Embedded Realtime OS FreeRTOS auf STM32F4

Michael Ebert
Ad-hoc Networks GmbH
ebert@ad-hoc.network

Christoph Bläßer
Bundesamt für Sicherheit in der Informationstechnik
christoph.blaesser@gmx.de

Stichwörter
RTOS, FreeRTOS, ARM , STM32, Real Time.

KURZFASSUNG
Im Rahmen dieser Arbeit wird das Echtzeitbetriebssystem FreeRTOS vorgestellt. Hierzu werden zu Beginn die allgemeinen Eigenschaften für Echtzeitbetriebssysteme beschrieben. Im Verlauf des Textes wird an ausgewählten Beispielen dargestellt, wie FreeRos diese Anforderungen berücksichtigt und durch geeignete Programmfunktionen umsetzt.

WIP: Gliederung hat sich geändert

1. GRUNDLAGEN

1.1 Anforderung Desktop Betriebssystem vs. Anforderung Echtzeit Betriebssystem

Desktop Betriebssysteme verwalten den Hardwarezugriff und stellen sicher, dass eingesetzte Software die benötigte Rechenzeit zur Verfügung gestellt bekommt. Gleichzeitig regeln Sie den Hardwarezugriff und organisieren den konkurrierenden Zugriff, beispielsweise auf Netzwerkkarten und Festplatten. Sie stellen Funktionen für die Interprozesskommunikation bereit und übernehmen grundlegende Aufgaben wie die Organisation von Arbeitsspeicher. Im Gegensatz zu einem gewöhnlichen Desktop Betriebssystem liegt der Anwendungsfokus bei Echzeit Betriebssystemen nicht auf der direkten Userinteraktion. Die wenigsten Echtzeitbetriebssysteme bieten eine Benutzeroberfläche. Die Interaktion mit der Umwelt geschieht durch spezielle Hardware und ist zumeist für genau eine Aufgabe ausgelegt. Viele Funktionen die eine Desktop Betriebssystem übernimmt wie z.B. das Verwalten von konkurrierenden Zugriffen auf externe Geräte müssen vom Entwickler selbst übernommen werden. Ein Echtzeitbetriebssystem bietet bei weitem nicht die Funktionalitäten eines Desktop Betriebssystems, das liegt zum Einen an der Art von Zielsystemen, die zumeist nur einen beschränkten Speicher aufweisen und Zweitens an der Tatsache das die vom Echtzeitbetriebssystem bereitgestellten Funktionen deterministisch sein müssen.

WIP: Gefällt insgesamt noch nicht, benötigen wir Anforderungen an ein Destkop Betriebssystem ?? Echzeitbetriebsysteme unterscheiden sich erheblich von normalen Betriebssystemen. Wie schon angesprochen verwaltet ein RTOS nur die Zeit (Scheduler) + Speicher (Memory Allocation)

