Ravensburg-Weingarten University

Master Thesis

Implementierung eines Earliest First Deadline Scheduler in FreeRTOS auf einem STM32

Fabian Wicker 32235

22. 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 22. 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. Die korrekte Funktion des EDF-Schedulers soll anhand zwei Testanwendungen garantiert werden. Folgende Aufgaben gehören zu dieser Masterarbeit:

1. Implementierung des EDF-Schedulers in FreeRTOS
2. Erstellung einer visuellen 'Blinky' Demonstration
3. Integration einer Fast Fourier Transformation (FFT)-Berechnung
4. Signalgenerator auf einem Host-Computer

HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Abstract

Lückenfüller...

HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Inhaltsverzeichnis

HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Abkürzungen

z.B.	zum Beispiel
WLAN	Wireless Local Area Network
FPGA	Field Programmable Gate Array
DSP	Digital Signal Processor
EDF	Earliest Deadline First
FreeRTOS	Free Real-Time Operating System
FFT	Fast Fourier Transformation
RMS	Rate Monotonic Scheduling
DMS	Deadline Monotonic Scheduling
CPU	Central Processing Unit
WCET	Worst Execution Time
MQTT	Message Queuing Telemetry Transport
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
API	Application Programming Interface
CLI	Command Line Interface
RAM	Random-Access Memory

ROM	Read-Only Memory
IRQ	Interrupt Request
ISR	Interrupt Service Routine
i.A.a	in Anlehnung an
FIFO	First Input First Output
DFT	Diskrete Fourier Transformation
CMSIS	Cortex Microcontroller Interface Standard
USART	Universal Asynchronous Receiver Transmitter
LED	Light Emitting Diode

Formelverzeichnis

1 Grundlagen

1.1 Echtzeitsysteme

Unter Echtzeit wird die Anforderung bezeichnet, dass innerhalb einer kürzesten definierten Zeitspanne die geforderte Aufgabe korrekt ausgeführt wird. Echtzeitsysteme sind Systeme, welche diese Anforderungen erfüllen und kommen in diversen Technikgebieten zur Anwendung, wie zum Beispiel (z.B.) in Signalstellanlagen, Robotik und Motorsteuerungen. Da nicht bei allen Systemen eine Einhaltung der vorher definierten Zeitspanne von Nöten ist, wird die Echtzeitanforderungen in drei Kategorien unterteilt. Dabei wird bei Echtzeit unter drei verschiedenen Kategorien unterschieden:

- **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, die aber nicht eingehalten werden muss.
  - **Beispiel:** Druckgeschwindigkeit eines Druckers.
- **Harte Echtzeitanforderung** garantiert die Erfüllung der Aufgabe zu der vorher definierten Zeitspanne.
  - **Beispiel:** Bremspedal eines Autos.

Echtzeit beschreibt somit das zeitliche Ein- bzw. Ausgangsverhalten eines Systems, allerdings nichts über dessen Realisierung. Dies kann je nach Anforderung das Echtzeitsystem rein auf Software auf einem normalen Computer implementiert sein, jedoch wird bei harter Echtzeitanforderung oftmals eine reine Hardware-Lösung mit speziellen Architekturen in Hard- und Software wie z.B. Mikrocontroller-, Field Programmable Gate Array (FPGA)- oder Digital Signal Processor (DSP)-basierte Lösungen verwendet. Bei der Implementierung eines Echtzeitsystems wird unter drei verschiedenen Umsetzungen unterschieden:

- **Feste periodische Abarbeitung:** Es wird nur eine Aufgabe, wie z.B. bei der Umwandlung von analogen Signalen zu digitalen Signalen, mit einer fixen Frequenz f abgearbeitet, welche die Reaktionszeit $f = \frac{1}{Reaktionszeit}$ erfüllt.

- **Synchrone Abarbeitung:** Im Gegensatz zur festen periodischen Abarbeitung können bei diesem Ansatz mehrere Aufgaben, wie z.B. das Abfragen mehrere Sensoren und einer unterschiedlichen Reaktion darauf, erfüllt werden. Allerdings muss dabei die kleinste geforderte Reaktionszeit unter den Aufgaben die hälfte der maximalen Laufzeit für den Gesamtdurchlauf aller Aufgaben betragen. Dieser Ansatz wird vor allem für weiche Echtzeitsysteme verwendet, weil je nach Komplexität des Systems, das Vorhandensein mehrere Codepfade oder das Warten auf Ein- oder Ausgangssignalen mit unterschiedlicher Ausführungszeit ein Nichtdeterminismus besteht.

