - Function.....	23
Lisitng 27: Queue - vTask1 .....	23
Lisitng 28: Queue - vTask2 .....	23

# Abkürzungsverzeichnis

---

<i>RTOS</i> .....	<i>Real Time Operating System</i>
<i>LCD</i> .....	<i>Liquid Crystal Display</i>
<i>LSB</i> .....	<i>Least Significant Bit</i>
<i>MSB</i> .....	<i>Most Significant Bit</i>
<i>FIFO</i> .....	<i>First In First Out</i>
<i>LIFO</i> .....	<i>Last In First Out</i>
<i>HIFO</i> .....	<i>Highest In First Out</i>
<i>RAM</i> .....	<i>Random-Access Memory</i>
<i>SRAM</i> .....	<i>Static-Random-Access Memory</i>
<i>E/A</i> .....	<i>Ein-/Ausgabe</i>
<i>SPI</i> .....	<i>Serial Peripheral Interface</i>
<i>MISO</i> .....	<i>Master Input Slave Output</i>
<i>MOSI</i> .....	<i>Master Output Slave Input</i>
<i>SCLK</i> .....	<i>Serial Clock</i>
<i>SS</i> .....	<i>Slave Select</i>
<i>CS</i> .....	<i>Chip Select</i>
<i>CPOL</i> .....	<i>Clock Polarity</i>
<i>CPHA</i> .....	<i>Clock Phase</i>
<i>CTC</i> .....	<i>Clear Timer on Compare Match</i>
<i>Tx</i> .....	<i>Transmitter</i>
<i>Rx</i> .....	<i>Receiver</i>

# Inhaltsverzeichnis

---

Sperrvermerk .....	II
Abbildungsverzeichnis .....	III
Listingverzeichnis .....	IV
Abkürzungsverzeichnis .....	V
<b>1 Einleitung und Zielsetzung .....</b>	<b>1</b>
<b>2 Grundlagen .....</b>	<b>1</b>
<b>2.1 Echtzeitbetriebssystem.....</b>	<b>1</b>
<b>2.2 FreeRTOS .....</b>	<b>2</b>
<b>2.2.1 Speicherverwaltung.....</b>	<b>3</b>
<b>2.2.2 Kontextwechsel .....</b>	<b>5</b>
<b>2.3 Serial Peripheral Interface (SPI) .....</b>	<b>5</b>
<b>2.3.1 Datenübertragung .....</b>	<b>6</b>
<b>3 Implementierung von FreeRTOS .....</b>	<b>7</b>
<b>3.1 Implementierung der erforderlichen Quelldateien .....</b>	<b>7</b>
<b>3.2 die Änderungen und Anpassungen an port.c.....</b>	<b>8</b>
<b>4 Die wichtigsten Funktionen und Makros .....</b>	<b>14</b>
<b>4.1 Datentypen .....</b>	<b>14</b>
<b>4.2 Tasks .....</b>	<b>14</b>
<b>4.3 Queues .....</b>	<b>16</b>
<b>4.4 Mutexe .....</b>	<b>18</b>
<b>5 Projektaufbau .....</b>	<b>19</b>
<b>5.1 Verwendung von SPI-Schnittstelle .....</b>	<b>19</b>
<b>5.1.1 Initialisierung der SPI-Schnittstelle .....</b>	<b>19</b>
<b>5.1.2 SPI-Schreibprotokoll .....</b>	<b>20</b>

5.1.2 SPI-Lese-Protokoll .....	21
5.2 Sensorintegration in FreeRTOS .....	22
5.1.2 Implementierung von Tasks .....	22
5.1.3 Implementierung vom Mutex.....	22
5.1.4 Implementierung vom Queue.....	23
6 Ergebnisse und Fazit.....	23
Literaturverzeichnis.....	24
Anhang .....	25

# 1 Einleitung und Zielsetzung

---

Die Menschen leben zurzeit in einem Zeitalter, in dem andauernd nach modernen Technologien und entwickelten Systemen gesucht wird. Die winzigen Computer, sogenannte „Embedded System“ haben für diesen großen Fortschritt eine wichtige Rolle gespielt. Sie verbergen sich in vielen Anwendungen, die wir täglich nutzen, von der Uhr bis zum Auto.

In dieser Projektarbeit wird eine Anwendung davon beleuchtet, die im Vergleich zu den anderen Anwendungen geringfügig ist.

Das Hauptziel dieser Projektarbeit ist die Implementierung eines Echtzeitbetriebssystems auf dem ATmega2560 unter der Verwendung von FreeRTOS. All diese Themen werden später im Abschnitt „Grundlagen“ ausführlich erläutert.

Die Zielsetzung dieser Projektarbeit wird nachfolgend in Stichpunkten zusammengefasst:

- Portierung des Open-Source Quellcodes von FreeRTOS für den ATmega2560.
- Programmierung einer Kommunikationsschnittstelle zwischen dem Mikrocontroller und einem Temperatur- und Drucksensor
- Implementierung von zwei Tasks (Threads):
  1. Der erste Task fragt den Sensor ab, und liest kontinuierlich die erforderlichen Daten.
  2. Der zweite Task stellt die erfassten Daten auf dem LCD dar.
- Programmierung der notwendigen Elemente und Funktionen, die zur Thread-Synchronisierung und Thread-Sicherheit dienen.

Die Erreichung diese Ziele veranschaulicht, was ein Echtzeitbetriebssystem ist, und wie es einfach implementiert werden kann.

## 2 Grundlagen

---

### 2.1 Echtzeitbetriebssystem

Ein Betriebssystem (Operating System) ist die Gesamtheit der Programme eines digitalen Computersystems, die zusammen mit den Funktionen der Computeranlage die Grundlage für die möglichen Betriebsarten bilden [vgl. 3, S. 4]. Es steuert und überwacht die Ausführung von Programmen und ermöglicht dem Computer, unterschiedliche Aufgaben effektiv und organisiert auszuführen. [vgl. 2]

Ein Betriebssystem wird als Echtzeitbetriebssystem betrachtet, wenn es die Ausführung wichtige Operationen und Aufgaben innerhalb eines bestimmten Zeitfensters garantieren kann, oder zumindest in der Lage ist, dieses Zeitfenster einzuhalten. Echtzeitbetriebssysteme, die diese Zeitfenster zuverlässig einhalten können, werden als „harte“ Echtzeitbetriebssysteme bezeichnet. Die „weiche“ Echtzeitbetriebssystem hingegen können diese Zuverlässigkeit nur in den meisten Fällen erfüllen. Entscheidend ist, dass ein Echtzeitbetriebssystem durch eine richtige Programmierung die Ausführung eines Programms mit sehr zuverlässigem Timing ermöglicht. [vgl. 4]

## 2.2 FreeRTOS

FreeRTOS ist ein Open-Source-Echtzeitbetriebssystem für eingebettete Systeme und wurde im Jahr 2003 von Richard Barry entwickelt. Die Software ist größtenteils in Programmiersprache C geschrieben, und kann sowohl für private als auch für kommerzielle Zwecke kostenlos verwendet werden. [vgl. 1, S. 2] [vgl. 5]

Mit FreeRTOS können mehrere Tasks erstellt werden, die durch einen konfigurierbaren Scheduler verwaltet werden. Dieser ist ein essenzieller Bestandteil von FreeRTOS und übernimmt die Verwaltung von der Ablaufsteuerung und Zeitmanagement für die verschiedenen Tasks oder Threads in eingebetteten Systemen. Er trägt zur effizienten Verwendung von Systemressourcen bei, indem er die Prozessorzeit zwischen den verschiedenen Tasks verteilt und deren Prioritäten berücksichtigt. [vgl. 1, S. 2]

Auf einem Prozessor mit nur einem Kern kann nur ein Task ausgeführt werden. Der Kernel bestimmt, welcher Task ausgeführt wird, indem er die vom Programmierer zugewiesene Priorität jedes Threads prüft. Im einfachsten Fall könnte der Programmierer höhere Prioritäten für Tasks zuweisen, die harte Echtzeit-Anforderungen erfüllen, und niedrigere Prioritäten für Tasks, die weiche Echtzeit-Anforderungen erfüllen. Dies würde sicherstellen, dass die Tasks mit harten Echtzeit-Anforderungen immer vor Tasks mit weichen Echtzeit-Anforderungen ausgeführt werden. [vgl. 1, S. 2]

Der Scheduler in FreeRTOS ist normalerweise ein prioritätsbasierter, präemptiver Scheduler. Das heißt, er kann die Ausführung eines Tasks jederzeit unterbrechen, um einen anderen mit höherer Priorität zu starten. Nachdem der Task abgeschlossen ist oder in den Wartezustand eintritt, wird der ursprüngliche Task weiter ausgeführt. Dadurch wird sichergestellt, dass wichtige Tasks zeitnah erledigt werden und das gesamte System reaktionsfähig bleibt. [vgl. 5, 6]

Zusätzlich gibt es Methoden zur Kommunikation und Synchronisation zwischen Tasks: Queues, Mutexes, Semaphore. Die Queues und Mutexes werden im Abschnitt „Projektaufbau“ noch tiefer behandelt.

- Queues:

Queues werden für die Übertragung von Daten zwischen Tasks verwendet. Queues ähneln einem großen First In – First Out (FIFO) Puffer. Grundsätzlich ist es tatsächlich FIFO, aber es ist auch möglich, Daten im ersten Puffer zu überschreiben. [vgl. 1, S. 104]

- Semaphore

Binäre Semaphore können nur zwei Zustände annehmen und sind ideal für die Synchronisation zwischen Tasks oder zwischen Tasks und einem Interrupt. Zählende Semaphore hingegen können mehr als zwei Werte annehmen. Sie erlauben vielen Tasks die gemeinsame Nutzung von Ressourcen oder die Durchführung komplexerer Synchronisationsoperationen. [vgl. 1, S. 192] [vgl. 6]

- Mutexe (Mutual Exclusion):

Mutexe sind spezielle binäre Semaphore und dienen dazu, dass nur ein Task auf einen Codeabschnitt zugreifen kann, während sie alle anderen Tasks blockiert. Dadurch wird sichergestellt, dass alles, was in diesem kritischen Abschnitt ausgeführt wird, thread-safe ist und Informationen nicht durch andere Tasks (Threads) beschädigt werden. [vgl. 6]



## 2.2.1 Speicherverwaltung

In Bezug auf die Speicherverwaltung wird vor allem zwischen dem statischen und dynamischen Speicher unterschieden.

