Ravensburg-Weingarten University

Master Thesis

Implementierung eines Earliest Deadline First Scheduler in FreeRTOS auf einem STM32

Fabian Wicker
Matrikelnummer: 32235

UPDATEN!!!!!!! 25.Januar 2021

HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit „Implementierung eines Earliest First Deadline Scheduler in FreeRTOS“ selbständig angefertigt und mich nicht fremder Hilfe bedient habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichtem und unveröffentlichtem Schriftgut entnommen sind, habe ich als solche kenntlich gemacht.

Unterwaldhausen, den 31. Januar 2021

_____

Fabian Wicker

HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Aufgabenbeschreibung

Ziel dieser Masterarbeit ist es, einen Earliest Deadline First (EDF)-Scheduler in Free Real-Time Operating System (FreeRTOS) auf einem STM32-Mikrocontroller zu integrieren, welcher unter bestimmten Bedingungen im Gegensatz zu anderen Schedulern, eine Central Processing Unit (CPU)-Auslastung von 100% erreichen kann. Die korrekte Funktion des EDF-Schedulers soll anhand von zwei Testanwendungen nachgewiesen werden. Daraus ergeben sich folgende Aufgaben in dieser Masterarbeit:

1. Implementierung des EDF-Schedulers in FreeRTOS

2. Erstellung einer 'Blinky-Demo'

3. Erstellung einer Fast Fourier Transformation (FFT)-Demo

4. Signalgenerator mit Graphical User Interface (GUI) auf einem Host-Computer für die FFT-Demo

HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Inhaltsverzeichnis

HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Abbildungsverzeichnis

Tabellenverzeichnis

Formelverzeichnis

HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Programmcode-Verzeichnis

Abkürzungen

API	Application Programming Interface
CMSIS	Cortex Microcontroller Interface Standard
CLI	Command Line Interface
CPU	Central Processing Unit
DFT	Diskrete Fourier Transformation
DMS	Deadline Monotonic Scheduling
DSP	Digital Signal Processor
EDF	Earliest Deadline First
FFT	Fast Fourier Transformation
FIFO	First Input First Output
FPGA	Field Programmable Gate Array
FreeRTOS	Free Real-Time Operating System
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
i.A.a	in Anlehnung an
IDE	Integrated Development Environment
IoT	Internet of Things
IP	Internet Protocol
IRQ	Interrupt Request
ISR	Interrupt Service Routine
JSON	JavaScript Object Notation

LED	Light Emitting Diode
MQTT	Message Queuing Telemetry Transport
RAM	Random-Access Memory
RMS	Rate Monotonic Scheduling
ROM	Read-Only Memory
RTOS	Real-Time Operating System
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UART	Universal Asynchronous Receiver Transmitter
WCET	Worst Execution Time
WLAN	Wireless Local Area Network

HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Abstract

In den meisten Real-Time Operating System (RTOS) werden fixed-priority pre-emptive Scheduler verwendet, bei denen jeder Prozess eine feste Priorität zugewiesen bekommt. Vorteil dieses Scheduler Prinzips ist die weniger komplexe Implementierung im Kernel und der geringe Overhead. Die Unterstützung mehrerer Schedule-Verfahren im Kernel eines RTOS würde dessen Weiterentwicklung stark beeinflussen. Allerdings gibt es die Möglichkeit ein anderes Scheduling-Verfahren außerhalb des Kernels eines RTOS zu implementieren. Bei einem Scheduling-Verfahren nach dem EDF-Prinzip kann die CPU maximal ausgelastet werden. Außerdem muss die Prozessstruktur und deren Prioritäten nicht vorher festgelegt werden. Aus diesen Gründen besteht das Ziel dieser Arbeit darin, einen EDF-Scheduler im Anwendungsbereich von FreeRTOS zu implementieren. Durch die Entwicklung zweier Demo-Applikationen soll die korrekte Funktion des Schedulers nachgewiesen werden. Dabei dient eine Demo als Vorführung des EDF-Schedulers, bei der alle Status innerhalb und außerhalb der Grenzen des EDF-Schedulers abgefragt werden. Die zweite Demo stellt eine komplexere Anwendung dar, wobei drei unterschiedliche Prozesse Daten verarbeiten und untereinander kommunizieren müssen.
Einleitend werden die Grundlagen für das Verständnis der Arbeit beschrieben, anschließend wird die Implementierung des EDF-Schedulers erläutert. Danach werden die zwei erarbeiten Demo-Applikation vorgestellt und die verschiedenen Status des Schedulers innerhalb der Applikationen.
Der erarbeitete Scheduler soll in Zukunft als Erweiterung für FreeRTOS dienen. Durch die Verwendung der FreeRTOS-Application Programming Interface (API) kann die Portierbarkeit gewährleistet werden. Dadurch können alle Plattformen die FreeRTOS unterstützt, den erarbeiteten Scheduler verwenden. Des Weiteren weisen die erarbeiteten Demo-Applikationen die korrekte Funktion des Schedulers nach.

1 Grundlagen

In diesem Kapitel werden die Grundlagen dieser Arbeit zum aktuellen Stand der Technik vorgestellt. Dabei werden die verwendeten Systeme, sowie auch benötigte Protokolle und Algorithmen erklärt.

1.1 Echtzeitsysteme

Unter Determinismus eines Algorithmus versteht sich, dass dieser zu jedem Zeitpunkt 'nachvollziehbar' ist. Damit ist gemeint, dass ein Algorithmus unter gleichen Eingabewerten immer die gleichen Ausgabewerte als Ergebnis liefert (Hoegel, 2020). Außerdem ist für Echtzeitsysteme unabdingbar zu wissen, wie lange eine Abarbeitung eines Codes benötigt, was bei deterministischem Verhalten eindeutig gegeben ist. Um das Verhalten eines Echtzeitsystems bestimmen und deren Rechtzeitigkeit und Korrektheit einhalten zu können, muss das Verhalten eindeutig im Voraus bestimmt, also deterministisch sein. Echtzeitsysteme sind Systeme, welche diese Anforderungen erfüllen und werden in diversen Technikgebieten, wie z.b. in Signalverarbeitung, Robotik und Steuerungen zur Anwendung gebracht. Da nicht bei allen Systemen eine Einhaltung der vorher definierten Zeitspanne von Nöten ist, wird die Echtzeitanforderungen in drei Kategorien unterteilt:

- **Feste Echtzeitanforderung**: Führt bei einer Überschreitung der vorher definierten Zeitspanne zum Abbruch der Aufgabe.
  - **Beispiel**: Verbindungsaufbau eines Smartphones in das lokale Wireless Local Area Network (WLAN).
- **Weiche Echtzeitanforderung**: Kann die vorher definierte Zeitspanne überschreiten, arbeitet die Aufgabe normalerweise aber schnell genug ab. Die definierte Zeitspanne kann auch als Richtlinie bezeichnet werden, welche aber nicht eingehalten werden muss. Sollte die Zeitspanne überschritten werden, wird die Aufgabe nächstmöglichst bearbeitet, jedoch hat dies keine direkte Auswirkungen auf das Gesamtsystem.
  - **Beispiel**: Druckgeschwindigkeit eines Druckers.
- **Harte Echtzeitanforderung**: Garantiert die Erfüllung der Aufgabe in der vorher definierten Zeitspanne.
  - **Beispiel**: Bremspedal eines Autos oder die Abschaltung der Maschine beim Betreten der Sicherheitszone.

Echtzeit beschreibt somit das zeitliche Ein- bzw. Ausgangsverhalten eines Systems, allerdings nichts über dessen Realisierung. Dies kann je nach Anforderung für das Echtzeitsystem rein auf Software auf einem normalen Computer oder auch als reine Hardware-Lösung implementiert sein. Allerdings werden bei harter Echtzeitanforderung oftmals spezielle Architekturen in Hard- und Software wie z.b. Mikrocontroller-, Field Programmable Gate Array (FPGA)- oder Digital Signal Processor (DSP)-basierte Lösungen verwendet. Ein Prozess, auch in FreeRTOS Task genannt, beschreibt die Abarbeitung von Software, welche eine bestimmten Aufgabe erfüllt (Baumgartl, 2008).

1.2 Scheduler

Als Scheduler (zu Deutsch: Steuerprogramm) wird eine Logik bezeichnet, welche die zeitliche Ausführung von mehreren Prozessen steuert. Diese Logik wird als präemptiver oder kooperativer Scheduler in Betriebssystemen eingesetzt. Der Scheduler ist für die zeitliche Anordnung der Prozesse verantwortlich. Für die Neuanordnung, welche Task CPU-Zeit bekommen soll, löst der Scheduler einen Context Switch, auch Taskwechsel genannt, aus. Die CPU ist für einen Scheduler die einzige Ressource von Relevanz, andere Hardware ist für einen Scheduler nicht relevant, wie z.B. Read-Only Memory (ROM) oder Random-Access Memory (RAM). Bei einem kooperativen Scheduling werden einem Prozess die benötigten Ressourcen übergeben und der Scheduler wartet, bis der Prozess vollständig abgearbeitet wurde. Im Gegensatz zu einem kooperativen Scheduler kann ein präemptiver Scheduler einem Prozess die Ressourcen vor der Fertigstellung entziehen, um zwischenzeitlich diese anderen Prozessen zuzuweisen (Mikrocontroller.net, 2019).

Allgemein lassen sich die unterschiedliche Scheduler-Systeme in drei Rubriken unterteilen (Wikipedia, 2020a):

- **Stapelverarbeitungssysteme**: Zur Anfangszeit der Computer wurden Prozesse von Lochkarten eingelesen und nacheinander abgearbeitet und die Ergebnisse auf einem Magnetband abgespeichert oder ausgedruckt. Dabei bestanden Stapelverarbeitungssysteme oftmals aus mehreren Computern mit unterschiedler Rechenleistung, wobei die preiswerteren Computer mit weniger Rechenleistung Aufgaben wie das Lesen und Speichern übernahmen und die rechenstärkeren Computer die Abarbeitung des Programms vornahmen (Mandl, 2014).

- **Interaktive Systeme**: Eingaben von Benutzern sollten schnellstmöglich abgearbeitet werden, weniger wichtige Aufgaben wie z.B. die Aktualisierung der Uhrzeit werden unterbrochen oder erst im Nachhinein abgearbeitet („Embedded Systems Engineering“, n. d.).

- **Echtzeitsysteme**: Das Echtzeitsystem muss die Fertigstellung des Prozesses innerhalb einer definierten Zeitspanne garantieren, sofern die CPU Auslastung nicht über 100% beträgt („Embedded Systems Engineering“, n. d.).

Für die Anordnung der Prozesse selbst, wann welcher Prozess ausgeführt wird, gibt es verschiedene Arten von Schedulern. In den nachfolgend vorgestellten Scheduler-Verfahren wird bei der Berechnung der Auslastung die Laufzeiten von Taskwechsel und Interrupts vernachlässigt.