- **Prozessbasierte Abarbeitung:** Bei diesem Ansatz können, wie auch bei der Synchronen Abarbeitung mehrere Aufgaben erledigt werden, jedoch mit viel höheren Komplexität. Dabei laufen in der Regel verschieden Prozesse gleichzeitig und mit unterschiedlicher Priorität ab, geregelt durch das Echtzeitbetriebssystem. Die minimale Reaktionszeit definiert sich durch die Zeitdauer für einen Wechsel von einem Prozess niederer Priorität zu einem Prozess höherer Priorität, anschließend beginnt erst die Abarbeitung des Prozesses mit der höheren Priorität. Durch die Unterbrechung eines Prozesses niedriger Priorität zu einem Prozess höherer Priorität wird für diesen die Erfüllung einer harten Echtzeitanforderung erzwungen, dies wird auch als Präemptives Multitasking bezeichnet. Der Wechsel wird dann eingeleitet, wenn ein definiertes Ereignis eintritt, z.B. durch den internen Timer des Prozessors oder einem externen Trigger (Interrupt) wie z.B. ein anliegendes Signal.

(Siemers, 2017; Wikipedia, 2020a)

1.2 Scheduler

Als Scheduler (zu Deutsch Steuerprogramm) wird eine Logik bezeichnet, welche die zeitliche Ausführung von mehreren Prozessen steuert und wird als präemptiver oder kooperativer Scheduler in Betriebssystemen eingesetzt. Bei einem kooperativen Scheduling wird einem Prozess die benötigten Ressourcen übergeben und wartet, bis der Prozess vollständig abgearbeitet wurde. Im Gegensatz zu einem kooperativen Scheduler kann ein präemptiven Scheduler ein Prozess die Ressourcen vor der Fertigstellung entziehen, um zwischenzeitlich diese anderen Prozessen zuzuweisen (Wikipedia, 2020d).

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

- **Stapelverarbeitungssysteme**: Ankommende Prozesse werden nach Eingangszeit in eine Reihe (Queue) angeordnet, die dann nacheinander abgearbeitet wird.

- **Interaktive Systeme**: Eingaben von Benutzen sollten schnellstmöglich abgearbeitet werden, weniger wichtige Aufgaben wie z.B. die Aktualisierung der Uhrzeit werden

unterbrochen oder erst im Nachhinein abgearbeitet.

- **Echtzeitsysteme**: Das Echtzeitsystem garantiert die Fertigstellung des Prozesses innerhalb einer definierten Zeitspanne, sofern die CPU Auslastung nicht über 100% beträgt.

Für die Anordnung selbst, wann welcher Prozess ausgeführt wird, gibt es verschiedene Arten von Scheduler.

1.2.1 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 der Gleichung 1 kann die maximale Menge an Jobs, 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 (Wikipedia, 2020c).

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

U	CPU-Auslastung
C_i	Ausführungszeiten
T_i	Periodenlänge
n	Anzahl der Jobs

Formel 1: RMS Berechnung der CPU-Auslastung (Wikipedia, 2020c)

1.2.2 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 mit der gleichen Formel 1, dies führt zur gleichen maximalen Auslastung von 69,3%, bei der keine Deadline überschritten werden (Wikipedia contributors, 2021).

1.2.3 EDF

Im Unterschied zu DMS wird bei EDF beim Eintreffen eines Prozesses mit höherer Priorität der aktuelle Prozess mit niedriger Priorität nicht unterbrochen, auch sind die Prioritäten nicht statisch, sondern werden bei jedem Scheduling-Vorgang neu vergeben. Ein Context Switch, das heißt ein Wechsel zwischen den Prozessen, 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% betragen, dabei darf die Zeitspanne der einzelnen Prozesse bis zur Deadline nur jeweils größer oder gleich der Periode des Prozesses sein (Wikipedia contributors, 2021).

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{2}$$

U	CPU-Auslastung
C_i	Ausführungszeiten
T_i	Periodenlänge
n	Anzahl der Jobs

Formel 2: EDF Berechnung CPU-Auslastung (Wikipedia contributors, 2021)

1.3 FreeRTOS

FreeRTOS ist ein Echtzeitbetriebssystem für eingebettete Systeme, welches unter der freizügigen Open-Source Lizenz MIT steht. Für eine leichte Wartbarkeit wurde FreeRTOS weitestgehend in C entwickelt, außerdem ist der Scheduler des Betriebsystems zwischen präemptiver und kooperativer Betrieb konfigurierbar um verschiedene Einsatzzwecke abzudecken (Wikipedia, 2019). Des Weiteren wurde FreeRTOS auf über 40 Mikrocontroller-Architekturen portiert („FreeRTOS“, n. d.), um eine größere Bandbreite zu erreichen. FreeRTOS hat wenig Overhead und der Kernel unterstützt Multithreading, Warteschlangen, Semaphore, Software-Timer, Mutexes und Eventgruppen („FreeRTOS Features“, n. d.). Des Weiteren stellt FreeRTOS unter der Bezeichnung ”FreeRTOS Plus”verschiedene Erweiterungen zur Verfügung, welche erweiterte Funktionen bereitstellen (FreeRTOS, 2020b).