1.2 Echtzeitsysteme und Echtzeitbetriebssysteme

Mit der steigenden Leistungsfähigkeit von modernen μ Prozessoren, steigen auch die Anforderungen an die Software die auf diese Systeme aufsetzt. Viele dieser Systeme verlangen trotz ihrer Komplexität, dass Teile des Programmablauf in bestimmten zeitlichen Grenzen ausgeführt wird und somit vorhersehbar und deterministisch sind. Systeme die eine solche Anforderung unterliegen werden Echtzeitsysteme genannt. Echtzeitsysteme unterliegen einer weiteren Unterteilung in weiche Echtzeitsystem (soft realtime systems) und harte Echtzeitsysteme (hard realtime systems). Ein weiches Echtzeitsysteme soll eine Aufgabe in den vorgegeben zeitlichen Grenzen ausführen, ein überschreiten ist aber erlaubt und führt nicht unmittelbar zu einem Fehler. Ein hartes Echtzeitsystem hingegen muss die gestellte Aufgabe in den vorgegebenen Grenzen ausführen. Eine Überschreitung macht das System unbrauchbar. Einige Beispielsysteme und deren Echtzeitzuordnung wir in Tabelle 1 gezeigt. Um die grundsätzliche Funktionalität eines Echtzeitbetriebssystems zu erläutern, müssen zu erst die Grundmodelle für den Programmablauf eingebetter Systeme beschrieben werden. Der Programmablauf eingebetteter Systeme lässt sich auf drei Modelle zurückführen (Abbildung 1). Eingebettete Anwendungen können in einer einzigen Schleife (mit oder ohne Interrupt Unterbrechungen) laufen oder aber in event-gesteuerten nebenläufigen eigenständigen Programmabschnitten (Thread oder Task[1]) ausgeführt werden. Die nebenläufige Ausführung der unterschiedlichen Programmsegmente ist nur durch einen RTOS-Kernel (Scheduler) zu erreichen. Ein RTOS Kernel abstrahiert Timing Informationen und kümmert sich darum, dass die nächste Task rechtzeitig ausgeführt wird. Der Entwickler ist dafür verantwortlich, dass die Task die gewünschte Aufgabe im zeitlichen Rahmen ausführt. Wie sichergestellt werden kann, dass eine Task harten oder weichen Echtzeitanforderungen entspricht wird Abschnitt 5 beschrieben. Für viele kleine Anwendungen kann die Nutzung einer einzigen Schleife durchaus sinnvoll sein, sollten beispielsweise die Ressourcen so knapp sein, dass ein Overhead an Funktionalität ausgeschlossen werden muss. Ein großer Nachteil der „einschleifen Variante“ ist die permanente Nutzung des Prozessors. Besonders bei akkubetriebenen Geräten wie IoT Devices oder Mobiltelefonen wird sehr genau auf die Energieaufnahme geachtet. Ein RTOS bietet hingegen Funktionen mit denen sehr leicht ermittelt werden kann, ob ein Gerät in einen Schlafmodus wechseln kann, dies wird in Abschnitt 2.9 an Beispielen von FreeRTOS und einem ARM μProzessor demonstriert. Neben der Echtzeitfähig gibt es aber noch viele weitere Vorzüge für den Einsatz eines Echtzeitbetriebssystems. Durch das Herunterbrechen der Anwendung in Task entstehen viele kleine Module, die jeweils eine kleine Teilaufgabe des Gesamtsystems übernehmen.Durch ein sauber definiertes

[1]Nachfolgenden wird Task benutzt, da dies der geläufige Begriff bei FreeRTOS ist. In der Literatur zu Echtzeitsystemen ist der Begriff nicht exakt definiert.

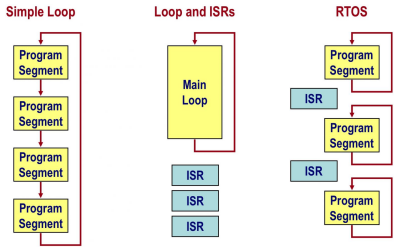

Abbildung 1. Übersicht Programmabläufe
Quelle: `http://www.embedded.com/`

Interface zur Kommunikation der Tasks, lässt sich die Entwicklungsarbeit leicht auf mehrere Entwickler-Teams verteilen. Dies ermöglicht auch den Einsatz von agilen Entwicklungsmethoden wie Scrum in der Entwicklung von eingebetteten Systemen.

WIP: Christophs Part integrieren, Überleitung FreeRTOS

2. FREERTOS

2.1 Geschichte

WIP: Christoph

2.2 Zielsysteme STM32F4 (ARM Cortex M3)

32 bit Prozessor - Funktionsübersicht, Hinweis Port Teil von FreeRTOS

2.3 Entwicklungsumgebung

FreeRTOS ist im Prinzip nicht an eine spezielle Entwicklungsumgebung gebunden. Bevor eine Entwicklung beginnt ist es dennoch ratsam sich einen Überblick über die verfügbaren IDEs[2] zu machen. Der wichtigste Punkt der hier zu nennen ist, ist das Debugging. Da ein Echtzeitbetriebssystem eine weitere Abstraktionsebene hinzufügt und wie eine Art Middleware fungiert, lassen Sich viele RTOS spezifische Funktionen und Eigenschaften wie Queues, Task Stacks etc. nur mühsam mit einem Debugger wie GDB oder OpenOCD untersuchen. Viele der marktgängigen Entwicklungsumgebungen bieten daher spezielle RTOS aware Pakete, so dass ein einfacherer Zugriff auf RTOS Objekte und Eigenschaften möglich ist. Wie die RTOS awareness beim Debugging eingesetzt wird und welche Funktionalitäten sie einem Entwickler bietet wird in Abschnitt 4 aufgezeigt. Ein weiterer Punkt der bei der Auswahl der IDE getroffen werden muss sind die Kosten. Bei Propritäre IDEs können oft mehrere tausend Euro Lizenzkosten anfallen, bieten aber den Vorteil der nahtlosen Einbindungen von μProzessoren und Echtzeitbetriebssystem (RTOS awareness). Bei der Entwicklung von ARM uProzessoren sind hier Keil (Arm), IAR Workbench und True Studio (Atollic) zu nennen. Diese Entwicklungsumgebungen lassen sich zum Teil auch frei verwenden, allerdings mit starken Einschränkungen wie z.B. maximal Codesize. Auf der nicht proprietären Seiten steht Eclipse CDT zur Verfügung, es ist komplett frei in der Verwendung und hat keine Beschränkungen. Nachteil ist hier, dass die Integration nicht so einfach ist wie bei den proprietären IDEs. RTOS awarness wird bei Eclipse durch die Installation weiterer Plugins erreicht. Ein weiterer Nachteil ist, dass es keine Beispielprojekte für Eclipse CDT und FreeRTOS zur Verfügung stehen, daher müssen Projekte von Grund auf selbst konfiguriert und installiert werden. Da im Laufe dieser Arbeit Eclipse CDT für alle Beispiele verwendet wird, wird in Abschnitt 2.4 das Aufsetzen einer Basiskonfiguration erklärt.