1.2.1 Rate Monotonic Scheduling (RMS)

RMS ist ein präemptives Scheduling-Verfahren, welches die Prioritäten einzelner Prozesse nach deren Periodenlänge statisch anordnet. Je niedriger die Periode eines Prozesses ist, je höher ist dessen Priorität. Mit Hilfe von Formel 1 kann die maximale Menge an Prozessen, die mit RMS verarbeitet werden können, berechnet werden. Durch eine zunehmende Anzahl von Prozessen nähert sich die Grenze dem Wert von $ln(2) \approx 0.693$, welches eine maximale Auslastung von 69,3% bedeutet. Die maximale Auslastung bezieht sich dabei auf die CPU-Auslastung, alle anderen Ressourcen wie z.B. Arbeitsspeicher werden als unbegrenzt angesehen. Sofern die maximale Auslastung nicht übertreten wird, verpasst kein Prozess die dazugehörige Deadline (Siemers, 2017).

$$U = \sum_{i=1}^{n} \frac{\Delta e_i}{\Delta d_i} \leq n \cdot \left(\sqrt[n]{2} - 1 \right) \tag{1}$$

U	CPU-Auslastung
Δe_i	Zeitspanne
Δd_i	Deadline
n	Anzahl der Prozesse

Formel 1: RMS Berechnung der CPU-Auslastung (Siemers, 2017)

1.2.2 Deadline Monotonic Scheduling (DMS)

Analog zu RMS arbeitet DMS präemptiv prioritätsbasiert, wobei bei DMS der Prozess mit der kürzesten Deadline die höchste Priorität bekommt. Falls bei der Abarbeitung des aktuellen Prozesses ein neuer Prozess mit einer höheren Priorität eintrifft, wird der aktuelle Prozess unterbrochen und der Prozess mit der höheren Priorität bekommt die Rechenzeit. Die Berechnung der maximalen Auslastung erfolgt auch nach Formel 1, dies führt zur gleichen maximalen Auslastung von 69,3%, bei der keine Deadline überschritten wird (Siemers, 2017).

1.2.3 Earliest Deadline First (EDF)

Bei EDF bekommt die Task mit der am naheliegendsten Deadline CPU-Zeit zugeteilt. Im Unterschied zu DMS wird bei EDF nur am Anfang oder Ende einer Task, die CPU-Zeit zugeteilt bekommen hat, die Anordnung der Tasks aktualisiert. Dadurch wird eine Task, die einmal CPU-Zeit zugeteilt bekommen hat, niemals unterbrochen, auch wenn die Deadline einer neu eingetroffen Task eine näherliegende Deadline besitzt. Auch ist die Anordnung, wie bei RMS und DMS nicht statisch durch fest gesetzte Prioritäten, sondern wird dynamisch am Anfang oder Ende jeder Task neu anhand der Deadlines angeordnet. Ein Context Switch, was ein Wechsel von einem Prozess zu einem anderen Prozess entspricht, findet nur vor Beginn eines Prozesses oder am Ende eines Prozesses statt. In Zuge dieser Masterarbeit wird ein Context Switch nur nach Beendigung eines Prozesses ausgelöst, sofern ein anderer Prozess eine höhere Priorität erhalten hat. Im Gegensatz zu RMS und DMS kann bei EDF der Scheduler die CPU bis auf 100% auslasten, dabei darf die Zeitspanne der einzelnen Prozesse bis zur Deadline nur jeweils größer oder gleich der Periode des Prozesses sein (Siemers, 2017).

$$U = \sum_{i=1}^{n} \frac{\Delta e_i}{\Delta d_i} \tag{2}$$

U	CPU-Auslastung
Δe_i	Zeitspanne
Δd_i	Deadline
n	Anzahl der Prozesse

Formel 2: Berechnung der EDF CPU-Auslastung (Siemers, 2017)

1.3 FreeRTOS

FreeRTOS ist ein Echtzeitbetriebssystem für eingebettete Systeme, welches unter der freizügigen Open-Source Lizenz MIT steht. Außerdem ist der Scheduler des Betriebsystems zwischen präemptiver und kooperativer Betrieb konfigurierbar um verschiedene Einsatzzwecke abzudecken (FreeRTOS, 2020g). FreeRTOS unterstützt über 40 Mikrocontroller-Architekturen, um eine größere Bandbreite zu erreichen (FreeRTOS, 2020a). Für nicht unterstütze Architekturen stellt FreeRTOS eine Portierungs-Anleitung zur Verfügung, welche als Unterstützung bei der Portierung von FreeRTOS auf andere CPUs dient („Creating a New FreeRTOS Port“, 2020). FreeRTOS ist einfach zu implementieren und der Kernel unterstützt Multithreading, Warteschlangen, Semaphore, Software-Timer, Mutexes und Eventgruppen (FreeRTOS, 2020b). Des Weiteren stellt FreeRTOS unter der Bezeichnung ”FreeRTOS Plus”verschiedene Erweiterungen zur Verfügung, welche erweiterte Funktionen bereitstellen (FreeRTOS, 2020d). Folgende Erweiterungen sind zum Stand dieser Arbeit verfügbar:

- **FreeRTOS + TCP**: Socketbasiertes Transmission Control Protocol (TCP)/ UDP/Internet Protocol (IP) Interface.
- **Application Protocols**: Message Queuing Telemetry Transport (MQTT) und Hypertext Transfer Protocol (HTTP) Anwendungsprotokolle für Internet of Things (IoT).
- **coreJson**: Effizienter Parser für JavaScript Object Notation (JSON)
- **corePKCS#11**: Verschlüsselungs API-Layer
- **FreeRTOS + IO**: Erweiterung für Peripheriegeräte
- **FreeRTOS + CLI**: Effiziente Command Line Interface (CLI)-Eingaben

(in Anlehnung an (i.A.a) FreeRTOS, 2020d)

Im Zuge dieser Masterarbeit wurde als einziges der Stack ’FreeRTOS + TCP’ für die Verbindung zwischen Computer und Mikrocontroller verwendet. Wobei die Integration von anderen Erweiterungen keinen Einfluss auf den in dieser Arbeit implementierten EDF-Scheduler hat.

1.3.1 FreeRTOS Scheduling Richtlinie

Der FreeRTOS-Scheduler arbeitet prioritätsbasiert und beinhaltet standardmäßig zwei Eigenschaften:

- **Time Slicing Scheduling Policy**: Tasks mit gleicher Priorität erhalten die gleiche Anteile an CPU-Zeit, dieses Verfahren ist auch als 'Round-Robin Algorithmus' bekannt.

- **Fixed Priority Preemptive Scheduling**: Es wird immer die Task mit der höchsten Priorität ausgeführt. Das bedeutet eine niedriger priorisierte Task bekommt nur CPU-Zeit, wenn die höher priorisierten Tasks nicht den 'ready'-Status besitzen. Teilen sich zwei Tasks gleichzeitig die höchste Priorität, tritt die Time Slicing Scheduling Policy in Kraft.

Um einen oder beide dieser Eigenschaften abzuschalten, kann dies unter der Datei 'FreeRTOSConfig.h', wie in Programmcode 1 dargestellt, abgeändert werden. Je nach gewünschter Anwendung könnten diese Optionen diese negativ oder positiv beeinflussen (Microcontrollerslab, 2021).

```
1    #define configUSE_PREEMPTION                    ( 1 )
2    #define configUSE_TIME_SLICING                  ( 1 )
```

Programmcode 1: FreeRTOS Scheduling Policy Properties

1.3.2 Overhead und Footprint

Die Dauer eines Context Switches, auch Taskwechsel genannt, zwischen zwei Tasks ist abhängig von der Portierung, dem Compiler und der Konfiguration von FreeRTOS. FreeRTOS selbst gibt ein Beispiel anhand einer Cortex-M3 CPU eine Taskwechsel Zeit von 84 CPU Zyklen an (FreeRTOS, 2020e). Typischerweise besitzt ein FreeRTOS Kernel Binary Image eine Größe zwischen 6Byte und 12kByte (FreeRTOS, 2020c).

1.3.3 Task States

In FreeRTOS erzeugte Tasks können vier verschiedene Zustände erreichen. Diese Zustände ermöglichen komplexere Algorithmen zu implementieren und sehen wie folgt aus:

- **Running**: Die Task wird gerade ausgeführt.
- **Ready**: Die Task ist bereit zur Ausführung.
- **Blocked**: Die Task wartet auf ein Event, kann nicht ausgeführt werden.
- **Suspended**: Die Task wurde von der Anwendung deaktiviert.

(FreeRTOS, 2020f)

Die Möglichkeit eine Task in den 'Blocked'-Status zu setzen, ergeben sich viele neue Anwendungsbereiche. Ein Beispiel wäre das Warten der höher priorisierten Task auf Daten, welche von einer niederen priorisierten Task erzeugt werden.

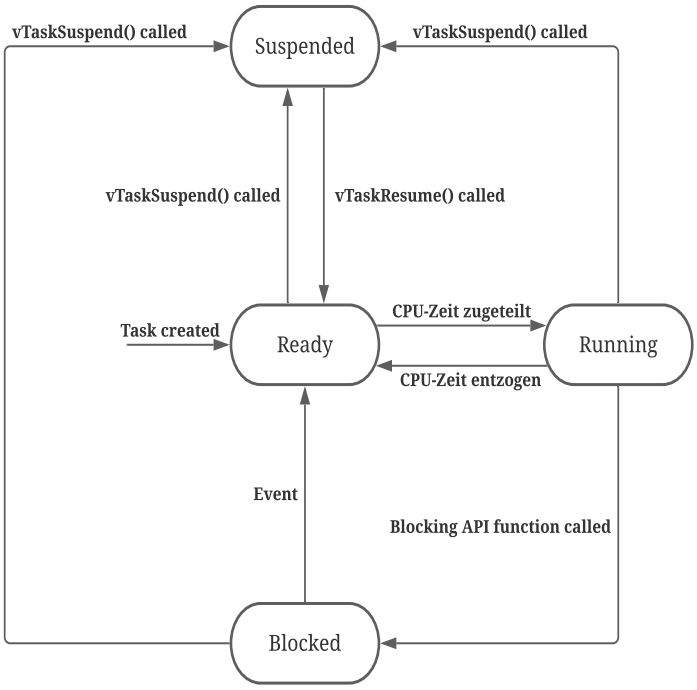


Abbildung 1: FreeRTOS Task States (i.A.a FreeRTOS, 2020f)

Ein Ablaufdiagramm der einzelnen Zustände und deren Wechsel mit Hilfe von FreeRTOS-Funktionen wird in Abbildung 1 dargestellt. Dabei wird erläutert, mit welchen FreeRTOS-Funktionen oder Events den Statuswechsel auslösen (FreeRTOS, 2020f).