- **FreeRTOS + TCP**: Socketbasiertes TCP/UDP/IP Interface.
- **Application Protocols**: MQTT und HTTP Anwendungsprotokolle für IoT.
- **coreJson**: Effizienter Parser für JSON

- **corePKCS#11**: Verschlüsselungs API-Layer
- **FreeRTOS + IO**: Erweiterung für Peripheriegeräte
- **FreeRTOS + CLI**: Effiziente CLI-Eingaben

Im Zuge dieser Masterarbeit wurde als einziges der Stack 'FreeRTOS + TCP' für die Verbindung zwischen Computer und Mikrocontroller.

1.3.1 FreeRTOS Scheduling Richtlinie

Der FreeRTOS Scheduler arbeitet prioritätsbasiert und beinhaltet standartmäßig hauptsächlich zwei Eigenschaften:

- **Time Slicing Scheduling Policy**: Gleich priorisierte Tasks 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 ausgeführt, mit der höchsten Priorität. 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 (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 dem Port, 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, 2020c). Der minimale Random-Access Memory (RAM)- und Read-Only Memory (ROM)-Footprint wird zwischen 6kByte und 12kByte angegeben (FreeRTOS, 2020a).

1.3.3 Task States

In FreeRTOS erzeugte Tasks können vier verschiedene Zustände haben (FreeRTOS, 2020d):

- **Running**: Wird gerade ausgeführt.
- **Ready**: Ist bereit zur Ausführung.
- **Blocked**: Wartet auf ein Event, kann nicht ausgeführt werden.
- **Suspended**: Wurde temporär deaktiviert.

Ein Ablaufdiagramm der einzelnen Zustände und deren Wechsel mit Hilfe von FreeRTOS Funktionen wird in Abbildung 1 dargestellt.

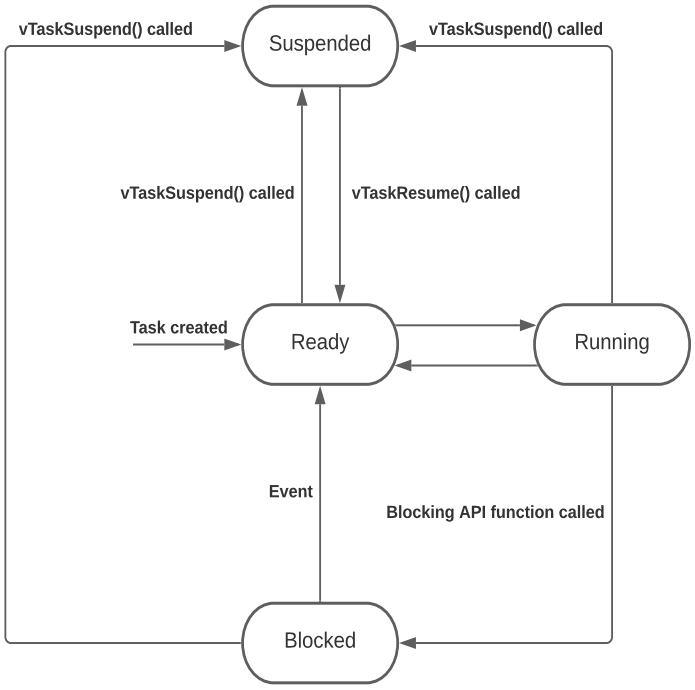


Abbildung 1: FreeRTOS Task States i.A.a: (FreeRTOS, 2020d)

1.4 Idle Task

Die FreeRTOS Idle Task wird immer dann aufgerufen, wenn keine 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 keine Task eine höhere Priorität als die Idle Task Priorität hat oder keine Task die eine höhere Priorität hat sich im 'ready' Status befindet. Sollte eine Task die gleiche Priorität mit der Idle Task teilen, wird die CPU Zeit, bei Standartkonfiguration, zwischen den zwei Tasks aufgeteilt. Des Weiteren wird die Idle Task in FreeRTOS Standartkonfiguration mit der Priorität 0, also der niedrigsten Priorität gestartet. Beim Ausführen der FreeRTOS Idle Task wird der zugewiesener Speicher von gelöschten Tasks endgültig gelöscht und sollte somit regelmäßig CPU Zeit bekommen, da ansonsten der Arbeitsspeicher überlaufen kann. Des Weiteren kann mit der Flag, wie in Programmcode 3 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, z.B. den Mikrocontroller in den Energiesparmodus zu setzen.

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

Programmcode 2: FreeRTOS Idle Task Hook

1.4.1 Interrupts

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

1.4.2 FreeRTOS Sys-Tick

Der FreeRTOS Sys-Tick, auch System Tick genannt, ist fundemental für die Taskwechsel in FreeRTOS, dieser Interrupt wird standartmäßig jede Millisekunde aufgerufen. Dieser Wert bietet eine gute Balance zwischen Taskgeschwindigkeit und Overhead von Taskwechsel. Bei jedem Sys-Tick wird die FreeRTOS Funktion 'vTaskSwitchContext()' aufgerufen, diese überprüft ob eine höher priorisierte Task unblocked, also ausführbar geworden ist und falls dies zutrifft wird ein Taskwechsel auf die höher priorisierte Task durchgeführt.