### 2.2.1.1 Statischer Speicher

Darunter versteht man ein Speicher mit einer festen Struktur, die während der gesamten Ausführung des Programms unverändert bleibt. Jedes Objekt oder jede Variable behält dauerhaft ihre festgelegte Position im Speicher. Hier werden sowohl globale als auch statische Objekte/Variablen gespeichert. Wenn statische Objekte oder Variablen innerhalb einer Funktion verwendet werden, dann sind sie außerhalb dieser Funktion unsichtbar. Der statische Speicher verändert seine Größe während der Programmausführung nicht. [vgl. 13, S. 144]

- Lokale Variablen sind ausschließlich in der Funktion, in der sie deklariert wurden, gültig und können dort gelesen bzw. beschrieben werden. Darüber hinaus ist Ihre Existenz auf den Funktionsblock begrenzt. Beim Verlassen dieser Funktion geht der Inhalt der Speicherzelle verloren [vgl. 14, S. 20]
- Globale Variablen erhalten einen festen Speicherplatz im SRAM, und der Inhalt der Speicherzelle bleibt während des gesamten Programmablaufs gültig. Deshalb eignen sich globale Variablen besonders gut zum Austausch von Daten zwischen mehreren Funktionen [vgl. 14, S. 21]
- Statische Variablen bekommen ebenso einen festen Speicherbereich im SRAM, die auch nach dem Verlassen der Funktion erhalten bleibt. Im Gegensatz zu den globalen Variablen sind die statischen Variablen nur in der Funktion gültig und sichtbar, in der sie deklariert wurden. [vgl. 14, S. 22]

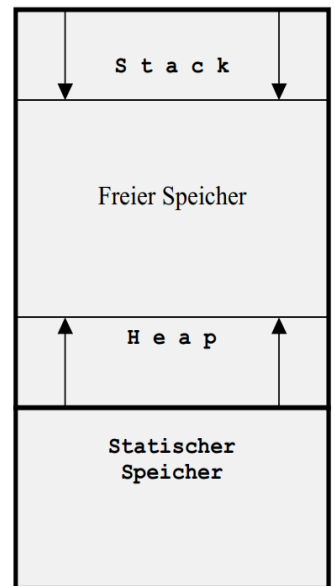


Abbildung 1: Speicherarten [13, S. 144]

### 2.2.1.2 Dynamischer Speicher

Die dynamische Speichertechnik bezieht sich darauf, dass während des Programmablaufs Speicherplatz zur Verfügung gestellt wird, auf dem ein oder mehrere neue Elemente einer Datenelemente platziert werden [vgl. 15, S. 11]. Dieser dynamische Speicher ist ein Teil der RAM, der dynamisch verwaltet wird. Im Gegensatz zum statischen Speicher erlaubt der dynamische Speicher die flexible Allokation und Freigabe von Speicherplatz während der Programmlaufzeit. Daher wird die dynamische Speicherverwaltung besonders wichtig, wenn beim Schreiben des Programms nicht vorher festgelegt ist, wie viele Variablen benötigt werden, sondern diese Entscheidung erst zur Laufzeit getroffen wird. [vgl. 16, S. 13-14]

Die wichtigsten und am häufigsten verwendeten dynamischen Speicher sind der Stack und der Heap.

- Der Stack (Stapelspeicher)  
Stack ist eine lineare Datenstruktur, und wird vom Compiler verwaltet [vgl.13, S. 145]. Er ist ein wichtiger Teil des RAMs und ist eine weit verbreitete dynamische Datenstruktur, die von den meisten Mikroprozessoren direkt über Maschinenbefehle unterstützt wird. Er folgt dem Prinzip der Last-In-First-Out (LIFO) Datenstruktur, bei der das zuletzt hinzugefügte Element als erstes entfernt wird [vgl. 17] [vgl. 18]. Da werden die lokalen Variablen gespeichert. [vgl. 16, S. 3]

Der Stack zeichnet sich durch hohe Effizienz und schnellen Zugriff aus. Die Hauptanwendungsgebiete des Stacks sind die Verwaltung lokaler Variablen und Methodenaufrufe. Außerdem übernimmt der Stack die automatische Verwaltung des Speichers. [vgl. 17] [vgl. 18]

Es gibt drei übliche Operationen, die bei einem Stack durchgeführt werden. [vgl. 19]

**Push-Operation:** Hier wird ein neues Element auf den Stack gelegt und anschließend wird der Stackpointer um eins inkrementiert (Wenn der Stack beispielsweise von unten nach oben aufgebaut ist, wie in der Abbildung 2 dargestellt).

**Pop-Operation** : Das oberste Element im Stack wird entnommen, dann der Pointer wird um eins dekrementiert.

**Peek-Operation** : diese Operation erlaubt, das oberste Element Im Stack anzusehen, ohne es zu entnehmen.

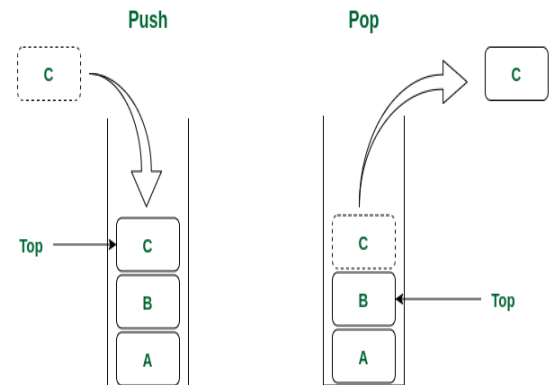


Abbildung 2: Funktionsprinzip des Stacks [20]

- Der Heap (Haufen)

Heap ist eine hierarchische Datenstruktur, und wird manuell verwaltet [vgl.13, S. 145]. Im Vergleich zum Stack ist der Heap langsamer, aber er hat den Vorteil, dass er beliebig groß sein kann. Die Speicherzuweisung im Heap erfolgt durch Funktionen wie *malloc* oder *new*, und für die Freigabe des Speichers werden Funktionen wie *free* oder *delete* verwendet. Die Variablen im Heap stehen global zur Verfügung, was bedeutet, dass alle Tasks einer Anwendung darauf zugreifen können. Im Vergleich zum Stack erfordert die Verwaltung des Heaps mehr Aufwand [vgl. 18]. Der Heap basiert auf der Highest-In-First-Out (HIFO) Datenstruktur [vgl. 21].

FreeRTOS verfügt über fünf Heap-Management-Implementierungen, die sich in Komplexität und Funktionalität unterscheiden. Eine individuelle Heap-Implementierung ist ebenfalls möglich. Bei der Implementierung von FreeRTOS werden die fünf Heap-Implementierungen kurz erläutert. [vgl. 6]

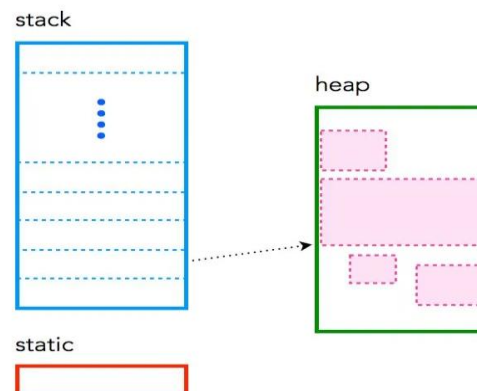


Abbildung 3: Speicherbereiche in C [23]

### 2.2.2 Kontextwechsel

Ein Kontextwechsel (context switch) bezeichnet einen Prozess in einem Betriebssystem, bei dem die Bearbeitung des aktuellen Prozesses (oder Tasks) unterbrochen wird, und zu einer anderen Routine gewechselt wird. Beispielsweise kann diese Unterbrechung durch einen Timer-Interrupt oder Systemaufrufen nach einer vordefinierten Zeit erfolgen [vgl. 10]. Für einen Prozess bedeutet dies, dass das System den aktuellen Prozess sichern muss, um ihn jederzeit genau an der aktuellen Stelle fortsetzen zu können, bevor es den anderen Prozess lädt. Dieser Vorgang ist erforderlich, um beispielsweise Speicherplatz freizugeben oder auf Ereignisse wie Tastatur- oder Mauseingaben zu reagieren [vgl. 11].

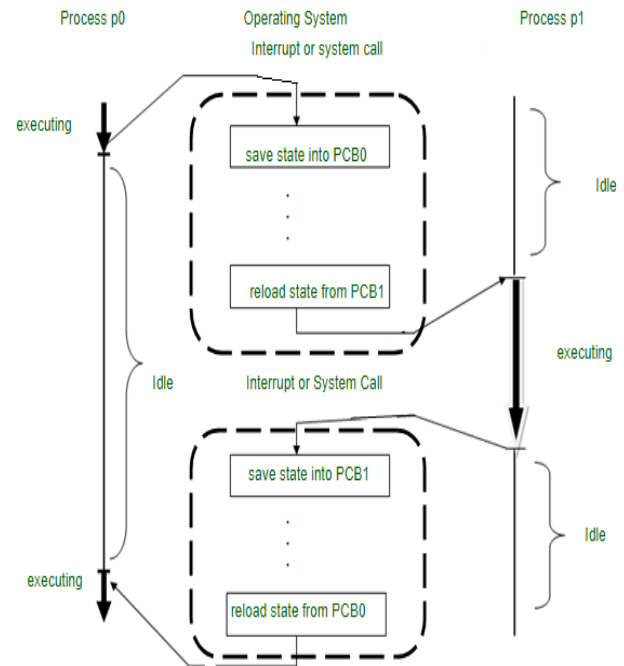


Abbildung 4: Context switching [12]

Kontextwechsel treten besonders beim Multitasking auf. Hierbei ist der Scheduler dafür zuständig sicherzustellen, dass alle im System laufenden Prozesse ihren Anteil an der gesamten Prozessorzeit bekommen, wenn er beispielsweise auf den Abschluss von E/A-Operationen wartet. [vgl. 10]

### 2.3 Serial Peripheral Interface (SPI)

Die synchrone serielle Schnittstelle Serial Peripheral Interface (SPI) wurde ursprünglich von Motorola entwickelt und wird heute häufig für die Kommunikation zwischen integrierten Komponenten verwendet. [vgl.7]

Das SPI ist synchron, vollduplexfähig, und zählt zu den am häufigsten eingesetzten Schnittstellen für die Kommunikation zwischen Mikrocontrollern und Peripherie-ICs. Es bietet eine optimale Lösung für diesen Anwendungsbereich. [vgl. 8]

Der SPI-Bus basiert ebenfalls auf einem Master/Slave-Prinzip. Der Master kann abwechselnd über vier Leitungen mit mehreren Slaves Daten austauschen: [vgl. 8]

- MOSI bezeichnet die Verbindung, über die der Slave Daten vom Master empfängt.
- MISO kennzeichnet die Verbindung, über die der Master die vom Slave ausgehenden Daten erhält
- SCLK ist das vom Master generierte Taktsignal.
- SS oder CS dient zur Auswahl der Slaves.

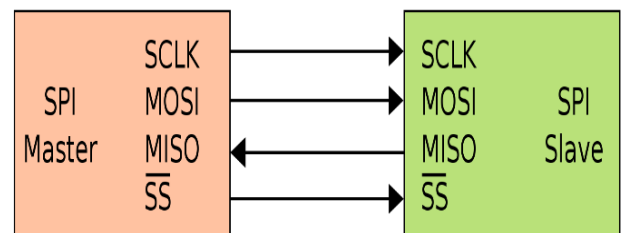


Abbildung 5: Aufbau eines SPI-Systems [9]

### 2.3.1 Datenübertragung

Um die SPI-Kommunikation zu starten, sendet der Master das Taktsignal und wählt den gewünschten Slave aus, indem er das SS-Signal freigibt. Der Master muss das SS-Signal auf logisch 0 setzen, um den Slave auszuwählen, da SS normalerweise eine Aktiv-Low-Charakteristik hat. In der SPI-Schnittstelle können Master und Slave gleichzeitig über die MOSI- bzw. MISO-Leitung Daten senden, da es sich um eine Vollduplex-Schnittstelle handelt. Die Übertragung von Daten erfolgt synchronisiert durch die Taktflanken, wobei die seriellen Taktflanken das Ausgeben und Abtasten der Daten steuern. Es ist die Entscheidung des Programmierers, ob die steigenden oder fallenden Taktflanken zur Synchronisierung des Abtastens und oder Ausgebens der Daten verwendet werden sollen. [vgl. 8]

Es gibt viele verschiedene Konfigurationsparameter, die bei der Konfiguration eines SPI-Geräts berücksichtigt werden müssen. Es ist erforderlich, zu bestimmen, ob die Übertragung mit dem LSB oder dem MSB beginnt. Außerdem gibt es vier verschiedene Modi, taktsynchrone Daten zu übertragen. Die Konfigurationsflags Clock Polarity (CPOL) und Clock Phase (CPHA) werden verwendet, um diese zu bestimmen. Das CPOL-Flag stellt fest, ob die SCLK-Leitung im Leerlauf logisch „0“ oder logisch „1“ ist. Die Taktflanke, auf die die Daten übertragen werden, wird dann vom CPHA-Flag bestimmt. [vgl. 24]