1.3.4 Idle Task

Die FreeRTOS Idle Task wird immer dann aufgerufen, wenn keine andere Task ausgeführt werden kann. Dabei wird die FreeRTOS Idle Task automatisch beim Start des FreeRTOS-Schedulers mit der niedrigsten verfügbaren Priorität erstellt, um keine Tasks im 'ready' Status zu blockieren. Das heißt die Idle Task wird nur ausgeführt, wenn es keine Task mit einer höheren Priorität gibt, oder keine Task mit einer höheren Priorität sich im 'ready' Status befindet. Sollte eine Task die gleiche Priorität mit der Idle Task teilen, wird die CPU-Zeit, bei Standardkonfiguration, zwischen den zwei Tasks aufgeteilt. Des Weiteren wird die Idle Task in FreeRTOS Standardkonfiguration mit der Priorität 0, also der niedrigsten Priorität gestartet. Beim Ausführen der FreeRTOS Idle Task wird nicht mehr benötigter zugewiesener Speicher gelöscht und sollte somit regelmäßig CPU-Zeit bekommen, da im schlimmsten Fall der Arbeitsspeicher überlaufen kann. Des Weiteren kann mit der Flag, wie in Programmcode 2 zu sehen, der Idle Task Hook aktiviert werden. Der Idle Task Hook wird bei jedem Aufruf der Idle Task ausgeführt und ermöglicht dem Benutzer dabei einen zusätzlichen Code auszuführen. Dies kann für verschiedene Einsatzbereiche benutzt werden, wie beispielsweise um den Mikrocontroller in den Energiesparmodus zu setzen (FreeRTOS, 2020i).

```
1    #define configUSE_IDLE_HOOK                     ( 1 )
```

Programmcode 2: FreeRTOS Idle Task Hook Flag

1.3.5 Interrupts

Interrupts werden durch ein Ereignis ausgelöst, wie z.B. einem bestimmten Wert eines Timers ausgelöst und führen zur einer Unterbrechung der aktuellen Programmausführung, um in der Regel kurzen, zeitlich kritischen, Vorgang abzuarbeiten. Nach der Unterbrechungsanforderung (Interrupt Request (IRQ)) führt der Prozessor eine Unterbrechungsroutine (Interrupt Service Routine (ISR)) mit erweiterten Privilegien aus. Im Anschluss wird der vorherige Zustand des Prozessors wiederhergestellt und die vorherige Ausführung an der unterbrochenen Stelle fortgesetzt (Mandl, 2014; Wikipedia, 2021).

1.3.6 FreeRTOS SysTick

Der FreeRTOS SysTick, auch System Tick genannt, ist fundemental für die Taskwechsel in FreeRTOS. Dieser Interrupt wird standardmäßig jede Millisekunde aufgerufen. Dabei bietet der Wert eine gute Balance zwischen Taskgeschwindigkeit und Overhead von

Taskwechsel. Bei jedem SysTick wird die FreeRTOS Funktion 'vTaskSwitchContext()' aufgerufen. Die Funktion selbst überprüft ob eine höher priorisierte Task unblocked, also ausführbar (ready) geworden ist und falls dies zutrifft wird ein Taskwechsel auf die höher priorisierte Task durchgeführt (FreeRTOS, 2020h).

1.3.7 FreeRTOS Queues

Queues, zu Deutsch Warteschlangen, werden oft auch außerhalb von FreeRTOS für die sichere Kommunikation zwischen verschiedenen Tasks und/oder Interrupts verwendet. Bei Queues wird oft das Prinzip First Input First Output (FIFO) verwendet, bei dem die Daten, welche zu erst in die Queue gesendet wurden, als erstes weiterverarbeitet werden.

1.4 Periodische Tasks

Anwendungen werden in Echtzeitsystemen oft periodisch ausgeführt. Periodische Tasks setzen sich aus folgenden Parametern zusammen:

$$T_i(t_{\phi,i}, t_{p,i}, t_{e,i}, t_{d,i}) \tag{3}$$

T	Task
t_ϕ	Phase
t_e	Ausführungszeit (Worst Execution Time (WCET))
t_d	(relative) Deadline

Formel 3: Parameter von periodischen Tasks (i.A.a **echtzeit_systeme)**

Mit den in Formel 3 genannten Parametern jeder einzelnen periodischen Task können die verschiedenen Scheduler-Verfahren die Ausführung der einzelnen Tasks planen und umsetzen. Im Zuge der Aufgabenstellung bei dieser Masterarbeit wurden keine Tasks mit einer Phasenverschiebung verlangt, daher kann der Parameter t_ϕ als Null angesehen werden (Baumgartl, 2008).

1.5 User Datagram Protocol (UDP)

Für die Kommunikation zwischen Host-Computer und Mikrocontroller bei der FFT-Demo wurde UDP als Protokoll ausgewählt Das Protokoll erfordert als Hardware-Interface eine Ethernet-Schnittstelle, welche weit verbreitet und daher auf auf vielen Mikrocontroller und Computer verbaut ist. Viele Anwendungen wie z.B. IP-Telefonie oder Livestreams

benutzen UDP als Übertragungsprotokoll. Dadurch können Anwendungen gezielt Daten zu der gewünschten Gegenstelle versenden („UDP - User Datagram Protocol“, 2021). Das UDP-Protokoll selbst erfüllt für diese Arbeit folgende Eigenschaften:

- **Verfügbarkeit**: Ethernet-Schnittstellen sind in vielen Computern und Mikrocontroller verbaut, dadurch ist eine breite Verfügbarkeit dieses Verbindungsart gewährleistet.
- **Latenzarm**: Die gesendeten UDP-Pakete werden vom Empfänger nicht bestätigt. Das heißt der Sender sendet Daten ohne deren Ankunft zu überprüfen.
- **Wenig Overhead**: Das Empfangen und Senden sollte so wenig wie möglich die CPU unterbrechen und somit blockieren.
- **Hohe Übertragungsrate**: Es sollten so viele Daten wie möglich, innerhalb einer bestimmten Zeit übertragen werden können.

1.6 Fast Fourier Transformation (FFT)

FFT berechnet die Fourier Transformierte aus und ist und ist ein Algorithmus der die gleichen Ergebnisse wie die DFT liefert, jedoch mit wesentlich weniger Rechenzeit. Dabei können beide Algorithmen ein zeitdiskretes Signal in dessen Frequenzteile (Spektrum), wie in Abbildung 2 dargestellt, aufspalten. Dies ist sehr nützlich für viele Bereiche der Signalverarbeitung z.B. für das Komprimieren von Musikaufnahmen, wo nicht hörbare Frequenzen entfernt werden.

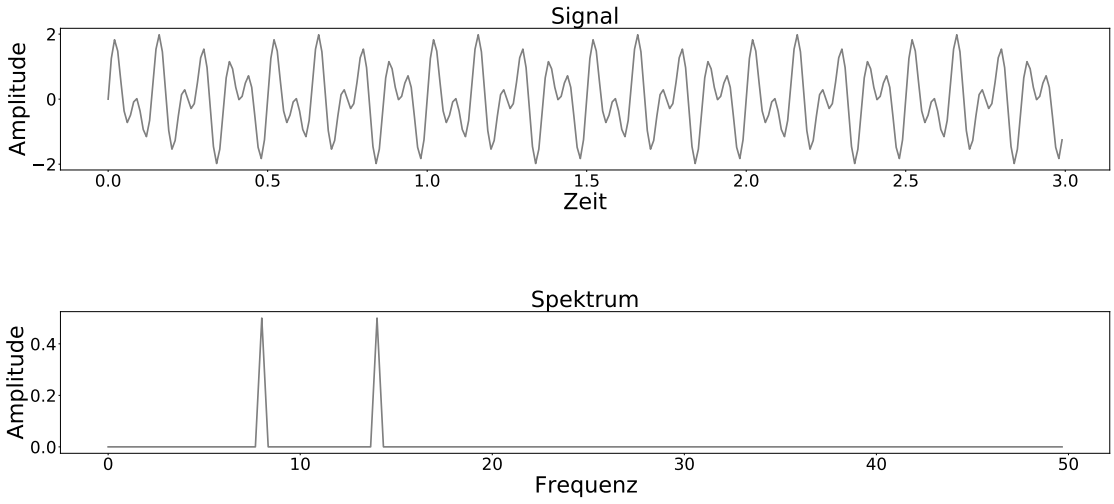


Abbildung 2: Aufspaltung eines Signals in dessen Frequenzteile (eigene Darstellung)

Bei der Berechnung einer DFT steigt dessen Komplexität O mit der Anzahl der Abtastpunkte N quadratisch an, wie in der Formel 4 dargestellt. Dies führt bei hohen Samplegrößen zu einer extremen Rechenzeit, was nicht praktikabel ist (Werner, 2012).

$$O(N) = (N^2) \tag{4}$$

Formel 4: Berechnung der DFT-Komplexität (Werner, 2012)

Im Gegensatz zur DFT stellt die FFT ein Algorithmus dar, mit welcher der Rechenaufwand deutlich reduziert wird. Aus diesem Grund wird häufig die FFT genutzt, welche nach dem 'Teile und Herrsche'-Verfahren funktioniert und nur noch eine Komplexität von $O(N) = (N\,log(N))$ besitzt (Werner, 2012). Durch diese enorme Zeit- und Rechenersparnis fand die FFT in vielen Bereichen, wie z.B. in Ingenieurswissenschaften, Musik, der Wissenschaft und der Mathematik Verwendung (Wikipedia, 2020b). Für die Berechnung der FFT sind mehrere Algorithmen verfügbar, wobei in dieser Masterarbeit der Radix-2 Algorithmus von der Cortex Microcontroller Interface Standard (CMSIS)-DSP Bibliothek verwendet wurde. Als Vorraussetzung des Algorithmus der FFT muss die Anzahl der Messwerte N, auch Samples genannt, einer Zweierpotenz entsprechen (Werner, 2012).

$$N = 2^N, N \in \mathbb{N} \tag{5}$$

N Anzahl von Samples
$\in$ natürliche Zahlen

Formel 5: Rechenbedingung der FFT (Werner, 2012)

Wird die Rechenbedingung in Formel 5 erfüllt, kann, wie in Formel 6 zu sehen, diese in zwei Teilsummen zerlegt werden, je eine für gerade und ungerade Indizes.

$$X(k) = \sum_{n=0}^{N-1} x(n)\, w_N^{n\,k} = \sum_{n=0,2,\ldots}^{N-2} x(n)\, w_N^{n\,k} + \sum_{n=1,3,\ldots}^{N-1} x(n)\, w_N^{n\,k} \tag{6}$$

$w = e^{-j\,\frac{2\,\pi}{N}}$ komplexer Drehfaktor der DFT

Formel 6: Aufteilung der DFT (i.A.a Werner, 2012)

Anschließend wird eine Substitution wie in Tabelle 1 zu sehen, angewandt.