1.4.3 FreeRTOS Queues

Queues, zu Deutsch Warteschlange, werden oft, auch außerhalb von FreeRTOS für die sichere Kommunikation zwischen verschiedenen Tasks und/oder Interrupts verwendet. Diese Implementierung von FreeRTOS vermeidet Fehler wie ein Deadlock, z.B. eine höher priorisierte Task wartet auf einen Wert einer niederen priorisierten Task, wobei die niedere priorisierte Task nie ausgeführt wird. Dabei wird oft das Prinzip First Input First Output (FIFO) verwendet, um eine zeitliche Abarbeitung der Daten abzuarbeiten.

1.5 Periodische Tasks

Anwendungen auch Prozesse oder Tasks genannt, werden in Echtzeitsystemen oft periodisch, z.B. das Abrufen eines Sensorwertes, ausgeführt. Periodische Tasks setzen sich aus folgenden Parametern zusammen:

$$T_i(\Phi_i, P_i, e_i, D_i) \tag{3}$$

T_i	Task Nummer
Φ_i	Phase
e_i	Ausführzeit (Worst Execution Time (WCET))
D_i	Deadline

Formel 3: Parameter von periodischen Tasks

Mit diesen vier Parameter jeder einzelnen periodischen Tasks 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 $\Phi_i = 0$ angesehen werden.

1.6 UDP

Für die Erfüllung der Aufgabe dieser Masterarbeit musste ein Kommunikationsmittel zwischen Computer und Mikrocontroller ausgesucht werden, welches verschiedene Eigenschaften erfüllt:

- **verbindungslos**: Der Sender sendet Daten ohne eine Bestätigung des Eingangs der Daten zu bekommen. Dies hat den Vorteil einer latenzärmeren Datenübertragung.

- **wenig Overhead**: Das Empfangen und Senden sollte so wenig wie möglich Ressourcen, wie CPU-Zeit, blockieren.

- **hohe Übertragungsrate**: Es sollten so viele Daten wie möglich, innerhalb einer bestimmten Zeit übertragen.

Aus diesen Gründen wurde das Netzwerkprotokoll UDP, welches heutzutage in vielen Anwendungen, wie z.B. Internet Protocol (IP)-Telefonie und Videostreams, verwendet. Für den Verbindungsaufbau verwendet UDP Ports, welche Teil einer Netzwerk-Adresse darstellen, um versendete Daten dem gewünschten Programm an der Gegenstelle zukommen zu lassen (Wikipedia, 2020f).

1.7 FFT

Die DFT kann ein zeitdiskretes Signal in dessen Frequenzteile (Spektrum) aufspalten, wie in Abbildung 2 dargestellt. Dies ist sehr nützlich für viele Bereiche der Signalverarbeitung z.B. für das Komprimieren von Musikaufnahmen, wo nicht hörbare Frequenzen rausgefiltert werden.

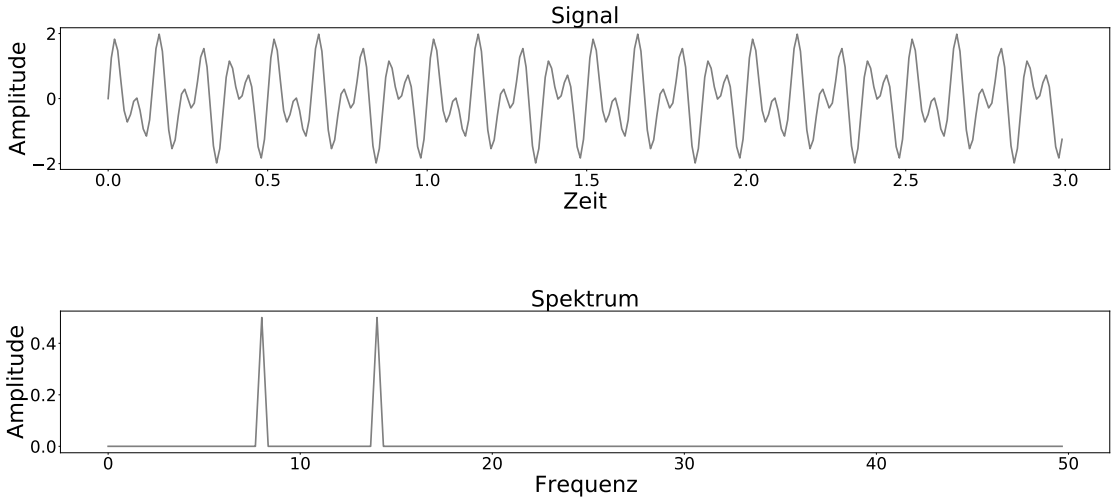


Abbildung 2: Aufspaltung eines Signals in dessen Frequenzteile

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 dazu, dass eine reine Berechnung mit Hilfe der DFT auf kleinen Mikrocontrollern häufig nicht praktikabel ist.