[2] Integrated Development Environment

2.4 Einrichten und Konfiguration

WIP: Eclipse CTD, RTOS AwareNess , Debugger, File-Structure

- https://eclipse.org/cdt/
- https://launchpad.net/gcc-arm-embedded
- http://gnuarmeclipse.github.io/plugins/download/
- http://gnuarmeclipse.github.io/windows-build-tools/
- http://gnuarmeclipse.github.io/debug/jlink/
- http://gnuarmeclipse.github.io/debug/openocd/
- http://freescale.com/lgfiles/updates/Eclipse/KDS
- Thread Aware
- Beispiel Projekt
- STM32 Cube MX
- FreeRTOS.org

2.5 Memory Allocation

Beim Erzeugen von RTOS Objekten wie Tasks, Queues oder Semaphore wird Speicher im RAM benötigt. Für die dynamische Speicherverwaltung wird in C und C++ gewöhnlich die Standard C Funktionen `malloc()` und `free()` verwendet. Die Funktion `malloc()` dient zur Allozierung von freiem Speicher und `free()` zur Freigabe von alloziertem Speicher. Für Echtzeitsysteme die auf einem RTOS aufsetzen, sind diese Funktionen aufgrund der folgende Eigenschaften[2] ungeeignet[3]:

- nicht thread safe
- nicht deterministisch
- tendieren zur Fragmentierung des RAM
- schwer zu debuggen
- Bibliotheksfunktionen benötigen viel Speicher

Des Weiteren sind für einige Einsatzgebiete von embedded Anwendungen Zertifikate erforderlich. Speziell in sicherheitskritischen Anwendungen (medical, military) ist die dynamische Speicherverwaltung als eine potentielle Fehlerquelle auszuschließen. Für einen solchen Fall bietet FreeRTOS ab Version 9.0 die Möglichkeit der statischen Speicherallozierung, diese werden wir am Ende dieses Abschnitts betrachten. In FreeRTOS werden `malloc()` und `free()` durch die Funktionen

```
void *pvPortMalloc( size_t xSize );
und
void vPortFree( void *pv );
```

[3] Heap3 stellt hier eine Ausnahme dar

Beispiel	Echtzeit Typ
Tastatur Controller	Soft Realtime
Echtzeit Media Streaming	Soft Realtime
Controller CD Laufwerk	Hard Realtime
Airbag System	Hard Realtime

Tabelle 1. Beispiele Echzeitsystem

ersetzt. Dies hat den Vorteil, dass die Implementierung dieser Funktionen an die jeweilige Anwendung angepasst werden kann. Grundsätzlich bietet FreeRTOS fünf unterschiedliche Beispiel Implementierungen (Heap1 bis Heap5), siehe Abbildung 2. Diese stellen prinzipiell schon die ge-