$n = 2m$	für n gerade
$n = 2m + 1$	für n ungerade
$M = N/2$	als Abkürzung

Tabelle 1: Substitution der DFT (i.A.a Werner, 2012)

Unter Berücksichtigung der Umformungen aus Formel 7 resultieren zwei Formeln mit der halben Länge, wie in Formel 8 dargestellt.

$$w_N^{2m\,k} = w_M^{m\,k} \qquad \text{und} \qquad w_N^{(2m+1)\,k} = w_N^k\, w_M^{m\,k} \tag{7}$$

Formel 7: Umformung DFT (i.A.a Werner, 2012)

$$X(k) = \sum_{m=0}^{M-1} x(2m)\, w_M^{m\,k} + \sum_{m=0}^{M-1} x(2m+1)\, w_M^{m\,k} \tag{8}$$

Formel 8: Resultierende Gleichungen nach Aufspaltung der DFT (i.A.a Werner, 2012)

Ist die resultierende Länge der DFT wieder eine Zweierpotenz, wird die Zerlegung solange fortgeführt, bis N Vektoren der Länge 2^0 erhält (Werner, 2012). Anschließend werden die Ergebnisse zusammengerechnet und die DFT wurde gelöst (Werner, 2012).

1.7 Plattform

Als Plattform für dieses Projekt dient ein STM32F769I-Disc0 Board der Firma ST, welcher unter anderem folgende Hauptmerkmale besitzt:

- Prozessor: ARM®Cortex®-M7
- ROM: 2 Mbytes
- RAM: 532 Kbytes
- Schnittstellen:
  - Ethernet-Schnittstelle
  - Zwei Push-Buttons
  - 7 One-Board Light Emitting Diode (LED)s

Dieses ist mit einen ARM®Cortex®-M7 Prozessor ausgestattet und hat für die verwendete Arbeit und erarbeiteten Demo-Applikationen ausreichende Rechenleistung. Für die Kommunikation zwischen Host-Computer und Mikrocontroller wurde die integrierte Ethernet Schnittstelle verwendet. Für die 'Blinky-Demo' wurden außerdem die drei, auf dem Board integrierten, LEDs verwendet.

Abbildung 3: STM32F769I-Disc0 Board (Quelle: Components, 2020)

2 Umsetzung des EDF-Schedulers

In diesem Kapitel werden unter anderem die benötigten Datenstrukturen, die Umsetzung und die Verwendung des in dieser Arbeit erstellten EDF-Schedulers vorgestellt. Für die Implementierung des EDF-Schedulers selbst, wurde die Idee verfolgt, den FreeRTOS-Scheduler weiterzuverwenden und am Ende jeder Task eine Neuanordnung der Prioritäten nachdem EDF-Prinzip auszulösen. Um den Scheduler in ein FreeRTOS-Projekt zu integrieren, muss ein kleines Modul, bestehend aus einer C- und H-Datei in den Anwendungsbereich kopiert werden.

2.1 Datenstrukturen

Für die Implementierung eines EDF-Schedulers wurde als erstes zwei neue Datenstrukturen erstellt. Die erste Struktur mit der Bezeichnung 'edfTaskStruct', wie in Programmcode 3 zu sehen, beinhaltet ausschließlich alle Variablen und Parameter einer einzelnen Task die für den EDF-Scheduler notwendig sind. Unter anderem besteht die Struktur aus einem Pointer zum Taskcode, der Name der Task, sowie verschiedene EDF-Variablen.

```
typedef struct
{
    TaskHandle_t taskHandle;                // task handle
    const char* taskName;                   // task name
    BaseType_t taskCreated;                 // task created flag
    TickType_t wcet;                        // worst compution execution time
    TickType_t period;                      // period of task
    TickType_t lastStartTime;               // task relative deadline
    TickType_t absoluteDeadline;            // task absolute deadline
    TickType_t relativeDeadline;            // task relative deadline
    uint32_t callCounter;                   // task execution counter
}edfTaskStruct;
```

Programmcode 3: edfTaskStruct Struktur

In der zweiten Struktur werden alle EDF-Tasks, bzw. deren 'edfTaskStruct', in einem Array , wie in Programmcode 4 abgebildet, gespeichert. Außerdem beinhaltet diese Struktur noch verschiedene Flags, welche für die Initialisierung des EDF-Schedulers notwendig sind.

Wird eine EDF-Task erstellt, wird eine neue 'edfTaskStruct'-Struktur erstellt und der 'edfTasks'-Struktur hinzugefügt.

```
typedef struct
{
    edfTaskStruct tasksArray[MAX_EDF_TASKS];            // tasks array
    TaskHandle_t idleTask;                              // taskhandle idle
    task
    BaseType_t idleTaskCreated;                         // idle task created?
    unsigned int numberOfEDFTasks;                      // total number of
    tasks
}edfTasks;
```

Programmcode 4: edfTasks Struktur

2.2 Erstellung einer Earliest Deadline First (EDF)-Task

Für die Erstellung einer EDF-Task muss die 'createEDFTask'-Funktion , mit allen Parametern die in Tabelle 2 aufgelistet sind, aufgerufen werden. Bei dem Aufruf der Funktion wird eine EDF-Struktur mit allen erforderlichen Parametern für den EDF Scheduler hinzugefügt, anschließend wird die FreeRTOS-Task erstellt.

Parameter	Datentyp	Beschreibung
taskCode	TaskFunction_t	Funktionsname der Task
TaskName	const char*	Name der Task
stackDepth	configSTACK_DEPTH_TYPE	Größe des Taskstacks
pvParameters	void*	Task Parameter
wcet	TickType_t	maximale Ausführzeit
period	TickType_t	Periode der Task
deadline	TickType_t	Deadline der Task

Tabelle 2: Task Parameter von der Funktion 'createEDFTask'

Der Beispielaufruf der 'createEDFTask' ist in Programmcode 5 dargestellt. Diese beinhaltet eine Test-Funktion, so wie auch der Aufruf der createEDFTask.

```
/* Task to be created. */
void vTestFunction( void * pvParameters )
{
    for( ;; )
    {
        /* Task example code goes here. */
        int i = 1 + 2

        /* At the end of every EDF Task, the rescheduleEDF
        Function must be called */
        rescheduleEDF();
    }
}

BaseType_t createEDFTask(    // Task Creation
    vTestFunction,           // Pointer to task entry function
    "Test Function",         // A descriptive name for the task
    300,                     // The number of words to allocate
    NULL,                    // Task parameters
    1,                       // WCET in ms
    5,                       // Period of Task in ms
    4 );                     // Deadline of Task in ms
```

Programmcode 5: createEDFTask Beispiel

2.3 Löschen einer Earliest Deadline First (EDF)-Task

In manchen Applikationen wird das Löschen von EDF Tasks benötigt und wurde aus diesem Grund ebenfalls in dieser Arbeit implementiert. Dabei muss die Funktion 'deleteEDFTask' und der Stringname der zu löschenden Task aufgerufen werden. Bei dem Aufruf wird die zu löschende Task aus der EDF-Tasks Struktur entfernt, anschließend wird die Task in FreeRTOS gelöscht. Ein Beispielaufruf, angelehnt an das Beispiel aus Programmcode 5 ist in Programmcode 6 dargestellt.

```
1    deleteEDFTask("Test Function")
```

Programmcode 6: deleteEDFTask Beispiel

2.4 Implementierung

Für eine einfache und zugleich sehr performante Implementierung eines EDF-Schedulers in FreeRTOS zu gewährleisten, wurde in dieser Arbeit der FreeRTOS-Scheduler weiterverwendet. Der FreeRTOS-Scheduler arbeitet, wie in Abschnitt 1.3.1 bereits behandelt, prioritätsbasiert. Die eigentliche Scheduler-Logik befindet sich in der 'rescheduleEDF'-Funktion, welche bei jeder EDF-Task am Ende des Taskcodes, wie in Programmcode 5 zu sehen, aufgerufen werden muss. Mit dem Aufruf der Funktion 'rescheduleEDF' wird zwischen allen erstellten EDF-Tasks, die Priorität der nächsten Task nachdem EDF-Prinzip auf die höchste Priorität und die aktive Task auf eine niedere Priorität als die FreeRTOS Idle Task gesetzt. Aus diesem Grund wurde die Standard-Priorität der FreeRTOS Idle Task von dem Wert '0' auf den Wert '1' erhöht. Anschließend wird ein Context Switch ausgelöst und die nachdem EDF-Prinzip nächste Task wird von dem FreeRTOS-Scheduler aufgerufen und ausgeführt. Falls alle EDF-Tasks bereits in ihrer Periode ausgeführt wurden und alle Deadlines der Tasks weiter als ihre Periode in der Zukunft liegen, wird die FreeRTOS Idle Task aufgerufen.

Tasks	Priorität
Deaktivierte EDF Task(s)	0
FreeRTOS Idle Task	1
Auführende EDF Task	2
'rescheduleEDF' ausführende Task	3

Tabelle 3: EDF-Prioritäten der Tasks

Der EDF-Scheduler arbeitet mit vier unterschiedlichen Prioritäten, wie in Tabelle 3 dargestellt und inkrementiert bei der Implementierung die Priorität der Idle Task von der niedrigsten Priorität '0' auf die zweit niedrigste Priorität '1' in der FreeRTOS Umgebung. Alle EDF Tasks die nicht ausgeführt werden sollen, erhalten die Priorität 0, was dazu führt, dass die Idle Task immer ausgeführt wird und verhindert die nicht gewollte Ausführung einer EDF-Task. Um ein EDF-Scheduling in der Idle Task zu ermöglichen, wird

die 'rescheduleEDF'-Funktion mit Hilfe des Idle Task Hook aufgerufen. Die Task, welche aktuell ausgeführt wurde und am Ende die 'rescheduleEDF'-Funktion ausführt, erhält während der Funktionsausführung die höchste EDF-Priorität mit dem Wert '3', um einen vorzeitigen Context Switch zur nächsten, nach dem EDF-Prinzip ausgewählte Task, zu unterbinden. Nachdem alle EDF-Parameter von allen EDF-Tasks neu gesetzt wurden, gibt die aktuelle Task die höchste Priorität mit dem Wert '3' ab und erzwingt einen Context Switch zu der nächsten zu ausführenden EDF-Task mit der Priorität '2'. Für einen reibungslosen Start des EDF-Schedulers, werden die Prioritäten aller EDF-Tasks nachdem Erstellen automatisch auf die Priorität mit dem Wert '0' gesetzt, damit die Idle Task die allerste Ausführung übernimmt und somit zu Beginn die 'rescheduleEDF'-Funktion durch den Idle Task Hook ausgeführt wird. In Abbildung 4 werden die verschiedenen Prioritäten je nach Status der Task innerhalb einer Periode angezeigt.