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

Formel 4: Berechnung der DFT Komplexität

Im Gegensatz zur DFT stellt die FFT ein Algorithmus dar, mit der der Rechenaufwand deutlich reduziert wird. Aus diesem Grund wird häufig die FFT benutzt, die nach

dem 'Teile und Herrsche'-Verfahren funktioniert und nur noch eine Komplexität von $O(N) = (N * log(N))$ besitzt (TU-Chemnitz, 2020). Durch diese enorme Zeit- und Rechenersparnis fand die FFT in vielen Bereichen, wie z.B. in Ingenieurswissenschaften, Musik, Wissenschaft und der Mathematik Verwendung (Wikipedia, 2020e). 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.

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

N Anzahl von Samples
$\in$ natürliche Zahlen

Formel 5: Rechenbedingung der FFT

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

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

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

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

Anschließen wird eine Substitution wie folgt 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)

Und berücksichtigt folgende Umformungen:

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

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

Resultieren zwei Gleichungen mit der halben Länge:

Ist die resultierende Länge der DFT wieder eine Zweierpotenz, wird die Zerlegung solange

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

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

fortgeführt, bis N Vektoren der Länge 2^0 erhält (Werner, 2012). Anschließend werden die Ergebnisse zusammengerechnet und die DFT wurde gelöst.

1.8 Plattform

Als Plattform für dieses Projekt dient ein STM32F769I-Disc0 board der Firma ST. Dieses beinhaltet einen ARM®Cortex®-M7 Prozessor und hat für die verwendete Arbeit mehr als ausreichende Rechenleistung. Für die Kommunikation zwischen Host-Computer und Mikrocontroller wurde die integrierte Ethernet Schnittstelle verwendet, des Weiteren wurde eine 'Debug-Flag' integriert, welche über den integrierten Universal Asynchronous Receiver Transmitter (USART) Debug Informationen sendet. Dieses Flag sollte allerdings nicht im Produktivbetrieb benutzt werden, da dies zu unvorhersehbaren Verzögerungen der Tasks und im Scheduling kommen kann. Für die visuelle Demo-Applikation wurden außerdem die drei, auf dem Board integrierten, Light Emitting Diode (LED)s benutzt.

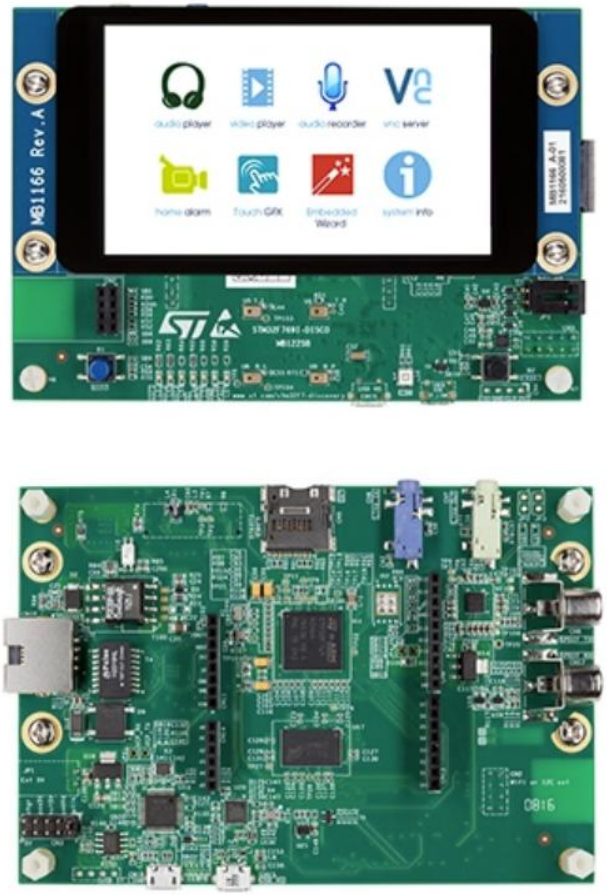


Abbildung 3: STM32F769I-Disc0 Board, Quelle: („STM32F769I-Disc0“, 2020)

2 Der EDF Scheduler

Der EDF Scheduler lässt sich mit einer einzelnen C-Datei, sowie einer Header-Datei in FreeRTOS implementieren. Dies vereinfacht die Implementierung auf andere FreeRTOS Versionen, sowie auch das portieren auf andere Plattformen. Um die Portabilität weiter zu vereinfachen, wurde der FreeRTOS SystemTick (SysTick) mit der standartmäßigen Genauigkeit von einer Millisekunde und nicht ein weiterer benutzerdefinierter Timer, wie es die Cortex-M Serie ermöglicht, benutzt. Aus dieser Entscheidungen resultiert auch die Genauigkeit des Schedulers von einer Millisekunde, wobei auftauchende Interrupts, wie die CPU Zeit des SysTick, in dieser Arbeit vernachlässigt werden. Die Periode des SysTick lässt sich unter der Datei 'FreeRTOSConfig.h' , wie in der Unterunterabschnitt 1.4.2 einstellen, jedoch ändert sich dabei auch die Genauigkeit des EDF Schedulers. Als Standart ist bei einer FreeRTOS Installation eine Millisekunde definiert, welche für den EDF Scheduler einen guten Kompromiss zwischen Applikationslaufzeit und Genauigkeit bietet.