Die Übertragungsfrequenz oder Datenrate wird vom Master SPI-Taktfrequenz bestimmt. Es ist wichtig zu beachten, dass die SPI-Taktfrequenz dem langsamsten Teilnehmer angepasst werden sollte. In der Mikrocontrollerpraxis wird der SPI-Bus häufig verwendet, wenn Daten mit einer hohen Übertragungsrate (1 MHz bis 10 MHz) über kurze Strecken oder innerhalb eines Gerätes übertragen werden müssen. [vgl. 24]

SPI-Mode	CPOL	CPHA	Taktpolarität im Ruhezustand	Taktphase zum Abtasten und/oder Verschieben der Daten
0	0	0	Logisch <i>low</i>	Daten werden bei steigender Flanke gesampled und bei fallenden Flanke verschoben
1	0	1	Logisch <i>low</i>	Daten werden bei fallender Flanke gesampled und bei steigender Flanke verschoben
2	1	1	Logisch <i>high</i>	Daten werden bei fallender Flanke gesampled und bei steigender Flanke verschoben
3	1	0	Logisch <i>high</i>	Daten werden bei steigender Flanke gesampled und bei fallenden Flanke verschoben

Abbildung 6: SPI-Modi [24, S. 10]

## 3 Implementierung von FreeRTOS

---

### 3.1 Implementierung der erforderlichen Quelldateien

FreeRTOS wird in Form eines Pakets von C-Quelldateien angeboten. Einige der Quelldateien sind für alle Ports gemeinsam, während andere für bestimmte Ports bestimmt sind. `FreeRTOSConfig.h` ist die Header-Datei, die verwendet wird, um FreeRTOS zu konfigurieren. [vgl. 1, S. 11]

Der Kern des FreeRTOS-Quellcodes liegt in zwei C-Dateien, die für alle FreeRTOS-Ports gleich sind. Diese sind `tasks.c` und `list.c`. Die beiden Dateien sind immer erforderlich und sind direkt im FreeRTOS/Source-Verzeichnis zu finden. Neben diesen beiden Dateien befinden sich auch die folgenden Quelldateien im gleichen Verzeichnis: [vg.1, S. 12-13]

- `queue.c` stellt sowohl Queue- als auch Semaphore-Dienste bereit und wird in den meisten Fällen benötigt.
- `timers.c` wird nur benötigt, wenn Software-Timer tatsächlich verwendet werden sollen.
- `event_groups.c` wird nur benötigt, wenn Ereignisgruppen tatsächlich verwendet werden sollen.
- `croutine.c` wird nur benötigt, wenn Co-Routinen tatsächlich verwendet werden sollen.

Darüber hinaus müssen spezifischen Quelldateien für den FreeRTOS-Port auch implementiert werden. Sie befinden sich im Verzeichnis FreeRTOS/Source/portable. Das portable Verzeichnis ist hierarchisch strukturiert, zunächst nach Compiler und dann nach Prozessorarchitektur. `port.c` und `portmacro.h` sind die beiden grundlegenden Dateien, die aus diesem Verzeichnis benötigt werden. [vgl. 1, S. 14]

Die Portierung ist für den ATmega32 konzipiert und FreeRTOS bietet für den ATmega2560-Controller keine eigene Implementierung bzw. Portierung an. Aus diesem Grund muss der `port.c` für den ATmega2560 angepasst werden.

Schließlich wird eine der von FreeRTOS bereitgestellten Heap-Implementierungen benötigt. Die fünf Implementierungen werden als `heap_1` bis `heap_5` bezeichnet und sind durch die Quelldateien `heap_1.c` bis `heap_5.c` implementiert. Sie können im Ordner FreeRTOS/Source/portable/MemMang gefunden werden. [vgl. 1, S. 14]

- `heap_1`: ist die zwar einfachste Implementierung, aber ermöglicht keine Speicherfreigabe. [vgl. 6]
- `heap_2`: arbeitet durch die Unterteilung eines Arrays und ist schneller als `heap_1` [vgl.1, S. 30, 32]. Außerdem ermöglicht er die Freigabe von Speicher, aber keine Zusammenführung benachbarter freier Blöcke. [vgl. 6]
- `heap_3`: nutzt die Standardbibliotheksfunktionen `malloc()` und `free()` und gewährleistet Thread-Sicherheit, indem es den FreeRTOS-Scheduler vorübergehend aussetzt. [vgl.1, S. 32]
- `heap_4`: kombiniert benachbarte freie Blöcke, um Fragmentierung zu reduzieren. Darüber hinaus bietet es die Option zur absoluten Platzierung von Adressen. [vgl. 6]
- `heap_5`: ist ähnlich wie `heap_4` und hat die Fähigkeit, den Heap über verschiedene, nicht benachbarte Speicherbereiche zu verteilen. [vgl. 6]

heap\_3 wird ausgewählt, weil hier nach der Thread-Sicherheit gesucht ist und auf die Zusammenführung oder Aufteilung von Blöcken verzichtet werden kann.

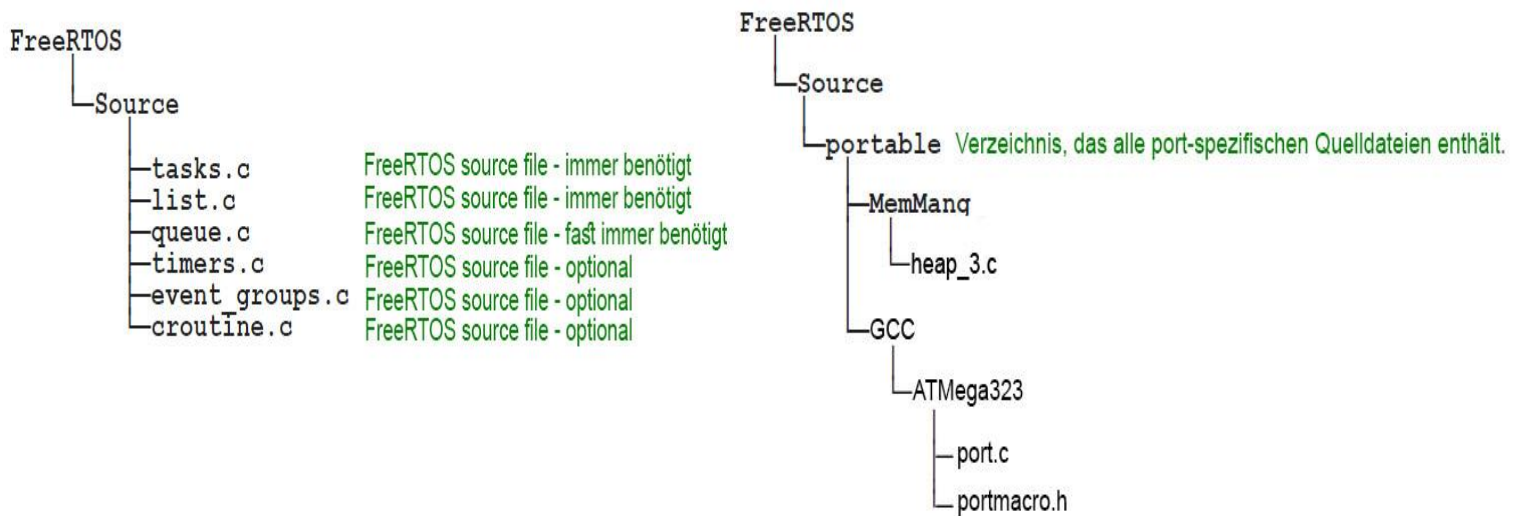


Abbildung 7: Free-RTOS Quelldateien  
 [1, S. 13-14]

Zum Schluss müssen für FreeRTOS im Include-Pfad des Compilers drei Verzeichnisse aufgenommen werden. [vgl. 1, S. 15]

1. Der Standardpfad für FreeRTOS-Headerdateien ist FreeRTOS/Source/include.
2. Der Pfad zu den Quelldateien, die spezifisch für den verwendeten FreeRTOS-Port sind. Wie zuvor beschrieben, befindet sich dieser Pfad unter FreeRTOS/Source/portable/GCC/ATmega32
3. Ein Pfad zur Headerdatei FreeRTOSConfig.h.

### 3.2 die Änderungen und Anpassungen an port.c

Wie bereits erwähnt, existiert keine spezifische Implementierung für den ATmega2560, sondern für den ATmega32. Zwischen den beiden sind einige Unterschiede zu berücksichtigen.

Der ATmega2560 verfügt über einen Flash-Speicher mit einer Größe von 256KBytes, im Gegensatz zu 32KBytes beim ATmega32. Der Hauptunterschied besteht darin, dass der ATmega2560 aufgrund der Größe des Flash-Speichers einen 3-Byte-Programm-Counter hat und nicht wie der ATmega32, der nur 2-Byte-Programm-Counter hat. Wenn eine Codeadresse auf dem Stack abgelegt wird, benötigt er 3 Bytes anstelle von 2 Bytes. [vgl. 25]

Wenn Daten mit der PUSH-Operation auf den Stack gelegt werden, wird der Stackpointer um eins dekrementiert, und um drei dekrementiert, wenn die Rückkehradresse mit einem Aufruf eines Unterprogramms oder Interrupt auf den Stack gelegt wird. Wenn Daten mit der POP-Operation vom Stack genommen werden, wird der Stack-Pointer um eins inkrementiert. Wenn Daten mit einem Rückkehrbefehl (RET) oder einem Rückkehrbefehl nach einem Interrupt (RETI) vom Stack genommen werden, wird der Stack-Pointer um drei inkrementiert. [vgl. 26, S. 15]

Der Kontext auf den AVR-Mikrocontrollern besteht aus:

- 1) Die 32 allgemeine Prozessorregister.
- 2) Statusregister: Die Ausführung von Anweisungen wird durch den Inhalt des Statusregisters beeinflusst und muss über Kontextwechsel hinweg erhalten bleiben.
- 3) Programm-Counter: Nach der Wiederaufnahme muss der Task die Ausführung ab der Anweisung fortsetzen, die unmittelbar vor ihrer Unterbrechung ausgeführt werden sollte.
- 4) Die beiden Stackpointer-register. [vgl. 29, S. 111]

Jeder Task hat seinen eigenen Stack, sodass der Kontext gesichert werden kann, indem die Prozessorregister auf den Stack des Tasks geschoben werden. Zur Speicherung des Kontexts spielt die Assemblersprache eine wesentliche Rolle und ist unvermeidlich. Die Kontextspeicherung erfolgt in einem Makro namens `portSAVE_CONTEXT()`. [vgl. 29, S. 111]

Um den Kontext erfolgreich und fehlerfrei zu speichern, sind zwei CPU-Register erforderlich:


- **RAMPX, RAMPY und RAMPZ**

Sie sind mit den X-, Y- und Z-Registern verbunden, dienen als Erweiterung und ermöglichen die indirekte Adressierung des gesamten Datenspeichers auf den Mikrocontroller mit mehr als 64 KB Datenspeicher sowie das konstante Abrufen von Daten vom Mikrocontroller mit mehr als 64 KB Programmspeicher. [vgl. 27, S. 8]

- **EIND**

Es ist mit dem Z-Register verbunden und ermöglicht indirekte Sprünge und Aufrufe zum gesamten Programmspeicher auf Mikrocontrollern mit mehr als 128 KB Programmspeicher. [vgl. 27, S. 8]

Die erste Anpassung besteht darin, dass die beiden zuvor genannten Register (RAMPZ, EIND) berücksichtigt werden müssen. Dies ist in zwei Makros vorzunehmen: `portSAVE_CONTEXT()` und `portRESOTRE_CONTEXT()`.