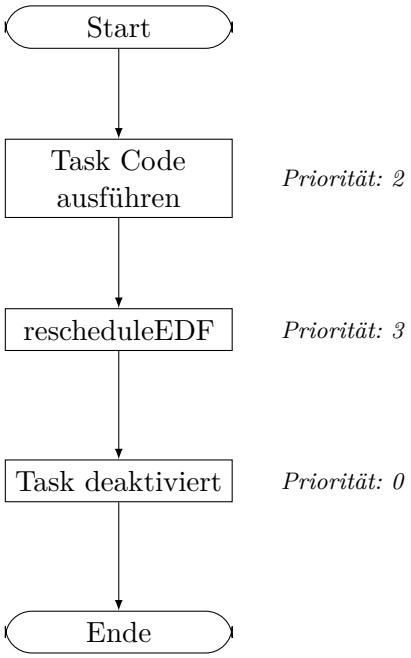
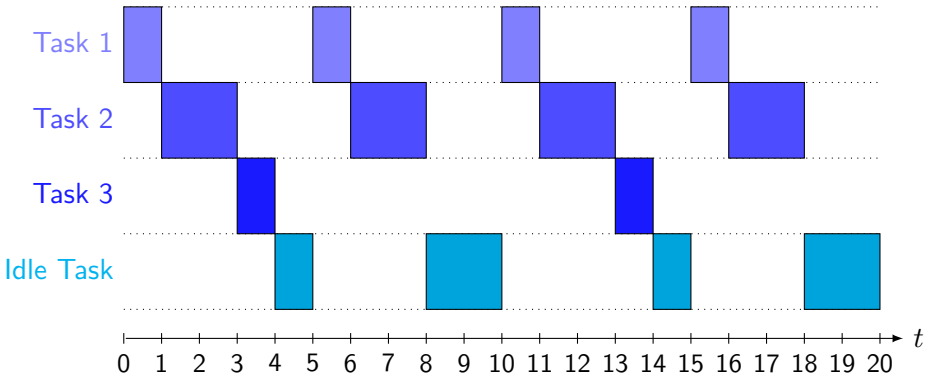


Abbildung 4: EDF-Task Ausführungsablauf (eigene Darstellung)

Bei der Berechnung der Prioritäten in der 'rescheduleEDF'-Funktion wird dabei die Deadline abzüglich der WCET berücksichtigt, um nicht einer Task CPU-Zeit zu geben, welche sowieso Ihre Deadline überschreiten würde. Um einen Einblick in das Scheduling-Verfahren, mit der in dieser Arbeit implementierten EDF-Schedulers, zu bekommen, ist in Abbildung 5 der Ablauf von drei Beispiel-Tasks dargestellt.



Taskname	Periode	Deadline	WCET
Task 1	5	3	1
Task 2	5	5	2
Task 3	10	10	1

Abbildung 5: Demonstrationsablauf des EDF-Schedulers (eigene Darstellung)

Bei der Darstellung in Abbildung 5 wurde die Anfangsphase, so wie der erste Aufruf der Idle Task vernachlässigt. Des Weiteren wurde als Vereinfachung der Darstellung der Starttick der ersten Task als Null angenommen. Als erstes bekommt Task 1 CPU-Zeit zugeteilt, da diese die kürzeste Deadline besitzt. Anschließend wird Task 2 ausgeführt, welche die nächst kürzeste Deadline besitzt und als letztes wird Task 3 ausgeführt. Nachdem nun alle drei Tasks innerhalb ihrer Periode unter Einhaltung der Deadlines ausgeführt wurden, wird die Idle Task solange ausgeführt bis die nächste Periode einer Task beginnt.

2.5 Genauigkeit

Der in dieser Arbeit erstelle EDF-Scheduler verwendet die FreeRTOS-API und somit auch den FreeRTOS-SysTick Timer, welcher standardmäßig jede Millisekunde einmal aufgerufen wird. Das bedeutete, der EDF-Scheduler kann maximal EDF-Tasks mit einer minimalen Ausführzeit von einer Millisekunde einplanen. Sollte eine Task eine kürzere Ausführzeit besitzen, wird dies von dem EDF-Scheduler nicht wahrgenommen. Die Periode des FreeRTOS-SysTick und somit die Genauigkeit des EDF-Schedulers kann unter

der Datei 'FreeRTOSConfig.h', wie in wie in Programmcode 7 dargestellt, konfiguriert werden. Eine kleinere Periode des FreeRTOS-SysTick führt zu einem häufigeren Aufruf und somit zu einer größeren CPU-Belegung, wobei auftauchende Interrupts, wie die CPU-Ausführzeit des SysTick, in dieser Arbeit vernachlässigt werden. Dieser Standardwert bietet für den EDF-Scheduler einen guten Kompromiss zwischen Applikationslaufzeit und Genauigkeit.

```
1    #define configTICK_RATE_HZ              ( 1000 ) // 1ms SysTick
```

Programmcode 7: FreeRTOS SysTick Deklaration

2.6 Debugmodus

Um das Entwickeln einer EDF-Applikation zu vereinfachen oder auch Bugs zu finden, wurde ein Debug-Flag integriert, welches folgende Features, sofern aktiviert, beinhaltet:

- **Universal Asynchronous Receiver Transmitter (UART)-Schnittstelle**: Über die UART Schnittstelle wird die erfolgreiche Konfiguration der Hardware, der Start des FreeRTOS-Schedulers, sowie auch der Start jeder EDF-Funktion mitgeteilt. Des Weiteren werden nicht eingehaltene Deadlines und die Anzahl der Deadlinefehler, sofern vorhanden, mitgeteilt.

- **Erweiterte 'edfTaskStruct'-Struktur:** Die 'edfTaskStruct'-Struktur wird um folgende fünf Parameter erweitert:

  - **lastRunningTime**: Enthält die letzte Ausführzeit der Task in SysTicks

  - **maxRunningTime**: Enthält die bisherige maximale Ausführzeit (WCET der Task in SysTicks

  - **startTime**: Enthält die letzte Startzeit der Task in SysTicks

  - **stopTime**: Enthält die letzte Stoppzeit der Task in SysTicks

  - **deadlineErrorCounter**: Enthält die Anzahl der nicht eingehaltenen Deadlines

- **Erweiterte 'edfTasks'-Struktur:** Die 'edfTasks'-Struktur wird um folgende drei Parameter erweitert:

  - **currentTick**: Enthält den aktuellen SysTick des EDF-Schedulers

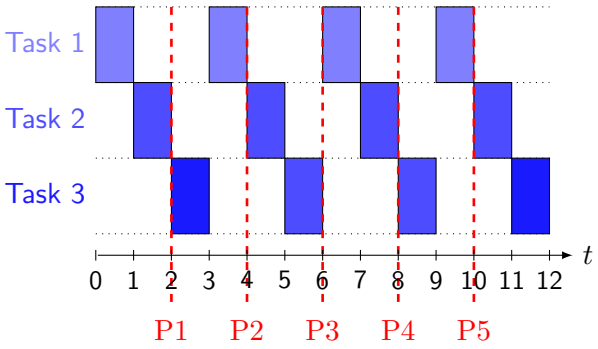
  - **wcet**: Enthält die höchste WCET

– **cet**: Enthält die aktuelle Ausführzeit der ausführenden Task

Dieses Flag sollte allerdings nicht im Produktivbetrieb benutzt werden, da dies zu unvorhersehbaren Verzögerungen der Tasks und im Scheduling führen kann.

2.7 Fehlerbehandlung

Eine Überschreitung der Deadline kann beispielsweise durch eine Überbelegung der CPU-Zeit oder durch fehlerhafte Parametrierung einer EDF-Task ausgelöst werden. Falls eine Task ihre Deadline überschritten hat oder die Zeit für die Ausführung nicht mehr reicht, werden die EDF-Parameter der Task um eine Periode erhöht. Dadurch wird keine Task bei der Fehlerbehandlung bevorzugt oder benachteiligt, wobei natürlich die Task mit der kürzesten Deadline und der kleinsten Periode eine höhere Wahrscheinlichkeit besitzt, ausgeführt zu werden.



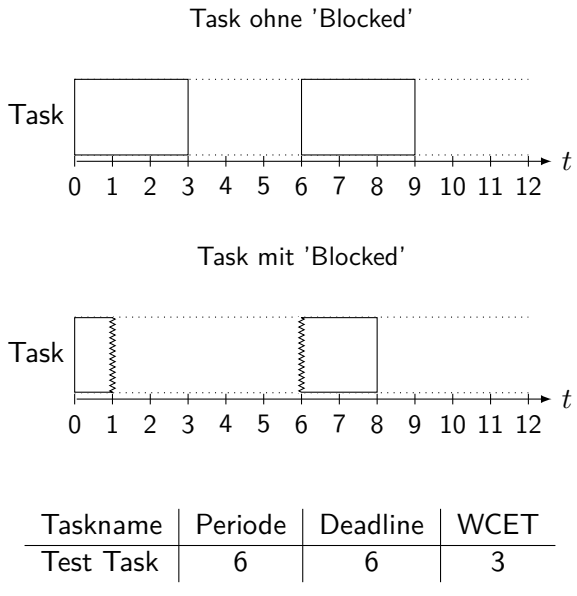
Taskname	Periode	Deadline	WCET
Task 1	2	2	1
Task 2	2	2	1
Task 3	2	2	1

Abbildung 6: Beispiel Deadline Errors

Wie in Abbildung 6 dargestellt, können aufgrund der Task Eigenschaften immer nur zwei Tasks in ihrer Periode, welche mit roten Linien eingezeichnet ist, ihre Deadline erfüllen. Durch die implementierte Deadline-Fehlerbehandlung, wird bei gleicher Deadline zweier Tasks, immer die Task mit weniger Aufrufen ausgeführt. Dieses Vorgehen ermöglicht die gleichmäßige Ausführung aller Tasks obwohl die Deadline nicht immer eingehalten werden kann. Eine andere Möglichkeit der Fehlerbehandlung wäre das unterbinden einer der drei Tasks, um den anderen zwei Tasks dauerhaft das Einhalten ihrer Deadlines zu ermöglichen.