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

Programmcode 3: FreeRTOS Idle Task Hook

2.1 Implementierung

Um 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 Unterunterabschnitt 1.3.1 bereits behandelt, prioritätsbasiert. Mit dem Aufruf der Funktion 'rescheduleEDF' am Ende jeder EDF Task, wird zwischen allen erstellten EDF Tasks, die Priorität der nächsten Task nachdem EDF-Prinzip auf die höchste Priorität gesetzt und die aktive Task auf eine niedere Priorität als die FreeRTOS Idle Task gesetzt. Anschließend wird ein Context Switch ausgelöst und die nach dem EDF-Prinzip nächste Task wird mit 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 2: EDF-Prioritäten der Tasks

Der EDF Scheduler arbeitet mit vier unterschiedlichen Prioritäten, wie in Tabelle 2 dargestellt und inkrementiert bei der Implementierung die Idle Task von der niedrigs-

ten Priorität ’0’ auf die zweit niedrigste Priorität ’1’ in der FreeRTOS Umgebung. Alle EDF Tasks die nicht ausgeführt werden sollen, erhalten die Priorität 0, das führt dazu, 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, wie in Programmcode 3 beschrieben, aufgerufen. Die Task, welche aktuell ausgeführt wurde und nun 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 ’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 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.

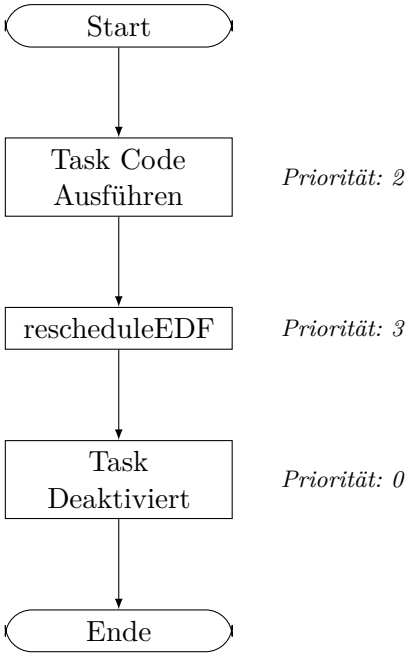
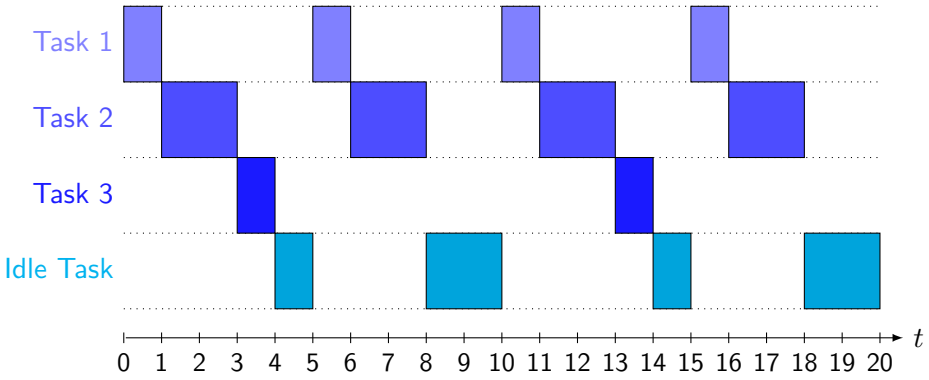


Abbildung 4: EDF Task Ausführungsablauf

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 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

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.

2.2 Erstellung einer EDF Task

Für die Erstellung einer EDF Task muss die 'createEDFTask'-Funktion , mit allen Parametern die in Tabelle 3 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 **reertos!** (**reertos!**) Task erstellt.

Parameter	Datentyp	Beschreibung
taskCode	TaskFunction_t	Funktionname 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 3: Task Parameter von der Funktion 'createEDFTask'

Der Beispielaufruf der 'createEDFTask' ist in Programmcode 4 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 rescheduleED
        Function must be called */
        rescheduleEDF();
    }
}

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 4: createEDFTask Beispiel

2.3 Löschen einer EDF Task

In manchen Applikationen ist das Löschen von EDF Tasks von nöten und wurde aus diesem Grund ebenfalls implementiert. Dabei muss die Funktion 'deleteEDFTask' und den String Namen der zu löschenden Task aufgerufen werden. Bei dem Aufruf wird die zu löschende Task aus der EDF Array Struktur entfernt, anschließend wird die Task in

FreeRTOS gelöscht. Ein Beispielaufruf, angelehnt an das Beispiel aus Programmcode 4 ist in Programmcode 5 dargestellt.