<pre>// r0= tmp_reg, r1= zero_reg, RAMPZ= 0x3B, //EIND= 0x3C #define portSAVE_CONTEXT() \     __asm __volatile ( \         "push    __tmp_reg__      \n\t" \ (1)         "in      __tmp_reg__, __SREG__ \n\t" \ (2)         "cli     \n\t" \ (3)         "push    __tmp_reg__      \n\t" \ (4)         "push    __zero_reg__     \n\t" \ (5)         "clr     __zero_reg__     \n\t" \ (6)         "push    r2               \n\t" \ (7)         "push    r3               \n\t" \         "push    r4               \n\t" \         "push    r5               \n\t" \         .         .         "push    r30              \n\t" \         "push    r31              \n\t" \         "lds     r26, pxCurrentTCB \n\t" \ (8)         "lds     r27, pxCurrentTCB + 1 \n\t" \ (9)         "in      __tmp_reg__, __SP_L__ \n\t" \ (10)         "st      x+, __tmp_reg__   \n\t" \ (11)         "in      __tmp_reg__, __SP_H__ \n\t" \ (12)         "st      x+, __tmp_reg__   \n\t" \ (13)     );</pre>		<pre>// r0= tmp_reg, r1= zero_reg, RAMPZ= 0x3B, //EIND= 0x3C #define portSAVE_CONTEXT() \     __asm __volatile ( \         "push    __tmp_reg__      \n\t" \         "in      __tmp_reg__, __SREG__ \n\t" \         "cli     \n\t" \         "push    __tmp_reg__      \n\t" \         "in      __tmp_reg__, 0x3B   \n\t" \ (4.1)         "push    __tmp_reg__      \n\t" \ (4.2)         "in      __tmp_reg__, 0x3C   \n\t" \ (4.3)         "push    __tmp_reg__      \n\t" \ (4.4)         "push    __zero_reg__     \n\t" \         "clr     __zero_reg__     \n\t" \         "push    r2               \n\t" \         "push    r3               \n\t" \         "push    r4               \n\t" \         "push    r5               \n\t" \         .         .         "push    r30              \n\t" \         "push    r31              \n\t" \         "lds     r26, pxCurrentTCB \n\t" \         "lds     r27, pxCurrentTCB + 1 \n\t" \         "in      __tmp_reg__, __SP_L__ \n\t" \         "st      x+, __tmp_reg__   \n\t" \         "in      __tmp_reg__, __SP_H__ \n\t" \         "st      x+, __tmp_reg__   \n\t" \     );</pre>
---	---	---


Lisitng 1: `portSAVE_CONTEXT()` - Anpassungen.



Erläuterung des obigen Quellcodes
Das Prozessorregister R0 wird zuerst gespeichert (1), da es für die Sicherung des Statusregisters verwendet wird.
Das Statusregister wird in R0 verschoben (2), damit es auf den Stack gespeichert werden kann (4).
Prozessor-Interrupts werden deaktiviert (3). Da das Makro <code>portSAVE_CONTEXT()</code> auch außerhalb von Interrupt-Service-Routinen verwendet wird, müssen die Interrupts so früh wie möglich gelöscht werden.
Das RAMPZ-Register wird in R0 verschoben (4.1), damit es auf den Stapel gespeichert werden kann (4.2).
Das EIND-Register wird in R0 verschoben (4.2), damit es auf den Stapel gespeichert werden kann (4.3).
Der vom Compiler generierte Code geht davon aus, dass R1 auf null gesetzt ist. Der ursprüngliche Wert von R1 wird gespeichert (5), bevor R1 gelöscht wird (6).
Zwischen (7) und (8) werden alle verbleibenden Prozessorregister in aufsteigender numerischer Reihenfolge gespeichert.
Der Stack der ausgesetzten Tasks enthält jetzt eine Duplikation des Ausführungskontexts dieses Tasks. Wenn der Task fortgesetzt wird, sichert der Kernel den Stackpointer des Tasks, um den Kontext abzurufen und wiederherzustellen. Das X-Prozessorregister wird mit der Adresse geladen, an der der Stackpointer gespeichert werden soll (8 und 9).
Der Stack-Zeiger wird zuerst in dem Low-Byte (10 und 11) gespeichert, dann im High-Nibble (12 und 13).

[vgl. 29, S. 111-114]

Das Makro `portRESTORE_CONTEXT()` ist das Gegenteil von `portSAVE_CONTEXT()`. Der Kontext des fortgesetzten Tasks wurde zuvor im Stack des Tasks gespeichert. Der Kernel ruft den Stackpointer für den Task ab und lädt dann den Kontext zurück in die entsprechenden Prozessorregister.

<pre>// r0 = __tmp_reg__, r1= __zero_reg__ #define portRESTORE_CONTEXT()  __asm__ __volatile__ (     "lds    r26, pxCurrentTCB      \n\t" \ (1)     "lds    r27, pxCurrentTCB + 1  \n\t" \ (2)     "ld     r28, x+                 \n\t" \     "out    __SP_L__, r28          \n\t" \ (3)     "ld     r29, x+                 \n\t" \     "out    __SP_H__, r29          \n\t" \ (4)     "pop    r31                     \n\t" \     "pop    r30                     \n\t" \     .     .     .     "pop    __zero_reg__           \n\t" \     "pop    __tmp_reg__            \n\t" \     "out    SREG_, __tmp_reg__     \n\t" \ (6)     "pop    __tmp_reg__            \n\t" \ (7) );</pre>		<pre>// r0 = __tmp_reg__, r1= __zero_reg__ #define portRESTORE_CONTEXT()  __asm__ __volatile__ (     "lds    r26, pxCurrentTCB      \n\t" \     "lds    r27, pxCurrentTCB + 1  \n\t" \     "ld     r28, x+                 \n\t" \     "out    __SP_L__, r28          \n\t" \     "ld     r29, x+                 \n\t" \     "out    __SP_H__, r29          \n\t" \     "pop    r31                     \n\t" \     "pop    r30                     \n\t" \     .     .     "pop    __zero_reg__           \n\t" \     "pop    __tmp_reg__            \n\t" \ (5)     "out    0x3C, __tmp_reg__      \n\t" \ (5.1)     "pop    __tmp_reg__            \n\t" \ (5.2)     "out    0x3B, __tmp_reg__      \n\t" \ (5.3)     "pop    __tmp_reg__            \n\t" \ (5.4)     "out    SREG_, __tmp_reg__     \n\t" \     "pop    __tmp_reg__            \n\t" \ );</pre>
---	---	---

Lisitng 2: `portSTORE_CONTEXT()` - Anpassungen

Erläuterung des obigen Quellcodes
Die Variable <code>pxCurrentTCB</code> enthält die Adresse, von der der Stackpointer des Tasks abgerufen werden kann. Diese Adresse wird in das X-Register geladen (1 und 2)
Der Stackpointer für den fortgesetzten Task wird in den AVR-Stackpointer geladen, zuerst das Low-Byte (3), dann das High-Nibble (4).
Die Prozessorregister werden in umgekehrter numerischer Reihenfolge bis zu R1 vom Stapel genommen
Der Statusregister, der auf dem Stapel zwischen den Registern R1 und R0 gespeichert ist, wird daher vor R0 (7) wiederhergestellt (6).

[vgl. 29, S. 114]



Bei der Initialisierung des Stacks sind ebenfalls diese Anpassungen erforderlich.

```
StackType_t *pxPortInitialiseStack(
StackType_t *pxTopOfStack, TaskFunction_t
pxCode, void *pvParameters )
{
    uint16_t usAddress;

    usAddress = ( uint16_t ) pxCode;
    *pxTopOfStack = ( StackType_t ) ( usAddress
    & ( uint16_t ) 0x00ff );
    pxTopOfStack--;

    usAddress >>= 8;
    *pxTopOfStack = ( StackType_t ) ( usAddress
    & ( uint16_t ) 0x00ff );
    pxTopOfStack--;

    *pxTopOfStack = ( StackType_t ) 0x00;
    pxTopOfStack--;
    *pxTopOfStack = portFLAGS_INT_ENABLED;
    pxTopOfStack--;

    .
    .
    .
}
```



```
StackType_t *pxPortInitialiseStack( StackType_t
*pxTopOfStack, TaskFunction_t pxCode, void *pvParameters )
{
    uint16_t usAddress;

    usAddress = ( uint16_t ) pxCode;
    *pxTopOfStack = ( StackType_t ) ( usAddress & ( uint16_t )
    0x00ff );
    pxTopOfStack--;

    usAddress >>= 8;
    *pxTopOfStack = ( StackType_t ) ( usAddress & ( uint16_t )
    0x00ff );
    pxTopOfStack--;

    /* Der AVR ATmega2560 verfügt über 256 KByte
    *Programmspeicher und einen 17-Bit-Programmzähler.
    *Wenn eine Code-Adresse im Stack gespeichert wird,
    *benötigt sie 3 Bytes anstelle von 2 für die anderen
    *ATmega-Chips.
    */
    *pxTopOfStack = 0;
    pxTopOfStack--;

    *pxTopOfStack = ( StackType_t ) 0x00;
    pxTopOfStack--;
    *pxTopOfStack = portFLAGS_INT_ENABLED;
    pxTopOfStack--;

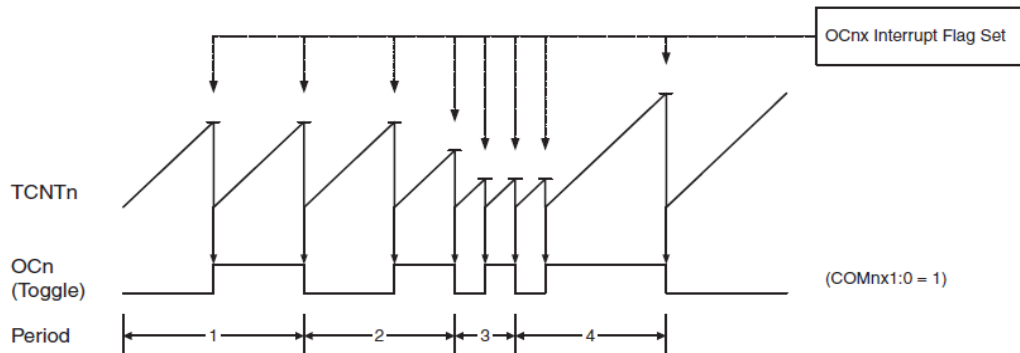
    /*das EIND-Register muss gespeichert werden, wenn einer
    *ATmega256x verwendet wird. Es ist ratsam, den
    *Standardwert auf null zu setzen.
    */
    *pxTopOfStack = ( StackType_t ) 0x00;    /*EIND */
    pxTopOfStack--;

    /* Das RAMPZ-Register muss ebenfalls gespeichert werden.
    *Der Standardwert sollte auf 0 gesetzt werden.
    */
    *pxTopOfStack = ( StackType_t ) 0x00;    /*RAMPZ*/
    pxTopOfStack--;
    #endif
    .
    .
    .
}
```

Lisitng 3: pxPortInitialiseStack() - Anpassungen

Der zweite Unterschied besteht in der Konfiguration des Timers. Für den Scheduler wird im ATmega32-Port der Timer1 im **Clear Timer on Compare Match (CTC)-Mode** konfiguriert.