2.8 Limitationen

Eine FreeRTOS-Task kann, wie in Abschnitt 1.3.3 beschrieben, in den 'Blocked' Status gehen und übergibt, trotz höhere Priorität, CPU-Zeit an eine niedrigere Zeit. Dieses Vorgehen macht Sinn, wenn z.B. eine höher priorisierte Task auf Daten warten muss oder in der Applikation die Funktion 'vTaskDelay' benutzt wurde. Aus der Überlegung ob dies der EDF-Scheduler unterstützen sollte, wurde die Laufzeit der 'rescheduleEDF'-Funktion mit und ohne der 'Blocked'-Implementierung getestet. Die Laufzeit eines Taskhandlings vergrößerte sich zwar nur gering, jedoch muss die Summe aller Taskhandlings einer Anwendung betrachtet werden. Aus diesem Grund unterstützt der in dieser Arbeit entworfene EDF-Scheduler keine 'Blocked'-Status. Aus diesem Grund und der verringerten Laufzeit der 'rescheduleEDF' darf eine EDF-Task nicht in den 'Blocked' Status gesetzt werden. Sollte dies dennoch in einer Applikation nötig sein, muss beachtet werden, dass die betreffende Task erst innerhalb ihrer nächsten Periode, wie in Abbildung 7 dargestellt, wieder aufgerufen wird.



Taskname	Periode	Deadline	WCET
Test Task	6	6	3

Abbildung 7: 'Blocked' Status in EDF-Tasks

Abbildung 7 zeigt, sind zwei Tasks mit den identischen EDF-Eigenschaften dargestellt, allerdings enthält eine Task einen Programmcode welcher zu einem 'Blocked' Status führt. Aus diesem Grund wird diese Task erst in der nächsten Periode fortgesetzt, wobei die Task ohne einen 'Blocked' Status in der gleichen Zeit zweimal ausgeführt wird.

2.9 Kompatibilität

Durch die Verwendung des originalen Schedulers können klassische FreeRTOS Tasks parallel erstellt und abgearbeitet werden. Hierbei ist aber zu beachten, dass die Priorität der klassischen FreeRTOS Task eine höhere Priorität als den Wert '3' besitzt, da es ansonsten zu Konflikten innerhalb des EDF-Schedulers kommen kann. Auch sollte bei der Implementierung von klassischen FreeRTOS-Task darauf geachtet werden, dass die EDF-Tasks genügend Zeit für Ihre Abarbeitung zur Verfügung gestellt bekommen. Außerdem sorgt die Verwendung der FreeRTOS-API für die gleiche Kompatibilität wie FreeRTOS zu anderen Architekturen.

3 Demo-Applikationen

Ein Teil der Aufgabenstellung beinhaltet das Erstellen von zwei Demo-Applikationen, welche die Funktion des EDF-Schedulers unter Beweis stellen. Die CPU-Auslastung bezeichnet dabei, die theoretische Belegung der CPU-Zeit von allen EDF-Tasks.

3.1 Blinky-Demo

Für eine optische Vorführung des EDF-Schedulers wurde eine 'Blinky-Demo' erstellt, welche drei EDF-Tasks beinhaltet. Jede der drei Tasks toggelt eine unterschiedliche LED, besitzt aber die gleichen EDF Parameter, welche in Tabelle 4 dargestellt sind.

Taskname	Periode	Deadline	WCET
Task 1	50	100	100
Task 2	50	100	100
Task 3	50	100	100

Tabelle 4: Blinky-Demo Task Parameter

Die drei Tasks wurden exakt so gewählt, dass sie die verschiedenen möglichen Status des EDF-Schedulers simulieren und optisch dargestellen werden. Der Benutzer kann mit Hilfe des integrierten Button auf dem STM32F769I-Disc0 Board die drei Tasks, wie in Abbildung 8 zu sehen, erstellen und löschen. Eine CPU-Auslastung über 100% ist nicht möglich und führt somit zur Fehlerbehandlung aus Abschnitt 2.7.

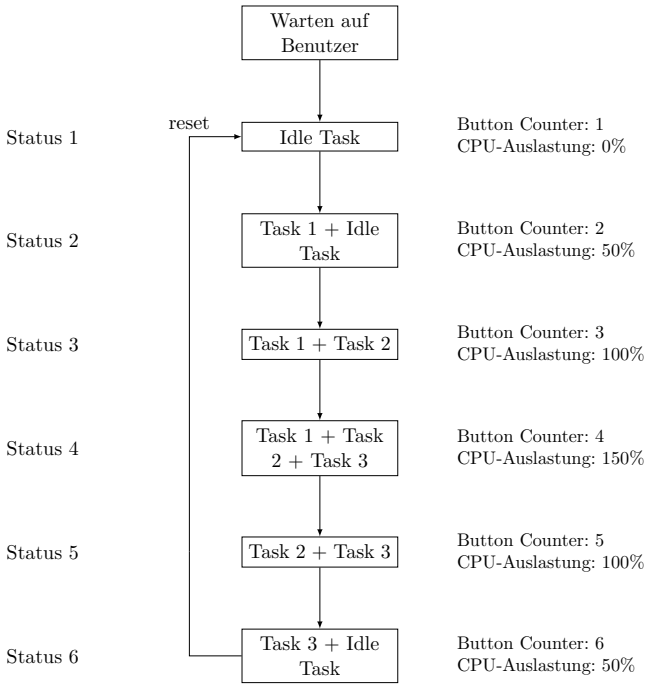


Abbildung 8: Blinky-Demo Ablaufdiagramm (eigene Darstellung)

Die Eigenschaften der einzelnen Status aus Abbildung 8 sind wie folgt:

- **Warten auf Benutzer**: Anfangs ist keine EDF-Task gestartet, die LEDs leuchten dauerhaft und die Idle Task wird dauerhaft ausgeführt.

- **Status 1**: Sobald der Benutzer zum ersten Mal den Button drückt, wird 'Task 1' erstellt und ausgeführt. Nun blinkt die erste LED regelmäßig, die anderen beiden LEDs leuchten weiterhin dauerhaft. Durch diese Konstellation teilen sich 'Task 1' und die Idle Task die CPU-Zeit und es ergibt sich eine CPU-Auslastung von 50%.

- **Status 2**: Wird nun der Button ein zweites Mal gedrückt, wird 'Task 2' erstellt und es blinkt nun auch die zweite LED regelmäßig. Aufgrund der Taskparameter von 'Task 1' und 'Task 2' teilen sich die beiden Tasks nun die komplette CPU-Zeit und lasten diese zu 100% aus, dadurch wird die Idle Task nicht mehr ausgeführt.

- **Status 3**: Drückt der Benutzer nun ein drittes Mal den Button, wird 'Task 3' erstellt, die dritte LED fängt nun auch an zu blinken. Aufgrund der gewählten Taskparameter beträgt die CPU-Auslastung nun 150%, hier kann der EDF-Scheduler die einzelnen Deadlines nicht einhalten. Dies wird anhand der LEDs sichtbar, da diese nun nur noch unregelmäßig blinken. Des Weiteren, sofern der Debug Mode aktiviert ist, können die nicht eingehaltenen Deadlines der Tasks über die UART-Schnittstelle,

sowie welche Task wie oft ihre Deadline verfehlt hat, ausgelesen werden.

- **Status 4**: Wird nun der Button ein weiteres Mal gedrückt, wird 'Task 1' gelöscht und die erste LED wird ausgeschaltet. Nun teilen sich 'Task 2' und 'Task 3' die CPU-Zeit und die CPU-Auslastung beträgt wieder 100%, aus diesem Grund wird die Idle Task auch nicht aufgerufen.

- **Status 5**: Ein weiteres Drücken des Buttons löscht 'Task 2', nun teilen sich 'Task 3' und die Idle Task jeweils die hälfte der CPU-Zeit.

- **Status 6**: Drückt der Benutzer nun ein weiteres Mal den Button wird 'Task 3' gelöscht und alle drei LEDs leuchten dauerhaft. Die CPU-Auslastung liegt nun bei 0%, dadurch wird die Idle Task dauerhaft ausgeführt. Das System befindet sich nun wieder am Anfang der Ausführung und kann erneut die einzelnen Status ausführen.

3.2 Fast Fourier Transformation (FFT)-Demo

Die FFT-Demo erstellt und sendet mit Hilfe der erarbeiteten GUI auf dem Host-Computer alle 10ms ein Signal, bestehend aus einem oder mehreren Sinus- oder Kosinus-Signalen, an den Mikrocontroller über das UDP-Protokoll. Das Interval von 10ms wurde so gewählt, dass eine ausreichend schnelle Aktualisierung und eine mittelmäßige Auslastung des Host-Computers gewährleistet ist. Dieses Interval kann in der Konfiguration der FFT-Demo angepasst werden, jedoch sollte das Interval nicht kleiner als 6ms gewählt werden, da die Summe der WCET aller Tasks auf dem Mikrocontroller 6ms beträgt. Der Mikrocontroller führt, basierend auf dem empfangenen Signal, eine FFT-Berechnung aus und sendet die Ergebnisse an den Host-Computer zurück. Das Ablaufdiagramm mit allen Komponenten für die FFT-Demo ist in Abbildung 9 zu sehen.

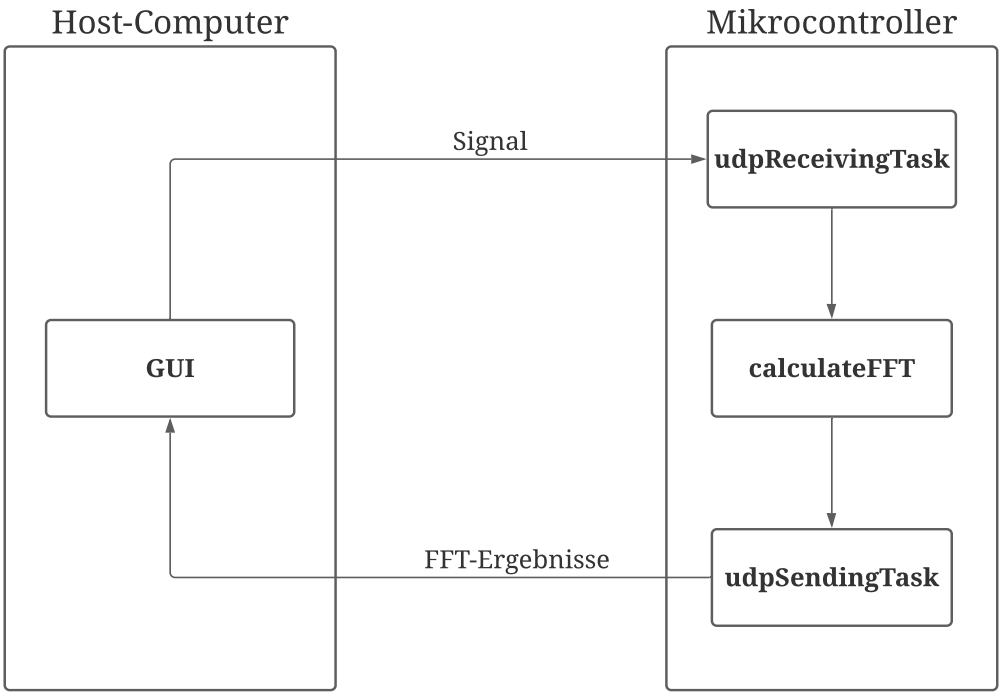


Abbildung 9: FFT-Demo Ablaufdiagramm (eigene Darstellung)

Für das Erstellen und Senden eines Signals, des Empfangs der Ergebnisse und das Darstellen der Signale wurde eine GUI, basierend auf der Programmiersprache Python, erstellt. Des Weiteren wurden für die Kommunikation mit dem Host-Computer sowie die Berechnung der FFT drei EDF-Tasks erstellt. Das in dieser Arbeit generierte Signal wurde mit 2048 Abtastpunkten erzeugt und besitzt somit die gleiche Größe wie die verwendete FFT. Die Standard UDP-Paketgröße beträgt 1454 Bytes. Um 2048 Abtastpunkte

von float-Werten (4 Byte) zu übermitteln, muss das Signal in mindestens 6 Pakete aufgeteilt werden. Jedoch besitzt das UDP-Protokoll keine Sortierfunktion, was bedeutet, falls ein Paket langsamer als das nächstfolgende Paket übertragen wird, würde dies zu einer Signalverfälschung führen. Aus diesem Grund wird bei jedem Paket auch die Paketnummer in einer Variable mit übertragen. Somit werden zur Übertragung von 2048 Abtastpunkten acht Pakte mit je 256 Samples übertragen.