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

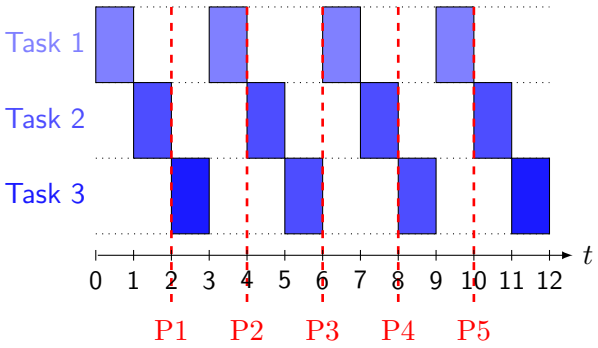
Programmcode 5: deleteEDFTask Beispiel

2.4 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. Wie auch schon eingangs erwähnt, sorgt das Verwenden des FreeRTOS SysTick für den EDF Scheduler für eine breite Kompatibilität auf verschiedenen CPU Architekturen.

2.5 Fehlerbehandlung

Falls eine Task ihre Deadline überschritten hat oder die Zeit für die Ausführung nicht mehr reicht, werden die EDF Parameter angepasst. 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 mit den Task dargestellt, können aufgrund der Task Eigenschaften immer nur zwei Tasks in einer Periode, welche mit roten Linien eingezeichnet sind, 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 können. Eine andere Möglichkeit der Fehlerbehandlung wäre das unterbinden einer der drei Tasks, um den anderen zwei Tasks dauerhaft das Einhalten Ihrer Deadlines zu ermöglichen.

2.6 Debug Mode

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

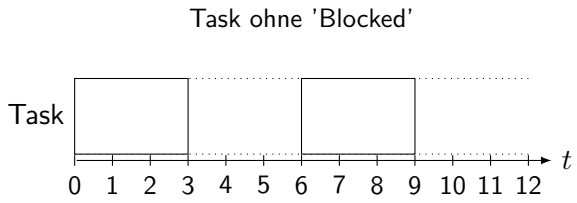
- **USART Schnittstelle**: Über die USART Schnittstelle wird die erfolgreiche Konfiguration der Hardware, Start des FreeRTOS Schedulers, sowie auch der Start jeder EDF Funktion mitgeteilt. Des Weiteren werden nicht eingehaltene Deadlines und die Anzahl der Deadline Fehler, sofern vorhanden, mitgeteilt.

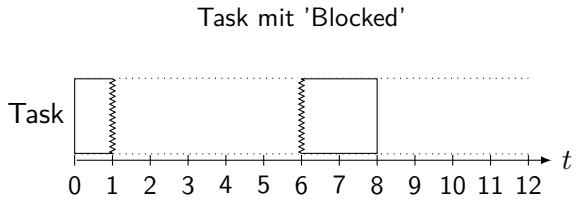
- **Erweiterte EDF Array Struktur:** Die EDF Array Struktur wird um folgende fünf Parameter erweitert:

- **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 Stopzeit der Task in SysTicks
- **deadlineErrorCounter**: Enthält die Anzahl der nicht eingehaltenen Deadlines

2.7 Limitationen

Eine FreeRTOS Task kann, wie in Unterunterabschnitt 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. die 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 vergrößerte sich nur gering, allerdings wurde dann entschieden, dass es bei einem Taskhandling nach dem EDF-Prinzip keinen Sinn macht, dies zu unterstützen. 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

Wie in Abbildung 7 zu sehen, sind zwei Tasks mit den identischen EDF Parametern 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.

3 Demo Applikationen

Ein Teil der Aufgabenstellung beinhaltet das Erstellen von zwei Demo Applikationen, welche die Funktion des EDF Schedulers unter Beweis stellen.

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, besitzen jedoch die gleichen EDF Parameter, welche in Tabelle 4 dargestellt sind.

Tasks	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 simuliert und optisch dargestellt 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.