In diesem Modus wird die Output Compare Unit verwendet. Dabei erfolgt bei jeder Änderung des Zählerregister **TCNT** ein Vergleich mit dem Compare Register **OCR**. Die Output Compare Flag wird gesetzt bzw. der Output Compare Interrupt wird ausgelöst und das Zählerregister **TCNT** auf 0 zurückgesetzt, wenn die Werte übereinstimmen. Dadurch muss der Zähler nicht unbedingt den Überlauf erreichen, sondern kann die obere Grenze **TOP** durch den Wert im **OCR** bestimmt werden. [vgl. 28, S. 38]



Für die Konfiguration des Timers werden die folgenden Schritte durchgeführt:

### 1- Timer-CTC-Mode auswählen

Mode	WGMn3	WGMn2 (CTCn)	WGMn1 (PWMn1)	WGMn0 (PWMn0)	Timer/Counter Mode of Operation	TOP	Update of OCRnX at	TOVn Flag Set on
4	0	1	0	0	CTC	OCRnA	Immediate	MAX

Abbildung 9: CTC-Mode-Bits [26, S. 145]

### 2- Prescaler auf 64 einstellen

CSn2	CSn1	CSn0	Description
0	1	1	clk <sub>I/O</sub> /64 (From prescaler)

Abbildung 10: Prescaler-Bits [26, S. 157]

Für diese beiden Schritte werden die folgenden Bits gesetzt:

#### TCCR1A – Timer/Counter 1 Control Register A

Bit (0x80)	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	TCCR1A
Initial Value	0	0	0	0	0	0	0	0	

Abbildung 11: TCCR1A Register [26, S. 154]

#### TCCR1B – Timer/Counter 1 Control Register B

Bit (0x81)	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	TCCR1B
Initial Value	0	0	0	0	0	0	0	0	

Abbildung 12: TCCR1B Register [26, S. 156]

### 3- Interrupt aktivieren

In diesem Schritt zeigt sich der Unterschied zwischen den beiden Mikrocontrollern. Der ATmega2560 besitzt sechs Timer und dementsprechend sechs TIMSK-Register. Im Gegensatz dazu hat der ATmega32 nur drei Timer und ein TIMSK-Register.

#### Timer/Counter Interrupt Mask Register – TIMSK

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Abbildung 13: TIMSK Register [22, S. 82]

#### TIMSK1 – Timer/Counter 1 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6F)	–	–	ICIE1	–	OCIE1C	OCIE1B	OCIE1A	TOIE1	TIMSK1
Read/Write	R	R	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Abbildung 14: TIMSK1 Register [26, S. 161]

Die Timer-Interrupt-Funktion sieht folgendermaßen aus:

```
#define portCLEAR_COUNTER_ON_MATCH ((uint8_t) 0x08)
#define portPRESCALE_64 ((uint8_t) 0x03)
#define portCLOCK_PRESCALER ((uint32_t) 64)
#define portCOMPARE_MATCH_A_INTERRUPT_ENABLE ((uint8_t)0x10 )

void prvSetupTimerInterrupt( void )
{
    uint32_t ulCompareMatch;
    uint8_t ucHighByte, ucLowByte;

    ulCompareMatch = configCPU_CLOCK_HZ / configTICK_RATE_HZ;
    //8000000/1000

    ulCompareMatch /= portCLOCK_PRESCALER; //8000/64

    ulCompareMatch -= ( uint32_t ) 1; //125-1

    ucLowByte =(uint8_t) (ulCompareMatch & (uint32_t)0xff);
    ulCompareMatch >>= 8;
    ucHighByte = (uint8_t) (ulCompareMatch & ( uint32_t) 0xff);
    OCR1AH = ucHighByte;
    OCR1AL = ucLowByte;
    //OCR1A = 124; //TOP

    ucLowByte = portCLEAR_COUNTER_ON_MATCH | portPRESCALE_64;
    //TCCR1B |= (1 << CS11) | (1 << CS10);
    //TCCR1B &= ~(1 << CS12);
    TCCR1B = ucLowByte;
    //TCCR1B |= (1 << WGM12); // CTC Mode
    //TCCR1B &= ~(1 << WGM13 ); // CTC Mode
    //TCCR1A &= ~(1 << WGM10 ); // CTC Mode
    //TCCR1A &= ~(1 << WGM11 ); // CTC Mode
    ucLowByte = TIMSK1;
    ucLowByte |= portCOMPARE_MATCH_A_INTERRUPT_ENABLE;
    TIMSK1 = ucLowByte;
    //TIMSK1 |= (1 << OCIE1A);
    //sei();
}
```

Lisitng 4: Konfiguration der Funktion prvSetupTimerInterrupt()

Der Timer1 steht nach dem Timer 2 und dem Watchdog-Timer in der Interrupt-Vektorliste des ATmega2560. Es wird empfohlen, einen Timer mit hoher Priorität zu verwenden, wie den Watchdog-Timer. [vgl. 26, S. 101]

## 4 Die wichtigsten Funktionen und Makros

### 4.1 Datentypen

Datentyp	Erklärung
<b>BaseType_t</b>	BaseType_t ist der am effizientesten geeignete Datentyp für die Architektur definiert. Normalerweise wird BaseType_t für Rückgabetypen verwendet, die nur einen sehr begrenzten Wertebereich annehmen können, sowie für Booleans vom Typ pdTRUE (logisch 1) / pdFALSE (logisch 0).
<b>TickType_t</b>	TickType_t ist der Datentyp, der verwendet wird, um den Wert des Tick-Zählers zu halten. FreeRTOS konfiguriert eine periodische Unterbrechung namens Tick-Interrupt. Die Anzahl der Tick-Interrupts, die seit dem Start des Schedulers aufgetreten sind, wird als Tick-Zähler bezeichnet. Der Tick-Zähler dient zur Zeitmessung.
<b>TaskHandle_t</b>	Ein Handle kann genutzt werden, um auf einen Task zu verweisen, beispielsweise um die Priorität der Task zu ändern oder den Task zu löschen.

[vgl.1, S. 21-22, 52]

### 4.2 Tasks

Jeder Task ist ein kleines Programm. Sie beginnt an einem Punkt und wird normalerweise in einer Endlosschleife kontinuierlich ausgeführt und sollte nicht beendet werden. FreeRTOS-Tasks dürfen nicht aus ihrer Implementierungsfunktion zurückkehren. Dies bedeutet, dass sie keine „return“-Anweisung enthalten und nicht über das Ende der Funktion hinaus ausgeführt werden können. Wenn ein Task nicht mehr benötigt wird, sollte sie gelöscht werden. [vgl.1, S. 46]

Der Task-Prototyp muss "void" zurückgeben und einen "void"-Pointer haben.

```
void ATaskFunction (void *pvParameters);
```

*Lisitng 5: Task-Prototyp [1, S. 46]*

Tasks werden mithilfe der FreeRTOS-Funktion `xTaskCreate()` erstellt. [vgl.1, S. 49]

```
BaseType_t xTaskCreate(    TaskFunction_t pvTaskCode,  
                           const char * const pcName,  
                           uint16_t usStackDepth,  
                           void *pvParameters,  
                           UBaseType_t uxPriority,  
                           TaskHandle_t *pxCreatedTask );
```

*Lisitng 6: xTaskCreate() Functoin [vgl. 1, S. 49]*

Parametername	Erklärung
<b>pvTaskCode</b>	Der Parameter pvTaskCode ist einfach ein Pointer auf den Task.
<b>pcName</b>	Eine Bezeichnung für den Task. Er wird als Hilfe für die Fehlersuche (Debugging) hinzugefügt.
<b>usStackDepth</b>	Bei der Erstellung erhält jeder Task einen eigenen Stack. Der Kernel wird durch den Wert usStackDepth informiert, wie groß der Stack sein soll. Zum Beispiel, wenn usStackDepth mit 100 übergeben wird und der Stack 32 Bits breit ist, werden 400 Bytes Speicherplatz für den Stack allokiert (100 * 4 Bytes).
<b>*pvParameters</b>	Task-Funktion, die einen Parameter als void-Pointer (void*) akzeptieren kann.
<b>uxPriority</b>	gibt die Priorität an, auf der der Task ausgeführt wird.
<b>pxCreatedTask</b>	pxCreatedTask kann verwendet werden, um eine Referenz auf den erstellten Task zu geben.
<b>Rückgabewert:</b>	Es gibt zwei mögliche Rückgabewerte: -pdPASS: zeigt an, dass der Task erfolgreich erstellt wurde. -pdFAIL: zeigt an, dass der Task nicht erstellt werden konnte, da nicht ausreichend Heap-Speicher vorhanden ist, damit FreeRTOS genügend RAM reservieren kann, um die Datenstrukturen und den Stack des Tasks aufzunehmen.

[vgl. 1, S. 49-52]

Ein Programm kann mehrere Tasks enthalten. Falls der Prozessor nur einen einzigen Kern hat, kann er nur einen Task gleichzeitig ausführen. Es gibt vier Zustände, in denen sich ein Task befinden kann: im „Running“-Zustand, im „Ready“-Zustand, im „Blocked“-Zustand und im „Suspended“-Zustand. [vgl. 1, S. 65-67]

- **Running-Zustand**  
Wenn ein Task sich im „Running“-Zustand befindet, führt der Prozessor den Code dieser Task aus. [vgl. 1, S. 48]
- **Blocked-Zustand**  
Tasks werden in den „Blocked“-Zustand gewechselt, um auf zeitliche Ereignisse oder das Erreichen einer bestimmten absoluten Zeit zu warten [vgl. 1, S. 65]
- **Suspendend-Zustand**  
Tasks im „Suspended“-Zustand stehen dem Scheduler nicht zur Verfügung, bis **vTaskResume()** aufgerufen wird. [vgl. 1, S. 66]
- **Ready-Zustand**  
Tasks, die sich weder im „Blocked“- noch im „Suspended“-Zustand befinden, werden als im „Ready“-Zustand betrachtet. Sie stehen bereit zur Ausführung. [vgl. 1, S. 66]

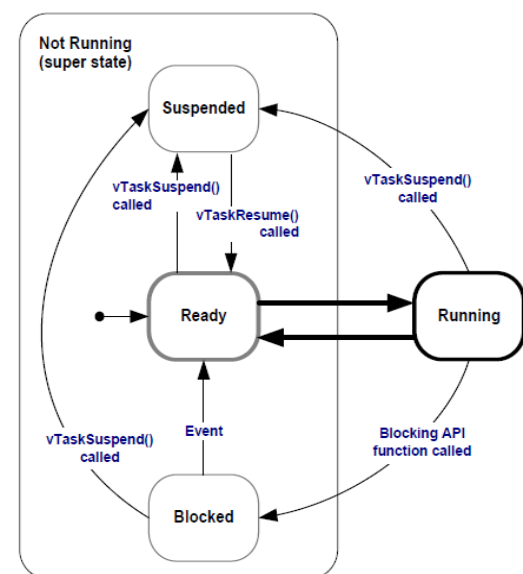


Abbildung 15: Tasks-Zustände [1]

Damit der Scheduler zwischen den Tasks wechseln kann, erfolgt ein abwechselnder Übergang von den Tasks zwischen den Zuständen „Blocked“, „Ready“ und „Running“ für festgelegte Zeitspannen. Dieser Wechsel wird durch einen periodischen Interrupt namens „Tick Interrupt“ gesteuert. Die Frequenz des „Tick-Interrupts“ ist in Hertz(Hz) angegeben und wird vom Programmierer durch das Makro `configTICK_RATE_HZ` in der `FreeRTOSConfig.h`-Datei konfiguriert. [vgl. 1, S. 61]

Die Funktion `vTaskDelay()` wird benutzt, um die Task, in der die Funktion aufgerufen wird, für eine festgelegte Anzahl von Tick-Interrupts in den „Blocked“-Zustand zu versetzen. Sie steht nur zur Verfügung, wenn das Makro `INCLUDE_vTaskDelay` in der `FreeRTOSConfig.h`-Datei auf 1 gesetzt ist. [vgl. 1, S.67]

```
void vTaskDelay( TickType_t xTicksToDelay );
```

*Lisitng 7: vTaskDelay()- Funktion [1, S. 68]*

<b>xTicksToDelay:</b>	Die Anzahl der Tick-Interrupts, während derer der aufrufende Task im „Blocked“-Zustand bleibt, bevor sie wieder in den „Ready“-Zustand gewechselt wird.
-----------------------	---

[vgl. 1, S. 68]

Mithilfe des Makros `pdMS_TO_TICKS()` lässt sich eine zeitliche Angabe in Millisekunden in Ticks umwandeln, ohne die festgelegte Frequenz des `configTICK_RATE_HZ`-Makros zu berücksichtigen. [vgl. 1, S. 68]

In `vTaskDelay()` ist die Zeit, zu der die Task den „Blocked“-Zustand verlässt, relativ zur Zeit des Aufrufs von `vTaskDelay()`. Bei `vTaskDelayUntil()` geben die Parameter hingegen den exakten Tick-Zählwert an, zu dem die aufrufende Task vom „Blocked“-Zustand in den „Ready“-Zustand versetzt werden soll. [vgl. 1, S. 71]

```
void vTaskDelayUntil(TickType_t * pxPreviousWakeTime, TickType_t xTimeIncrement);
```

*Lisitng 8: vTaskDelayUntil()- Funktion [1, S. 72]*

<b>pxPreviousWakeTime</b>	enthält die Zeit, zu der die der Task zuletzt den „Blocked“-Zustand verlassen hat.
<b>xTimeIncrement</b>	xTimeIncrement ist in „Ticks“ angegeben. Das Makro <code>pdMS_TO_TICKS()</code> kann hier wieder verwendet werden.