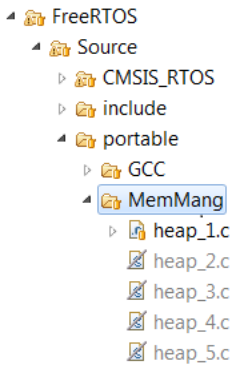

Abbildung 2. Einbindung von Heap1. Heap2 bis Heap5 sind vom Build ausgeschlossen

läufigsten Implementierungen zur Speicherverwaltung. Es bleibt aber auch weiterhin die Möglichkeit seine eigene Speicherverwaltung zu implementieren. In dieser Arbeit werden wir Heap1 etwas tiefer betrachten um ein grundsätzliches Verständnis für die FreeRTOS Speicherverwaltung zu bekommen. Heap2 - Heap 5 werden nur kurz beschrieben und können im Detail in [2][1] nachgelesen werden. Wie schon am Anfang dieses Abschnitts beschrieben, werden für alle RTOS Objekte Speicher benötigt, der Speicher für Objekte wie Semaphore und Tasks wird automatisch in den Erzeugerfunktionen alloziert, in dem intern die Funktion *pvPortMalloc()* aufgerufen wird. Die Funktion `xTaskCreate()` dient zum Erzeugen einer FreeRTOS Task. Listing 1 zeigt wie `xTaskCreate()` die Funktion `pvPortMalloc()` verwendet (Zeile 5, 11) um Speicher für den Stack und den Task Control Block zu allozieren. Alle Objekte die mittels pvPortMalloc() alloziert werden, darunter auch der Kernel selbst, teilen sich einen gemeinsamen Adressraum, siehe Abbildung 3. Eine Speicherzugriffsverletzung ist somit durchaus möglich. In Abschnitt 2.5.2 wird gezeigt welche Möglichkeit der STM32F4 und FreeRTOS bieten um Speicherzugriffe sicherer zu gestalten.

2.5.1 FreeRTOS Heap Implementierungen

Bevor Objekte erzeugt werden können, muss ein Pool an Speicher für die Objekte definiert werden. Die einfachste Form einen Memory Pool zu erzeugen ist ein Array. In FreeRTOS nennt sich dieses Array ucHeap.
Die Größe des Heaps wird durch das Präprozessor-Define `configTOTAL_HEAP_SIZE` konfiguriert. Die Gesamtgröße berechnet sich wie folgt

$$MaxHeapSize = \texttt{configTOTAL\_HEAP\_SIZE} * Wortbreite$$

In unserem Fall ist die Wortbreite 32 bit. Heap1 ist sehr

```
1 StackType_t *pxStack;
2 /* Allocate space for the stack
3 used by the task being created. */
4 pxStack =
5 ( StackType_t * ) pvPortMalloc(( ( ( size_t )
      usStackDepth )
6 * sizeof( StackType_t ) ) );
7
8 if( pxStack != NULL )
9 {
10  /* Allocate space for the TCB. */
11  pxNewTCB = ( TCB_t * ) pvPortMalloc( sizeof(
      TCB_t ) );
12
13  if( pxNewTCB != NULL )
14  {
15   /* Store the stack location in the TCB. */
16   pxNewTCB->pxStack = pxStack;
17  }
18 // ...
19 }
```

Listing 1. xTaskCreate() memory allocation. Aus Task.c

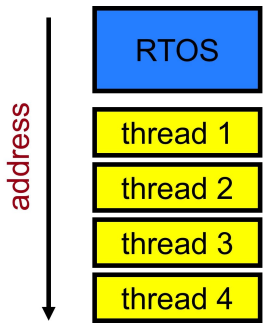

Abbildung 3. Adressraum FreeRTOS und Tasks
Quelle: Colin Walls - Embedded.com

einfach, es deklariert lediglich die Funktion pvPortMalloc(). Die Funktion pvPortFree() wird nicht ausimplementiert. Bei vielen embedded Anwendungen wird der Speicher für die benötigten Objekte vor dem Start des Schedulers erzeugt. Eine spätere Freigabe von diesen Resourcen ist nicht nötig, da die Objekte über die gesamte Laufzeit des Programms bestehen sollen. Genau für solche Anwendungen steht Heap1 zur Verfügung. Abbildung 4 zeigt wie sich der Speicher nach dem Erzeugen von zwei Tasks aussieht. Für jede Task wird ein TCB und ein Stack erzeugt, die Speicherobjekte liegen direkt hintereinander, da pvPortFree() nicht implementiert ist, kommt es auch nicht zu einer Fragmentierung des Speichers. Diese lineare Speicherzuweisung gilt für alle Objekte die mittels pvPortMalloc() alloziert werden, dazu gehören sowohl RTOS spezifische Objekte als auch Objekte die durch den Benutzer

```
static uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
```

erzeugt werden. Nachfolgend ein Kurzüberblick über die

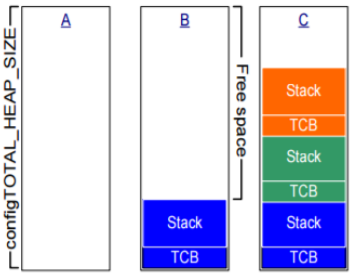

Abbildung 4. Beispiel Speicherbelegung nach drei Instanziierung von Tasks
Quelle: FreeRTOS.org

nicht beschriebenen Beispiel Implementierungen.