3.2.1 Graphical User Interface (GUI)

Die GUI wurde für eine einfache Bedienung, sowie mit einer übersichtlichen Anzeige gestaltet und ist in Abbildung 10 abgebildet. Dabei wurde auf eine einfache, flexible und übersichtliche Rücksicht genommen.

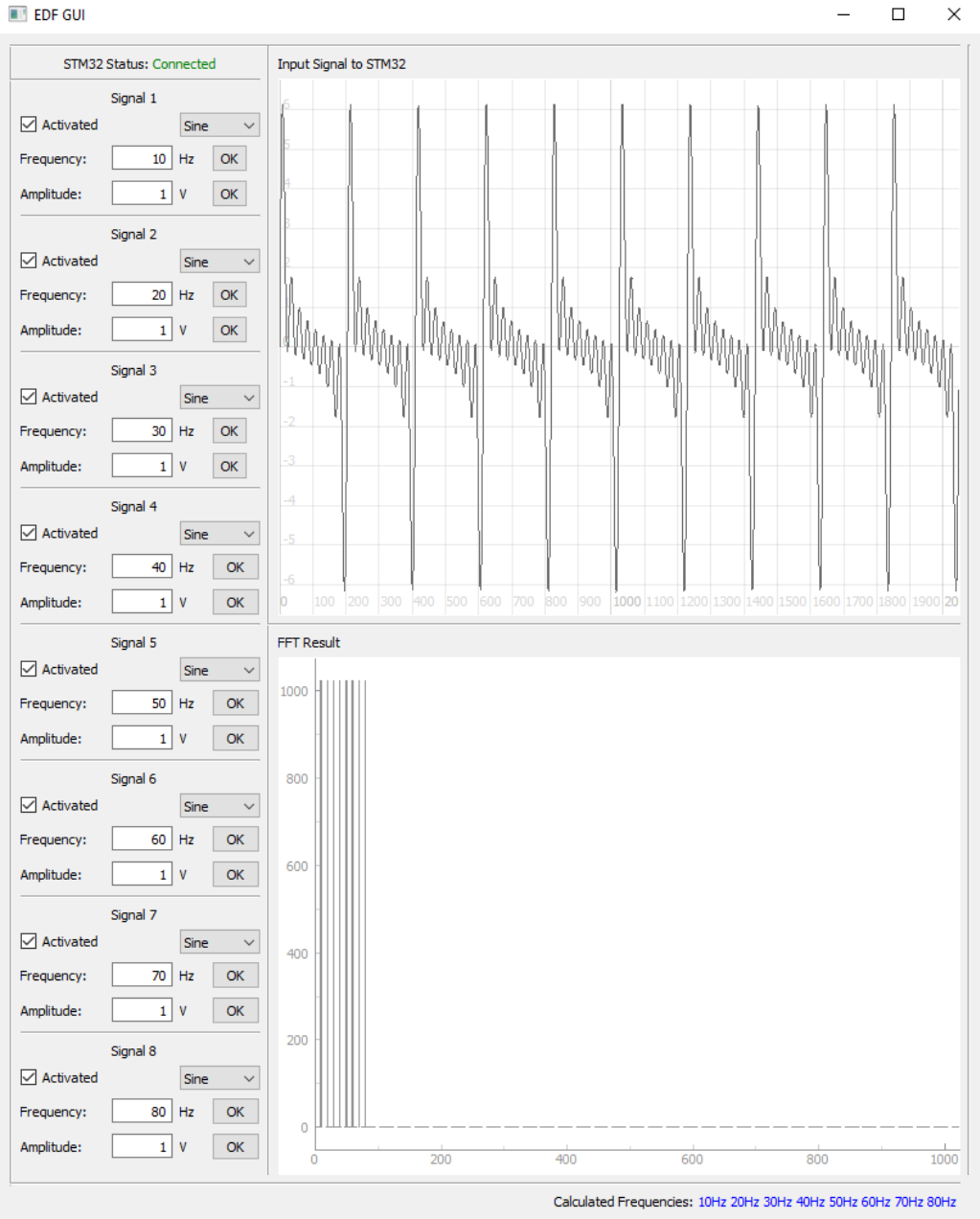


Abbildung 10: Grafische Benutzeroberfläche

In der linken oberen Ecke wird der Verbindungsstatus zum Mikrocontroller angezeigt, welcher mittels Pings von dem Host-Computer ermittelt wird. Unter dem Verbindungsstatus kann der Benutzer bis zu acht Signale aktiveren oder deaktivieren sowie auch deren Eigenschaften verändern. Das generierte Signal wird zentral in dem oberen Plotfenster dargestellt. Im unteren Plotfenster werden die Ergebnisse der FFT-Berechnung angezeigt, zusätzlich werden die erkannten Frequenzen in Textform unten eingeblendet.

3.2.2 Tasks des Mikrocontrollers

Die Perioden der drei EDF-Tasks, siehe Tabelle 5, wurden an das Interval der GUI auf 10ms angepasst. Jede der drei Task hat einen WCET von 2ms, welche mit Hilfe des Debugmodus ermittelt wurden, somit ergibt sich eine CPU-Auslastung von 60%, wie auch in Abbildung 11 dargestellt.

Taskname	Periode	Deadline	WCET	Aufgabe
Task 1	10	10	2	Empfangen des Signals über UDP
Task 2	10	10	2	Berechnung der FFT
Task 3	10	10	2	Senden der FFT Ergebnisse über UDP

Tabelle 5: FFT-Demo Task Parameter

Die WCET ändern sich mit der Größe des empfangenen Signals und der Größe der FFT, jedoch wurden die Werte so gewählt, dass ein guter Kompromiss zwischen Arbeitsspeicherbelegung und Signalverarbeitung gewährleistet ist. Für die Implementierung der FFT-Demo ist es notwendig, dass die Tasks untereinander Daten austauschen können. Aus diesem Grund wurden FreeRTOS Queues verwendet, damit die Tasks sich untereinander nicht blockieren können.

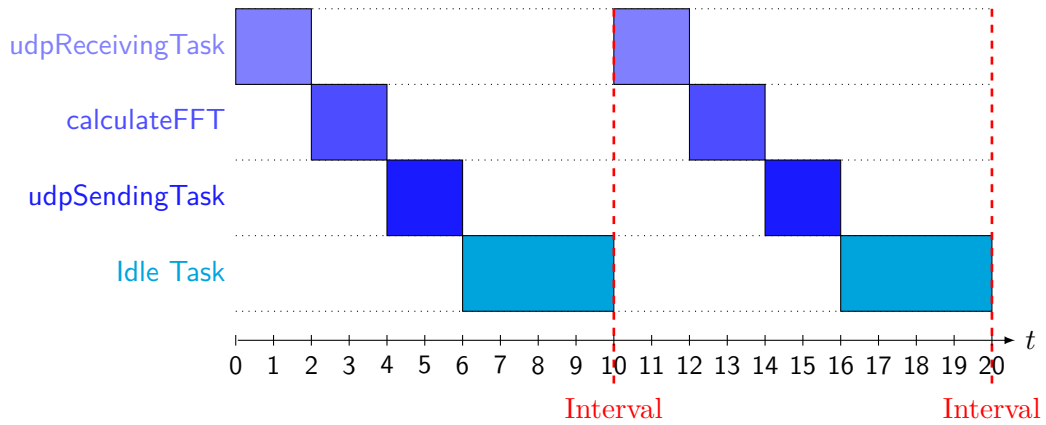


Abbildung 11: FFT-Demo Task Scheduling (eigene Darstellung)

4 Reflektion

Bei der Entwicklung des EDF-Schedulers traten folgende Probleme auf:

- **Natives FreeRTOS**: Als Integrated Development Environment (IDE) wurde in dieser Arbeit, aufgrund der einfachen Bereitstellung der Board-Treiber, die Software 'STM32CubeIDE' von ST verwendet. Bei der Konfiguration der Hardware und Software kann eine FreeRTOS Installation mit ausgewählt werden. Leider stellte sich heraus, dass dabei ein Wrapping Layer automatisch mit konfiguriert und implementiert wird, um einen einfachen Wechsel zwischen den unterschiedlichen RTOS Betriebssystemen zu ermöglichen. Dieser Wrapping Layer verhindert aber eine einfache Portierung auf andere Mikrocontroller, speziell Mikrocontroller die nicht von dem Hersteller ST stammen. Resultierend aus diesen Erfahrungen wurde dann ein natives FreeRTOS ohne Wrapping Layer in dieser Arbeit auf dem STM32F769I-Disc0 implementiert.

- **FreeRTOS+TCP**: Bei der Integration des FreeRTOS+TCP Stack konnte lange nicht die Ethernet-Schnittstelle aktiviert werden. Schließlich stellte sich heraus, dass bei der Konfiguration der Netzwerkschnittstelle in 'STM32CubeIDE' automatisch eine Initialisierungsfunktion integriert und diese in der Main-Funktion aufgerufen wird. Das Problem bestand darin, dass der FreeRTOS+TCP Stack eine eigene Initialisierungsfunktion verlangt und die Schnittstelle daher blockiert wurde. Nachdem die 'Standard'-Initialisierungsfunktion entfernt wurde, konnte die Netzwerkschnittstelle aktiviert werden.