[vgl. 1, S. 72]

### 4.3 Queues

Die Funktionsweise der Queues wurde bereits vorher erläutert. Sie werden durch Handles referenziert, die Variablen vom Typ `QueueHandle_t` sind. Die Funktion `xQueueCreate()` erstellt eine Queue und gibt ein `QueueHandle_t` zurück, das auf die erstellte Queue verweist. Beim Erstellen einer Queue allokiert FreeRTOS RAM aus dem FreeRTOS-Heap. Dieser RAM wird sowohl für die Verwaltung der Queue-Datenstrukturen als auch für die Elemente, die sich in der Queue befinden, genutzt.

[vgl.1, S. 109]

Die folgende Abbildung soll die Funktionsweise der Queues ausführlich erläutert.

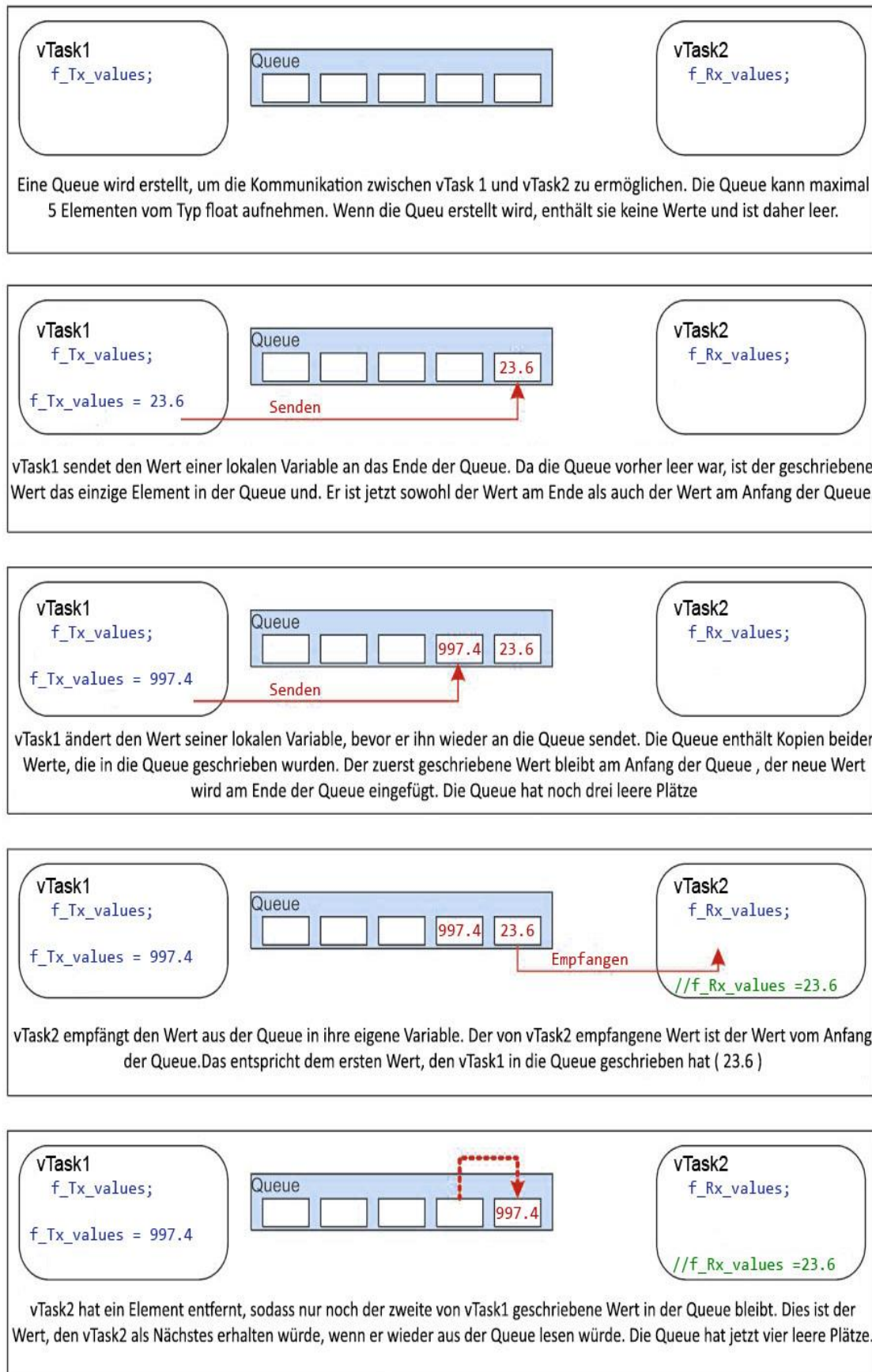


Abbildung 16: Queues-Funktionsweise [1, S. 105]



In diesem Projekt werden drei wesentliche Funktionen zur Verwaltung von Queues verwendet, um die erfassten Daten von dem ersten Task an den zweiten Task zu senden.

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

*Lisitng 9: xQueueCreate()-Funktion [1, S. 109]*

<pre>BaseType_t xQueueSendToBack( QueueHandle_t xQueue, const void * pvItemToQueue, TickType_t xTicksToWait );</pre> <p><i>Lisitng 10: xQueueSendToBack()-Funktion [1, 110]</i></p>	<pre>BaseType_t xQueueReceive( QueueHandle_t xQueue, const void * pvBuffer, TickType_t xTicksToWait );</pre> <p><i>Lisitng 11: xQueueReceive()-Funktion [1, S. 113]</i></p>
<b>uxQueueLength</b>	Die maximale Anzahl von Elementen, die die erstellte Queue gleichzeitig halten kann.
<b>uxItemSize</b>	Die Größe in Bytes für jedes Datenelement, das in der Queue gespeichert werden kann.
<b>xQueue</b>	Der Handle der Queue, zu der die Daten gesendet oder empfangen werden.
<b>pvItemToQueue</b>	Ein Pointer auf die Daten, die in die Queue kopiert werden sollen.
<b>pvBuffer</b>	Ein Pointer auf den Speicher, in den die empfangenen Daten kopiert werden sollen.
<b>xTicksToWait</b>	Die maximale Dauer, für die der Task im „Blocked“-Zustand bleiben sollte, um auf freien Platz zu warten (beim Senden) oder darauf, dass Daten in der Queue verfügbar werden (beim Empfangen), falls die Warteschlange bereits voll ist.

[vgl. 1, S. 109-115]

## 4.4 Mutexe

Die Funktionsweise der Mutexe wurde bereits erläutert. Sie sind nur verfügbar, wenn das Makro `configUSE_MUTEXES` in der `FreeRTOSConfig.h`-Datei auf 1 gesetzt ist. [vgl. 1, S. 244]

Um Mutexe zu verwenden, sind drei wichtige Funktionen erforderlich.

```
SemaphoreHandle_t xSemaphoreCreateMutex(void);
```

*Lisitng 12: xSemaphoreCreateMutex()-Funktion [1, S. 246]*

-Wenn NULL zurückgegeben wird, konnte der Mutex nicht erstellt werden, da nicht genügend Heap-Speicher verfügbar ist, um den Mutex zu erstellen.

-Wenn non-NULL zurückgegeben wird, heißt das, dass der Mutex erfolgreich erstellt wurde.

<pre>BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait );</pre> <p><i>Lisitng 13: xSemaphoreTake()-Funktion [1, S. 196]</i></p>	<pre>BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore);</pre> <p><i>Lisitng 14: xSemaphoreGive()-Funktion [1, S. 197]</i></p>
<b>xSemaphore</b>	Der Semaphor, der genommen oder für die nächste Task abgegeben werden sollte. In diesem Fall der Mutex.
<b>xTicksToWait</b>	Die maximale Dauer, für die der Task im „Blocked“-Zustand bleiben sollte, um auf der Semaphor zu warten, wenn er noch nicht verfügbar ist. Falls <code>configTICK_TYPE_WIDTH_IN_BITS</code> in <code>FreeRTOSConfig.h</code> ist <code>TICK_TYPE_WIDTH_16_BITS</code> , dann beträgt die maximale Dauer ( <code>portMAX_DELAY</code> ) 0xffff.

[vgl. 1, S. 196-199]



Die folgende Abbildung erklärt ausführlicher, wie die Mutexe funktionieren.