- Heap2 - Ähnlich Heap1. Erlaubt Speicherfreigabe. Best Fit Algorithmus zur Speicherallozierung.
- Heap3 - Verwendet C Library Malloc() und free() und deaktiviert den Scheduler zur Speicherallozierung.
- Heap4 - Ähnlich Heap1 und Heap2. Verwendet First Fit Algorithmus zur Speicherallozierung. Verbindet mehrer kleinere Speicherblöcke in einen großen. Minimiert Speicherfragmentierung
- Heap5 - Gleicher Algorithmus wie Heap4 allerdings können mehrere Memory Pools erzeugt werden.

2.5.2 Memory Protection
STM32F4 spezifisch, MPU vorhanden

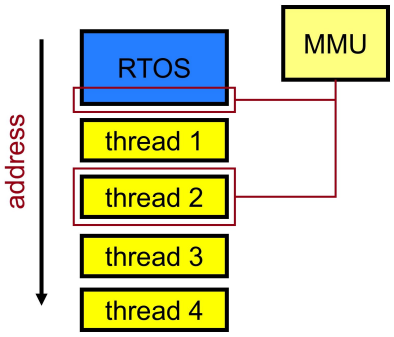

Abbildung 5. Adressraum FreeRTOS und Tasks mit Memory Protection
Quelle: Colin Walls - Embedded.com

2.5.3 Static Memory Allocation

2.6 Scheduling
Blocked - Running :D FSM fertig

2.7 Intertask Kommunikation
Queues, Semaphore, Notify, Event Groups

2.8 Interrupt Handling
Deamon Task,

2.9 Low Power Modes auf Stm32F4
Echtzeitbetriebssysteme kommen immer häufiger in akkubetriebenen embedded Systemen zum Einsatz. Solche Systeme verlangen eine effiziente Nutzung der Energieressourcen um einen möglichst langen Betrieb zu gewährleisten. Bezogen auf den uProzessor gibt es im Prinzip zwei Wege Energie einzusparen:

- Heruntertakten des uProzessors.
- Das System schlafenlegen, wenn keine weiteren Aufgaben anstehen.

Das Heruntertakten des uProzessors ist unabhängig vom Einsatz eines RTOS, daher werden wir hier nur den zweiten Punkt genauer betrachten, das Schlafenlegen des uProzessors. Abbildung 6 zeigt wie sich die Stromaufnahme

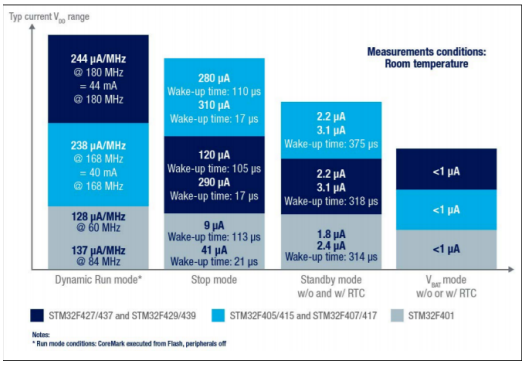

Abbildung 6. Energieaufnahme für STM32F4 in SleepModes
Quelle: STM32F4 - Power Modes

beim STM32F4 von 40mA (@168 MHz) auf 2,2μA im Tiefschlafmodus reduzieren lässt. In einfachen Anwendung ist der Zustand in dem ein Gerät schlafen gehen kann, relativ leicht zu ermitteln. In komplexen Systemen die auf einem Echtzeitbetriebssystem wie FreeRTOS aufsetzen und mehrere Task möglicherweise auf unterschiedliche Ressourcen warten, wird es schon schwierig. In diesem Abschnitt wird gezeigt welche Funktionen FreeRTOS zur Verfügung stellt, um einen energieeffizienten Betrieb zu gewährleisten. Eine Möglichkeit ist die Idle - Hook Funktion. Wie bereits in Abschnitt 2.6 beschrieben, wird die IDLE Task von FreeRTOS aktiviert, sobald sich alle User-Tasks im Blocked Zustand befinden. Durch konfigurieren des Präprozessor-Defines

```
#define configUSE_IDLE_HOOK    1;
```