- **Ethernetschnittstelle**: Für die Kommunikation zwischen Host-Computer und dem Mikrocontroller wurde das UDP-Protokoll und somit die Netzwerkschnittstelle des Mikrocontrollers ausgewählt. Damit nicht zwischen entwickeln und der recherche im Internet die RJ45-Kabel am Host-Computer gewechselt werden musste, wurde Anfangs der Mikrocontroller direkt an einem Switch im heimischen Netzwerk angesteckt. Nachdem die Integration des FreeRTOS+TCP Stacks abgeschlossen war, wurden mehrere Ping Test durchgeführt, doch der Mikrocontroller blieb nicht erreichbar. Nach der Überprüfung der Konfiguration wurde der Mikrocontroller direkt mit dem Host-Computer verbunden und antwortete sofort auf die Ping-Anfragen. Wahrscheinlich wurden in dem vorhandenen Netzwerk für den Mikrocontroller zu viele Broadcastanfragen gesendet.

- **FFT-Größe**: Je größer die FFT konfiguriert wird, je länger braucht deren Berechnung. Gleichzeitig sendet der Host-Computer weiterhin neue Pakte zum Mikrocontroller, was dazu führt, dass bei der Erhöhung der FFT-Größe auch die Stackgröße der udpReceivingTask vergrößert werden musste. Ansonsten kommt es zu einem Buffer Overflow. Das verwendete Board hat eine Arbeitsspeichergröße von 512kb und somit ausreichend Platz für größere Variablen. Allerdings wurde es als nicht

sinnvoll erachtet, den Hauptteil des Arbeitsspeichers mit Buffer Variablen zu belegen.

5 Fazit und Ausblick

Als Ergebnis dieser Thesis wurde ein EDF-Scheduler in FreeRTOS implementiert. Dabei wurde darauf geachtet, dass keine Funktionen oder Variablen der FreeRTOS Installation modifiziert wurden, um eine einfache Portierbarkeit auf andere Mikrocontroller zu ermöglichen. Die Entscheidung den FreeRTOS SysTick zu verwenden, trägt weiter dazu bei. Ebenfalls führt die Implementierung eines Debugmodus und deren Kommunikation mit dem UART zu einer einfachen Fehlerbehebung während eines Entwicklungsprozesses. Die Möglichkeit den Mikrocontroller in den Energiesparmodus zu setzen, solange die FreeRTOS Idle Task ausgeführt wird, wurde bewusst nicht implementiert. Diese Entscheidung beruht darauf, dass eine Implementierung des Energiesparmodus bei jeder CPU anders ist und somit eine Einschränkung der Portierbarkeit darstellen würde.
Im weiteren Verlauf wurde eine 'Blinky-Demo' erstellt, welche die verschiedenen Status des EDF-Schedulers anhand von LEDs darstellen kann. Die blinkenden LEDs zeigen dabei direkt den Status, in welcher sich die Demo-Applikation befindet, an. Sofern der Debugmodus aktiviert ist, können die einzelnen Status und wenn vorhanden, die nicht eingehalten Deadlines mit der Hilfe des UART mit verfolgt werden. Außerdem konnte mit der 'FFT-Demo' auch ein wesentlich komplexerer Anwendungsfall für die Überprüfung des EDF-Schedulers implementiert werden. Dabei wird die sichere Kommunikation der Tasks untereinander mit FreeRTOS Queues gewährleistet und somit können sich die Tasks nicht gegenseitig blockieren. Die einfache Konfiguration der FFT-Größe sowie die daraus resultierende UDP-Paketgröße trägt zu einer Vielzahl von Anwendungen bei, da somit verschiedenste Samplegrößen verarbeitet werden können.
Die erstellte GUI erleichtert dabei die Erzeugung verschiedenster Signale, sowie auch das Darstellen des erzeugten Signals und der FFT-Ergebnisse. Durch die freie Auswahl der Amplitudengröße sowie der Frequenz ist bei der Erzeugung eines Signals für die FFT-Berechnung keine Grenzen gesetzt. Bei der Betrachtung der Plots können diese interaktiv vergrößert und verkleinert werden. Das Anzeigen der erkannten Frequenzen hilft dabei schnell einen Überblick über das Spektrum zu erhalten. Somit konnte die korrekte Funktion des EDF-Schedulers mit zwei Demo-Applikationen unter Beweis gestellt werden.
Abschließend ist zu sagen, dass mit Hilfe dieser Arbeit in FreeRTOS unter den gesetzen Bedingungen nun ein Scheduler verwendet werden kann, der eine CPU-Auslastung bis zu 100% unterstützt. Die Implementierung eines EDF-Schedulers in FreeRTOS ermöglicht eine größere Auswahl des Scheduler-Betriebes, wodurch für jede Applikation die Möglichkeit steigt, den passenden Scheduler verwenden zu können.

In Zukunft könnte eine neue Flag implementiert werden, welche die 'rescheduleEDF'-Funktion erweitert, um den FreeRTOS 'Blocked'-Status, unter dem Wissen einer längeren Laufzeit, zu berücksichtigen. Außerdem könnten noch weitere Scheduler-Arten implementiert werden, welche den Einsatzzweck der Arbeit noch erweitern würde und somit mehr Anreize für die Weiterentwicklung der Arbeit bieten. Das zur Verfügung gestellte STM32F769I-Disc0 Board wird mit einem Display ausgeliefert, mit dem eine Anzeige aller Tasks und deren Status, unter Berücksichtigung der zusätzlichen CPU-Zeit, möglich ist. Des Weiteren wäre eine Implementierung eines parallelen Betriebes von EDF-Tasks und FreeRTOS-Tasks möglich und in manchen Anwendungen von Vorteil.

Literatur

Baumgartl, R. (2008). Prozessorzuteilung [[Online; Stand 27. Januar 2021]]. http://www.informatik.htw-dresden.de/~robge/ezs/vl/ezs1-kompendium.pdf

Components, R. (2020). STM32F769I-Disc0 [[Online; Stand 27. Januar 2021]]. https://befr.rs-online.com/web/p/microcontroller-development-tools/1231055/

Creating a New FreeRTOS Port [[Online; Stand 27. Januar 2021]]. (2020). https://www.freertos.org/FreeRTOS-porting-guide.html

Embedded Systems Engineering [[Online; Stand 27. Januar 2021]]. (n. d.). https://www.mikrocontroller.net/attachment/89657/1a_timer_irq_ESE_Skript.pdf

FreeRTOS. (2020a). FreeRTOS [[Online; Stand 9. November 2020]]. https://www.freertos.org/

FreeRTOS. (2020b). FreeRTOS Features [[Online; Stand 9. November 2020]]. https://www.freertos.org/

FreeRTOS. (2020c). FreeRTOS Footprint [[Online; Stand 11. Januar 2021]]. https://www.freertos.org/RTOS.html

FreeRTOS. (2020d). FreeRTOS Libraries [[Online; Stand 11. Januar 2021]]. https://www.freertos.org/FreeRTOS-Plus/index.html

FreeRTOS. (2020e). FreeRTOS Overhead [[Online; Stand 11. Januar 2021]]. https://www.freertos.org/FAQMem.html#ContextSwitchTime

FreeRTOS. (2020f). FreeRTOS Task States [[Online; Stand 11. Januar 2021]]. https://www.freertos.org/RTOS-task-states.html

FreeRTOS. (2020g). License Details [[Online; Stand 27. Januar 2021]]. https://www.freertos.org/a00114.html

FreeRTOS. (2020h). SysTick [[Online; Stand 27. Januar 2021]]. https://www.digikey.com/en/maker/projects/getting-started-with-stm32-introduction-to-freertos/ad275395687e4d85935351e16ec575b1

FreeRTOS. (2020i). Tasks [[Online; Stand 27. Januar 2021]]. https://www.freertos.org/RTOS-idle-task.html

Hoegel, B. (2020). Determinismus (Algorithmus) [[Online; Stand 27. Januar 2021]]. https://www.biancahoegel.de/computer/inform/berechnen/algorithmus_determin.html

Mandl, P. (2014). *Grundkurs Betriebssysteme* [Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation, Virtualisierung]. Springer Vieweg.

Microcontrollerslab. (2021). FreeRTOS Scheduler: Learn to Configure Scheduling Algorithm [[Online; Stand 11. Januar 2021]]. https://microcontrollerslab.com/freertos-scheduler-learn-to-configure-scheduling-algorithm/#:~:text=FreeRTOS%20Scheduling%20Policy,-FreeRTOS%20kernel%20supports&text=In%20this%20algorithm%2C%20all%20equal,before%20a%20low%20priority%20task.

Mikrocontroller.net. (2019). Multitasking [[Online; Stand 27. Januar 2021]]. https://www.mikrocontroller.net/articles/Multitasking

Siemers, P. D. C. (2017). Echtzeit: Grundlagen von Echtzeitsystemen [[Online; Stand 9. November 2020]]. https://www.embedded-software-engineering.de/echtzeit-grundlagen-von-echtzeitsystemen-a-669520/

UDP - User Datagram Protocol [[Online; Stand 27. Januar 2021]]. (2021). https://www.elektronik-kompendium.de/sites/net/0812281.htm

Werner, M. (2012). *Digitale Signalverarbeitung mit MATLAB®* [Grundkurs mit 16 ausführlichen Versuchen]. Springer Vieweg.

Wikipedia. (2020a). Prozess-Scheduler — Wikipedia, Die freie Enzyklopädie [[Online; Stand 11. Januar 2021]]. https://de.wikipedia.org/w/index.php?title=Prozess-Scheduler&oldid=205225114

Wikipedia. (2020b). Schnelle Fourier-Transformation — Wikipedia, Die freie Enzyklopädie [[Online; Stand 16. Januar 2021]]. https://de.wikipedia.org/w/index.php?title=Schnelle_Fourier-Transformation&oldid=201776759

Wikipedia. (2021). Interrupt — Wikipedia, Die freie Enzyklopädie [[Online; Stand 11. Januar 2021]]. https://de.wikipedia.org/w/index.php?title=Interrupt&oldid=207161465

6 Anlagen

Der EDF-Scheduler beinhaltet bis dato kein eigenes Repository sondern wurde zu den zwei Demo Applikationen hinzugefügt. Alle erstellten Projekte sind online unter folgenden Links zu erreichen:

- **FFT Demo**: https://gitlab.fa-wi.de/punicawaikiki/freertos-ethernet-edf
- **Blinky Demo**: https://gitlab.fa-wi.de/punicawaikiki/edf-blinky-test
- **GUI**: https://gitlab.fa-wi.de/punicawaikiki/edf-python-interface

6.1 Verzeichnis des Datenträgers

- DatenCD_Masterthesis
  - 01_PDF_Masterthesis
  - 02_FFT_Demo
  - 03_Blinky_Demo
  - 04_GUI

Anmerkungen:
bla binaries auf controller kopieren bla exe starten