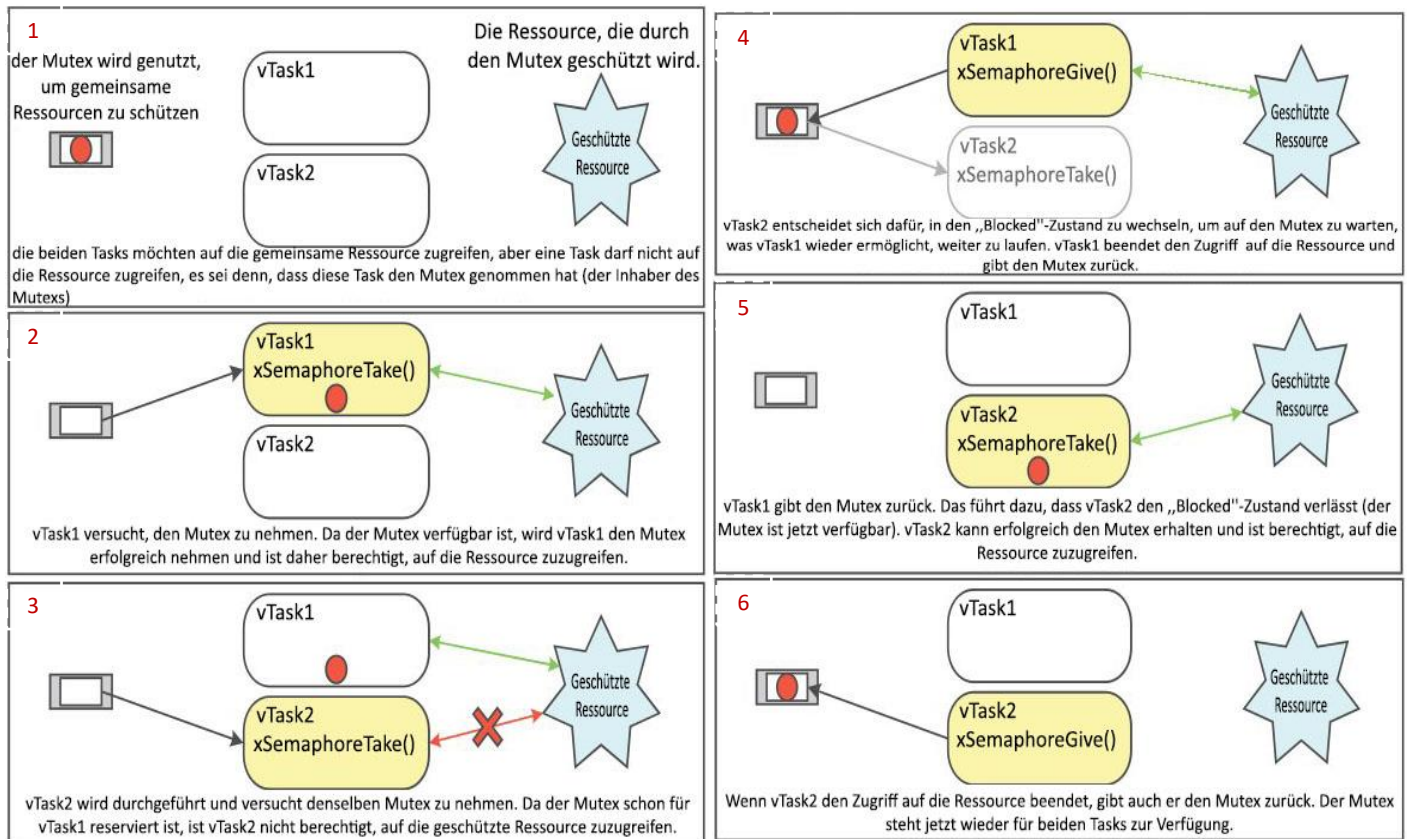


Abbildung 17: Mutexe-Funktionsweise [1, S. 245]

## 5 Projektaufbau

Das Projekt ist so aufgebaut, dass die Daten im ersten Task erfasst und dann im zweiten Task auf dem LCD dargestellt werden müssen. Dazu muss zwischen dem Controller und dem Sensor eine Kommunikationsschnittstelle implementiert werden.

### 5.1 Verwendung von SPI-Schnittstelle

#### 5.1.1 Initialisierung der SPI-Schnittstelle

Vor Beginn der Übertragung wird das CS-Signal auf logisch 1 gesetzt und dann auf 0 gesetzt, um den gewünschten Slave auszuwählen. Das CS-Signal bleibt während der gesamten Übertragung aktiv und wird erst nach Abschluss der Kommunikation wieder auf logisch 1 gesetzt. [vgl. 26, S. 191]

Die SPI-Schnittstelle arbeitet im Master-Modus mit einer maximalen Taktrate von  $\frac{f_{ocs}}{2}$ . Die Taktrate bestimmt die Geschwindigkeit der Datenübertragung zwischen dem Master und dem Slave. [vgl. 26, S. 198]

Laut der Angaben im Datenblatt des Sensors erfolgt die Übertragung mit dem Most Significant Bit (**MSB**) zuerst, und die Clock-Polarity (**CPOL**) und Clock-Phase (**CPHA**) müssen auf 1 gesetzt werden, um eine fehlerfreie Kommunikation sicherzustellen. [vgl. 30, S. 28]

Um diese Bedingungen zu erfüllen, müssen die folgenden Bits gesetzt werden.

SPI2X	SPR1	SPR0	SCK Frequency
0	0	1	$f_{osc}/16$

Abbildung 18: Prescaler-Bits [26, S. 198]

## 21.2.2 SPSR – SPI Status Register

Bit	7	6	5	4	3	2	1	0	
0x2D (0x4D)	SPIF	WCOL	–	–	–	–	–	SPI2X	SPSR
Read/Write	R	R	R	R	R	R	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Abbildung 19: SPSR-Register [26, S. 198]

## 21.2.1 SPCR – SPI Control Register

Bit	7	6	5	4	3	2	1	0	
0x2C (0x4C)	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Abbildung 20: SPCR-Register [26, S. 197]

```

void SPI_Init(void)
{
    DDRB |= (1<<PINB2) | (1<<PINB1) | (1<<PINB0);    /* MOSI, SCK, CS als Ausgänge*/
    DDRB &= ~(1<<PINB3);                               /* MISO als Eingang*/
    PORTB |= CS;                                       /* CS-pin auf 1 setzen*/

    /*SPI in Master-mode*/
    SPCR |= (1<<SPE) | (1<<MSTR) | (1<<CPOL) | (1<<SPR0) | (1<<CPHA);
    SPCR &= ~(1<<SPR1);                               /* F_osc/16 */
    SPCR &= ~(1<<DORD);                               /* Übertragung mit MSB beginnen*/
    SPSR &= ~(1<<SPI2X);                             /* Speed-Doubler deaktivieren*/
}

```

Lisitng 15: Initialisierung der SPI-Schnittstelle

## 5.1.2 SPI-Schreibprotokoll

Um die Daten vom Slave zu schreiben, sendet der Master 2 Bytes an ihn. Nachdem das CS-Signal auf logisch 0 gesetzt wurde, überträgt der Master zunächst über (**MOSI**) die Registeradresse (AD6 – AD0) mit dem R/W-Bit auf logisch 0, gefolgt vom gewünschten Inhalt des Registers (DI7 – DI0). Schließlich wird das CS-Signal wieder auf logisch 1 gesetzt, um die Transmission zu beenden.  
[vgl. 24, S. 30]

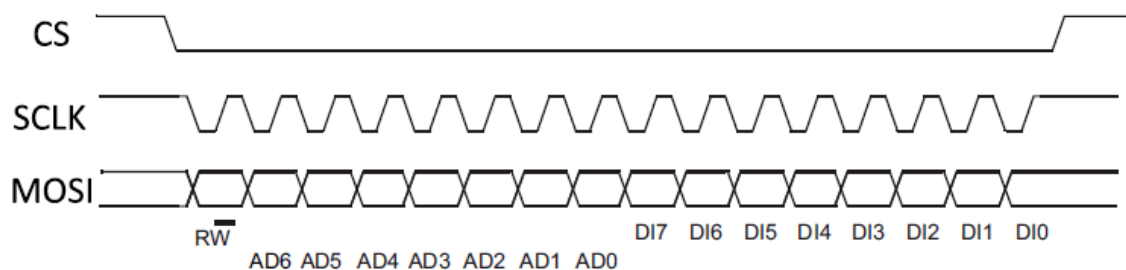


Abbildung 21: SPI-Schreibprotokoll [30, S. 30]

### 5.1.2 SPI-Lese-Protokoll

Um die Daten vom Slave zu lesen, sendet der Master zunächst 1 Byte an ihn. Nachdem das CS-Signal auf logisch 0 gesetzt wurde, überträgt der Master über (**MOSI**) die Registeradresse (AD6 – AD0) mit dem R/W-Bit auf logisch 1. Anschließend wird über (**MISO**) der zu lesende Registerinhalt gesendet. Schließlich wird das CS-Signal wieder auf logisch 1 gesetzt, um die Transmission zu beenden.  
[vgl. 24, S. 32]

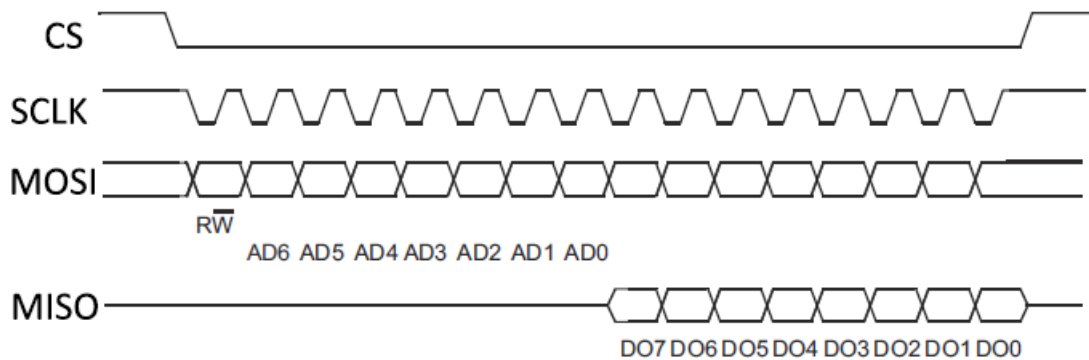


Abbildung 22: SPI-Lese-Protokoll [30, S. 29]

```
void SPI_MasterTransmit(uint8_t cReg, uint8_t
cData)
{
    PORTB &= ~CS;    /* CS-Pin aktivieren*/

    /* Transmission starten mit der Registeradresse*/
    SPDR= cReg | SPI_WRITE;

    /*warten bis die Transmission fertig */
    while(!(SPSR & (1<<SPIF)));

    /*Fortsetzung der Übertragung mit dem
    Registerinhalt */
    SPDR=cData;

    /*warten bis die Transmission fertig */
    while(!(SPSR & (1<<SPIF)));
    PORTB |= CS;    /*CS-Pin deaktivieren*/
}
```

Lisitng 16: SPI\_MasterTransmit() - Funktion

```
uint8_t SPI_MasterReceive(uint8_t cReg)
{
    uint8_t ret;

    PORTB &= ~CS;    /* CS-Pin aktivieren*/

    /*Transmission starten*/
    SPDR= cReg | SPI_READ;

    /*warten bis die Transmission fertig*/
    while(!(SPSR & (1<<SPIF)));

    /*Fortsetzung der Übertragung mit einigen Daten */
    SPDR=0xFF;

    /* warten bis die Transmission fertig */
    while(!(SPSR & (1<<SPIF)));

    ret= SPDR;
    PORTB |= CS; /*CS-Pin deaktivieren*/

    return ret; //Rückgabe der empfangenen Daten.
}
```

Lisitng 17: SPI\_MasterReceive() - Funktion

die Temperatur- und Druckdaten können jetzt abgerufen werden.

```
float read_Pressure()
{
    uint8_t values[3];

    SPI_MasterTransmit(CTRL_REG1,CONFIG);
    values[0] = SPI_MasterReceive(PRESS_OUT_XL);
    values[1] = SPI_MasterReceive(PRESS_OUT_L);
    values[2] = SPI_MasterReceive(PRESS_OUT_H);

    return ((float)((((uint32_t)values[2]<< 16) +
    ((uint16_t)values[1]<<8) + (values[0])))/4096.0f);
}
```

Lisitng 18: read\_Pressure() - Funktion

```
float read_Temperature()
{
    uint8_t values[2];

    SPI_MasterTransmit(CTRL_REG1,CONFIG);
    values[0] = SPI_MasterReceive(TEMP_OUT_L);
    values[1] = SPI_MasterReceive(TEMP_OUT_H);

    return ((float)((((uint16_t)values[1]<<8) +
    (values[0])))/100.0f);
}
```

Lisitng 19: read\_Temperature() - Funktion

## 5.2 Sensorintegration in FreeRTOS

### 5.1.2 Implementierung von Tasks

```
float f_values[2];
int main( void )
{
    char buffer[16];
    LCD_Init();
    SPI_Init();
    xTaskCreate( vTask1, "Read_Task", 1000, NULL, 1, NULL );
    xTaskCreate( vTask2, "Write_Task", 1000, NULL, 1, NULL );
    .
    vTaskStartScheduler();
    .
}
```