kann die Idle-Hook Funktion aktiviert werden. Diese wird immer aufgerufen, sobald die Idle Task in den Zustand Running wechselt. Die Funktionalität der Idle-Hook Funktion kann frei vom Entwickler implementiert werden. Listing 3 zeigt Pseudocode zu einer beispielhaften Implementierung der Idle Hook Funktion. Bevor das System schlafen gelegt werden kann müssen alle GPIOS und IRQs konfiguriert werden, so dass das System nicht unnötiger weise aufwacht. Des Weiteren werden alle nicht benötigten GPIOS auf Analog gestellt um Energie zu sparen. Als einzige Interrupt-Quelle wird hier eine externe RTC konfiguriert. Mit dem Aufruf von HAL_PWR_EnterSTOPMode() wird der uProzessor in den Schlafmodus versetzt. Die Funktion wird erst wieder verlassen sobald der externe Interrupt

```
1 static portTASK_FUNCTION( prvIdleTask,
      pvParameters )
2 {
3  /* Stop warnings. */
4  ( void ) pvParameters;
5
6  /** THIS IS THE RTOS IDLE TASK - WHICH IS
      CREATED AUTOMATICALLY WHEN THE
7  SCHEDULER IS STARTED. **/
8
9  for( ;; ) {
10 //skipped some code
11 #if ( configUSE_IDLE_HOOK == 1 )
12 {
13  extern void vApplicationIdleHook( void );
14  /* Call the user defined function from within
      the idle task.  This
15  allows the application designer to add
      background functionality
16  without the overhead of a separate task.
17  NOTE: vApplicationIdleHook() MUST NOT, UNDER
      ANY CIRCUMSTANCES,
18  CALL A FUNCTION THAT MIGHT BLOCK. */
19
20  vApplicationIdleHook();
21 }
22 //guess what.. skipped more code
23 }
```

Listing 2. Aufruf der IdleTask Hook Funktion. Aus Task.c

der RTC ausgelöst wurde. Danach werden alle GPIOs rekonfiguriert. Ein weiterer Schritt der noch unternommen werden muss, ist das informieren einer User-Task z.B. mittels Notify oder Message, so dass das System nicht beim nächsten Tick Interrupt wieder die Idle Task aktiviert. Nachteil dieser Variante ist, dass die Nutzung von Software Timer nicht mehr möglich ist. FreeRTOS würde die Idle Hook Funktion auch aufrufen und sich schlafen legen, wenn noch Software Timer aktiv sind. Die Nutzung von absoluten Zeiten ist ebenfalls nicht mehr möglich, da nach der Deaktivierung des Tick Interrupts der Tickcount nicht mehr korrekt ist. Abhilfe schafft hier eine weitere Funktionalität die FreeRTOS zur Verfügung stell, den sogenannten Tickless Idle Mode. To be continued ....

```
1 extern "C" void vApplicationIdleHook( void ){
2  /* Systick Interrupt deaktivieren */
3  SysTick->CTRL &= ~SysTick_CTRL_TICKINT_Msk;
4  //RTC konfigurieren
5  setRTCWakeupTime();
6  //externen Interrupt durch RTC aktivieren
7  enableRTCInterrupt();
8  //deaktiviere alle anderen Interrupt Quellen
9  deactivateExternalDevices();
10 setAllGPIOsToAnalog();
11 disableGPIOClocks();
12 //MCU stoppen und schlafen ZzZZz
13 HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON,
      PWR_STOPENTRY_WFI);
14 //Aufgewacht...the show must go on
15 //aktiviere Systick
16 SysTick->CTRL |= SysTick_CTRL_TICKINT_Msk;
17 //reaktiviere GPIO Clocks
18 enableGPIOClocks();
19 //reaktiviere Externe Interrupt Quellen
20 enableExternalInterrupts();
21 }
```

Listing 3. Beispiel Implementierung der Idle Hook Funktion

2.10 FreeRTOS in der Praxis - Ein real System

Ad-hoc System?

3. KOMPLEXITÄT DURCH NEBENLÄUFIGKEIT

Probleme die bei der Entwicklung auftreten, Häufige Bugs, Speicherüberlauf (Stacktools)

4. DEBUGGING VON ECHTZEITSYSTEMEN

TracerLyzer, RtosAwarenes, ThreadAwareness, Hardware Debugging Probes

5. ECHTZEITANALYSE

Uff :)

6. ZUSAMMENFASSUNG

Literatur

1. R. Barry. Freertos implemenation - advanced.

2. R. Barry. *Mastering the FreeRtos Real time Kernel*. Real time Engineers Ltd., pre-release 161204 edition edition, 2016.