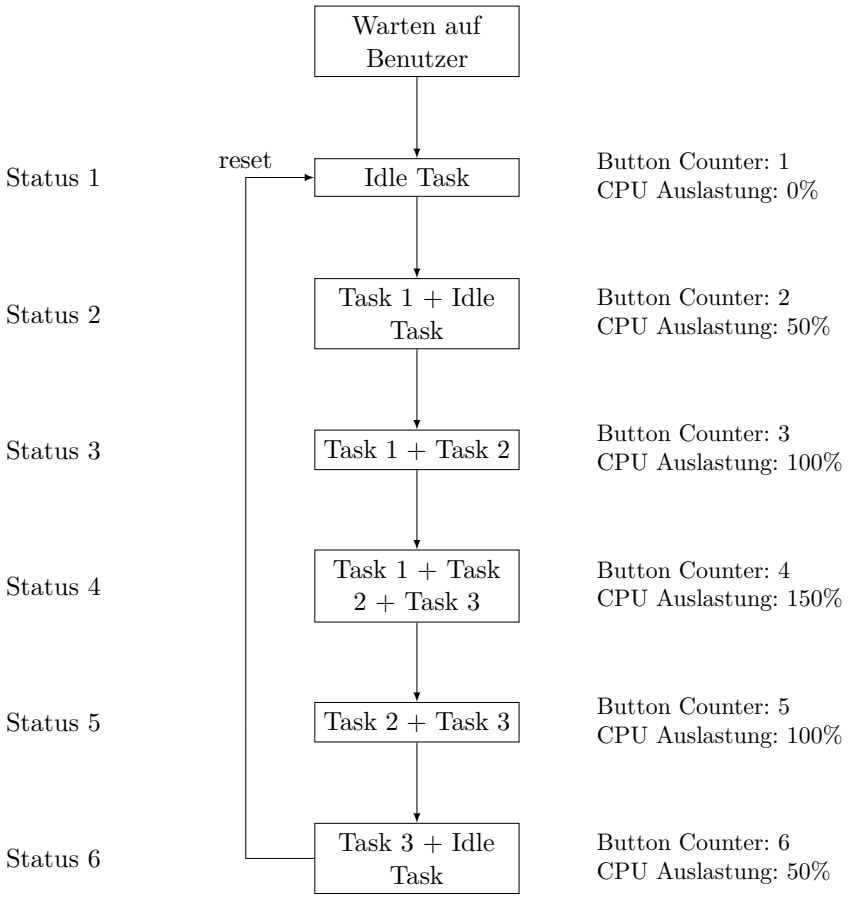


Abbildung 8: Blinky Demo Ablaufdiagramm

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 noch 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 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 USART 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 noch 100%, aus diesem Grund wird die Idle Task auch nicht aufgerufen.

- **Status 5**: Ein weiteres Drücken des Buttons löscht 'Task 2' und 'Task 3' teilt sich mit der 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 FFT Demo

4 Fazit und Ausblick

Als Ergebnis dieser Thesis wurde ein EDF Scheduler in FreeRTOS implementiert.

Abbildungen

Tabellen

Literatur

FreeRTOS [[Online; Stand 9. November 2020]]. (n. d.). %5Curl%7Bfreertos.org%7D

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

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

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

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

FreeRTOS Features [[Online; Stand 9. November 2020]]. (n. d.). %5Curl%7Bfreertos.org/features%7D

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.

Siemers, P. D. C. (2017). Echtzeit: Grundlagen von Echtzeitsystemen [[Online; Stand 9. November 2020]]. %5Curl%7Bhttps://www.embedded-software-engineering.de/echtzeit-grundlagen-von-echtzeitsystemen-a-669520/%7D

STM32F769I-Disc0 [[Online; Stand 18. Januar 2021]]. (2020). https://www.digikey.de/product-detail/de/stmicroelectronics/STM32F769I-DISCO/497-16524-ND/6004739

TU-Chemnitz. (2020). User Datagram Protocol — Wikipedia, Die freie Enzyklopädie [[Online; Stand 12. Januar 2021]]. Die%20Fourier-Transformation

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

Wikipedia. (2019). FreeRTOS — Wikipedia, Die freie Enzyklopädie [[Online; Stand 9. November 2020]]. %5Curl%7Bhttps://de.wikipedia.org/w/index.php?title=FreeRTOS&oldid=192678143%7D

Wikipedia. (2020a). Echtzeitsystem — Wikipedia, Die freie Enzyklopädie [[Online; Stand 9. November 2020]]. %5Curl%7Bhttps://de.wikipedia.org/w/index.php?title=Echtzeitsystem&oldid=200724790%7D

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

Wikipedia. (2020c). Rate Monotonic Scheduling — Wikipedia, Die freie Enzyklopädie [[Online; Stand 11. Januar 2021]]. https://de.wikipedia.org/w/index.php?title=Rate_Monotonic_Scheduling&oldid=196180419

Wikipedia. (2020d). Scheduling — Wikipedia, Die freie Enzyklopädie [[Online; Stand 9. November 2020]]. %5Curl%7Bhttps://de.wikipedia.org/w/index.php?title=Scheduling&oldid=202856507%7D

Wikipedia. (2020e). 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. (2020f). User Datagram Protocol — Wikipedia, Die freie Enzyklopädie [[Online; Stand 12. Januar 2021]]. https://de.wikipedia.org/w/index.php?title=User_Datagram_Protocol&oldid=204278447

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

Wikipedia contributors. (2021). Earliest deadline first scheduling — Wikipedia, The Free Encyclopedia [[Online; accessed 11-January-2021]]. https://en.wikipedia.org/w/index.php?title=Earliest_deadline_first_scheduling&oldid=999660324