*Lisitng 20: Verwendung von xTaskCreate() - Funktion*

```
void vTask1( void *pvParameters )
{
    TickType_t xLastWakeTime = xTaskGetTickCount();
    /*aktuelle Tickanzahl*/

    while(1)
    {
        f_values[0] = read_Pressure();
        f_values[1] = read_Temperature();

        vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(
            100UL ));
    }
}
```

*Lisitng 21: Erstellung von vTask1*

```
void vTask2( void *pvParameters )
{
    TickType_t xLastWakeTime = xTaskGetTickCount();
    char buffer[16];

    while(1)
    {
        LCD_Cursor_Position(0,1);

        // Pressure
        LCD_Print("p=");
        LCD_Print(ftoa(f_values[0],1,buffer));
        LCD_Print(" hPa");

        // Temperature
        LCD_Cursor_Position(0,2);
        LCD_Print("T=");
        LCD_Print(ftoa(f_values[1],1,buffer));
        LCD_Print("C ");

        vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS( 100UL
            ));
    }
}
```

*Lisitng 22: Erstellung von vTask2*

### 5.1.3 Implementierung vom Mutex

```
SemaphoreHandle_t xMutex;
int main( void )
{
    .
    xMutex = xSemaphoreCreateMutex();
    .
    vTaskStartScheduler();
    .
}
```

*Lisitng 23: Verwendung von xSemaphoreCreateMutex() - Funktion*

```
void vTask1( void *pvParameters )
{
    while(1)
    {
        if(xSemaphoreTake(xMutex, portMAX_DELAY) == 1)
        {
            .
            .
            xSemaphoreGive(xMutex);
        }
    }
}
```

*Lisitng 24: Mutex - vTask1*

```
void vTask2( void *pvParameters )
{
    while(1)
    {
        if(xSemaphoreTake(xMutex, portMAX_DELAY) == 1)
        {
            .
            .
            xSemaphoreGive(xMutex);
        }
    }
}
```

*Lisitng 25: Mutex - vTask2*

### 5.1.4 Implementierung vom Queue

```
QueueHandle_t xQueue;
```

```
int main( void )
```

```
{
```

```
·
```

```
xQueue = xQueueCreate (2,sizeof(float));
```

```
if( xQueue != NULL )
```

```
/*ob der Queue erfolgreich erstellt wurde*/
```

```
{
```

```
xTaskCreate( vTask1, "Read_Task", 1000, NULL, 2, &Task1Handle );
```

```
xTaskCreate( vTask2, "Write_Task", 1000, NULL, 1, &Task2Handle );
```

```
}
```

```
·
```

```
vTaskStartScheduler();
```

```
·
```

```
}
```

*Lisitng 26: Verwendung von xQueueCreate() - Funktion*

```
void vTask1( void *pvParameters )
```

```
{
```

```
TickType_t xLastWakeTime = xTaskGetTickCount();
```

```
float f_Tx_values[2];
```

```
while(1)
```

```
{
```

```
f_Tx_values[0] = read_Pressure();
```

```
f_Tx_values[1] = read_Temperature();
```

```
xQueueSendToBack(xQueue, &f_Tx_values[0], 0);
```

```
xQueueSendToBack(xQueue, &f_Tx_values[1], 0);
```

```
vTaskDelayUntil(&xLastWakeTime,
```

```
pdMS_TO_TICKS(100));
```

```
}
```

```
}
```

*Lisitng 27: Queue - vTask1*

```
void vTask2( void *pvParameters )
```

```
{
```

```
char buffer[16];
```

```
float f_Rx_values[2];
```

```
while(1)
```

```
{
```

```
if (uxQueueMessagesWaiting( xQueue ) != 0)
```

```
{
```

```
/*ob der Queue leer ist: Error: nichts empfangen*/
```

```
}
```

```
xQueueReceive(xQueue, &f_Rx_values[0],
```

```
pdMS_TO_TICKS(100));
```

```
xQueueReceive(xQueue, &f_Rx_values[1],
```

```
pdMS_TO_TICKS(100));
```

```
·
```

```
LCD_Print(ftoa(f_Rx_values[0],1,buffer));
```

```
·
```

```
LCD_Print(ftoa(f_Rx_values[1],1,buffer));
```

```
·
```

```
}
```

```
}
```

*Lisitng 28: Queue - vTask2*

## 6 Ergebnisse und Fazit

Das Hauptziel dieser wissenschaftlichen Arbeit war die erfolgreiche Implementierung von FreeRTOS und die vertiefte Auseinandersetzung mit dem **port.c**. Der Prozess dieser Implementierung führte zu einer Reihe von Herausforderungen und Schwierigkeiten, die jedoch im Verlauf der Arbeit erfolgreich bewältigt wurden.

Die Anwendung von Mutexen spielte eine entscheidende Rolle bei der Erreichung von Thread-Sicherheit. Die beiden Tasks wurden erfolgreich implementiert, wodurch Konflikte und Dateninkonsistenzen vermieden wurden.

Abschließend lässt sich feststellen, dass die Implementierung von FreeRTOS in diesem Projekt nicht nur die Zielsetzung erreicht hat, sondern eröffnet auch neue Perspektiven für die Entwicklung zukünftiger Anwendungen im Bereich der Embedded Systems und in Echtzeitbetriebssysteme.

# Literaturverzeichnis

---

- [1] FreeRTOS Books. (o. D.). *Mastering the FreeRTOS™ Real Time Kernel*. FreeRTOS.
- [2] Prof. Jürgen Plate. (2003, 02, Dezember). *Einführung in Betriebssysteme*. TU-Chemnitz.  
<https://www.tu-chemnitz.de/informatik/friz/Grundl-Inf/Betriebssysteme/Script/index.html>
- [3] Prof. Dr. Hellberg. (2016, 22, November). *Betriebssysteme*. FHDW.  
<https://silo.tips/download/betriebssysteme-vorlesungsskript-prof-dr-hellberg>
- [4] National Instruments. (2022, 16, Dezember). *Was versteht man unter einem Echtzeitbetriebssystem?*  
<https://www.ni.com/de/shop/data-acquisition-and-control/add-ons-for-data-acquisition-and-control/what-is-labview-real-time-module/what-is-a-real-time-operating-system--rtos--.html>
- [5] Wikipedia. (2021, 27. Oktober). *FreeRTOS*. Abgerufen am 15. Januar 2024.  
<https://de.wikipedia.org/wiki/FreeRTOS>
- [6] AWS Documentation. (o. D.). *Grundlagen zum FreeRTOS-Kernel*.  
[https://docs.aws.amazon.com/de\\_de/freertos/latest/userguide/dev-guide-freertos-kernel.html](https://docs.aws.amazon.com/de_de/freertos/latest/userguide/dev-guide-freertos-kernel.html)
- [7] ITWissen.info. (2019, 27, Februar). *SPI-Bus*.  
<https://www.itwissen.info/SPI-Bus-serial-peripheral-interface-SPI.html>
- [8] Piyu Dhaker. (2021, 08, März). *So funktioniert das Serial Peripheral Interface*. Elektronikpraxis.  
<https://www.elektronikpraxis.de/so-funktioniert-das-serial-peripheral-interface-a-e33ec4bff798375112b4d7ea81fc0d1f/>
- [9] Wikipedia. (2024, 10. Januar). *Serial Peripheral Interface*. Abgerufen am 19. Januar 2024.  
[https://de.wikipedia.org/wiki/Serial\\_Peripheral\\_Interface](https://de.wikipedia.org/wiki/Serial_Peripheral_Interface)
- [10] Wikipedia. (2024, 12. Januar). *Kontextwechsel*. Abgerufen am 17. Januar 2024.  
<https://de.wikipedia.org/wiki/Kontextwechsel>
- [11] BMU Verlag. (o. D.). *Multithreading*.  
<https://bmu-verlag.de/multithreading/>
- [12] GeeksforGeeks. (2023, 11, Dezember). *Context Switching in Operating System*.  
<https://www.geeksforgeeks.org/context-switch-in-operating-system/>
- [13] Peter Thömmes. (2003, 09, Oktober). *Notizen zu C++*. Springer Berlin Heidelberg.
- [14] Prof. Dr. Thomas Linkugle. (2021/2022). *Mikroprozessortechnik\_Vorlesung 6*.
- [15] Peter Sobe. (o. D.). *Dynamischer Speicher*. HTW Dresden.  
[https://www2.htw-dresden.de/~sobe/Prog1\\_Jg17/Vo/9\\_Dynamischer\\_Speicher.pdf](https://www2.htw-dresden.de/~sobe/Prog1_Jg17/Vo/9_Dynamischer_Speicher.pdf)

- [16] Stefan Brass. (2006). *Objekt-orientierte Programmierung*. Universität Halle.  
[https://users.informatik.uni-halle.de/~brass/oop06/db\\_dymem.pdf](https://users.informatik.uni-halle.de/~brass/oop06/db_dymem.pdf)
- [17] Wikipedia. (2023, 09. Dezember). *Stapelspeicher*. Abgerufen am 22. Januar 2024.  
<https://de.wikipedia.org/wiki/Stapelspeicher>
- [18] Daniel Kreiter. (2023, 23. November). *Was ist Heap and Stack? Grundlagen für Programmierer*. Der Informatikstudent.  
<https://www.derinformatikstudent.de/was-ist-heap-and-stack/#user-content-fn-1%5E>
- [19] Ralf Hartmut Güting, Stefan Dieker. (2018). *Datenstrukturen und Algorithmen*. Springer Vieweg.
- [20] Rishabh Prabhu. (2023, 06. Juni). *Stack Definition & Meaning in DSA*. GeeksforGeeks.  
<https://www.geeksforgeeks.org/stack-meaning-in-dsa/>
- [21] Wikipedia. (2022, 16. Mai). *Heap (Datenstruktur)*. Abgerufen am 22. Januar 2024.  
[https://de.wikipedia.org/wiki/Heap\\_\(Datenstruktur\)](https://de.wikipedia.org/wiki/Heap_(Datenstruktur))
- [22] Microchip Technology. (o. D.). *ATmega32/L Datasheet*.  
<https://ww1.microchip.com/downloads/en/DeviceDoc/doc2503.pdf>
- [23] The Craft of Coding. (2015, 07. Dezember). *Memory in C – the stack, the heap, and static*.  
<https://ww1.microchip.com/downloads/en/DeviceDoc/doc2503.pdf>
- [24] Prof. Dr. Thomas Linkugle. (2023/2024). *Embedded Systems\_Vorlesung 3*.
- [25] feilipu. (2012, 15. Januar). *EtherMega (Arduino Mega 2560) and freeRTOS*.  
<https://feilipu.me/2012/01/15/ethermega-arduino-mega-2560-and-freertos/>
- [26] Microchip Technology. (o. D.). *ATmega640/1280/1281/2560/2561 Datasheet*.  
[https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561\\_datasheet.pdf](https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf)
- [27] Microchip Technology. (o. D.). *AVR® Instruction Set Manual*.  
<https://ww1.microchip.com/downloads/en/DeviceDoc/AVR-InstructionSet-Manual-DS40002198.pdf>
- [28] Prof. Dr. Thomas Linkugle. (2021/2022). *Mikroprozessortechnik\_Vorlesung 7*.
- [29] Ming-Yuan Zhu. (2011, 01. Mai). *Understanding FreeRTOS: A Requirement Analysis*. Kizito Nkurikiyeyezu.  
[https://qiriro.com/ecs6264/static\\_files/references/understanding-freertos.pdf](https://qiriro.com/ecs6264/static_files/references/understanding-freertos.pdf)
- [30] STMicroelectronics. (2017, 29. Juni). *LPS22HB Datasheet*.  
<https://www.st.com/resource/en/datasheet/lps22hb.pdf>

## Anhang

---

- Quellcode-Dateien des entwickelten Sensorprogramms unter Verwendung von FreeRTOS