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
FreeRTOS, RTOS, 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 FreeRTOS diese Anforderungen berücksichtigt und durch geeignete Programmfunktionen umsetzt.

1. GRUNDLAGEN ECHTZEITSYSTEME

1.1 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 solchen Anforderungen unterliegen werden Echtzeitsysteme genannt. Echtzeitsysteme unterliegen einer weiteren Unterteilung in Echtzeitsysteme mit weicher Echtzeitanforderungen (soft realtime systems) und Echtzeitsysteme mit harter Echtzeitanforderung (hard realtime systems). Ein weiches Echtzeitsystem soll eine Aufgabe in den vorgegeben zeitlichen Grenzen ausführen, ein überschreiten der zeitlichen Grenzen ist grundsätzlich nicht erlaubt, führt aber nicht unmittelbar zu einem Fehler oder einem Versagen des Gesamtsystems. Ein hartes Echtzeitsystem hingegen muss die gestellte Aufgabe in den vorgegebenen Grenzen ausführen. Durch eine Überschreitung wird das System unbrauchbar und führt dazu, dass das System nicht im vorgesehenen Szenario eingesetzt werden kann. Dabei ist ausdrücklich zu beachten, dass Echtzeit nicht bedeutet, dass ein Programm besonders schnell ausgeführt wird. Die Ausführung eines Programms kann beispielsweise auch gewollt langsam sein und gerade deshalb den gestellten Echtzeitanforderung genügen. Einige Beispielsysteme und deren Echtzeitzuordnung wird in Tabelle 1 gezeigt. Um die grundsätzliche Funktionalität eines Echtzeitbetriebssystems zu erläutern, werden zuerst die Grundmodelle für den Programmablauf eingebetteter Systeme beschrieben. 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 geeigneten Scheduler, welcher Teil eines RTOS Kernels ist, zu erreichen. Ein RTOS Kernel abstrahiert von der zugrunde liegenden Hardware und ermöglicht weitergehende Steuerung, beispielsweise durch Verwaltung von Timing Informationen. Hierdurch wird sichergestellt, dass die nächste Task rechtzeitig ausgeführt wird. Der Entwickler ist dafür verantwortlich, dass die Task die gewünschte Aufgabe im zeitlichen Rahmen ausführt. Durch den Einsatz des RTOS Kernels kann der Entwickler jedoch auf Spezifika der Hardware verzichten und die Funktionen des Kernels verwenden. Wie sichergestellt werden kann, dass eine Task harten oder weichen Echtzeitanforderungen entspricht wird Abschnitt 3.2 beschrieben. Für viele kleine Anwendungen kann die Nutzung einer einzigen Schleife durchaus sinnvoll sein, wenn beispielsweise die Ressourcen so knapp sind, dass ein Overhead durch zusätzliche Verwaltungsfunktionen ausgeschlossen werden muss. Ein großer Nachteil der „einschleifen Variante“ ist die permanente Nutzung des Prozessors, auch „processor hogging“ oder „CPU hogging“ genannt. Um den Prozessor in dieser Variante in einen Energiesparmodus zu versetzen sind umfangreiche Kenntnisse über den Prozessor sowie eine sehr strukturierte Programmierung erforderlich, die gerade bei Anpassungen der Software zu Problemen führen kann. Besonders bei akkubetriebenen Geräten wie IoT Devices oder Mobiltelefonen wird sehr genau auf die Energieaufnahme geachtet. Ein RTOS Kernels hingegen arbeiten mit einem Event gesteuerten Programmablauf, ein „CPU hogging“ kann somit vermieden werden. Des Weiteren bieten viele RTOS Kernel sehr einfache Lösungen zur effektiven und skalierbaren Nutzung von Energiesparmodis. Dies wird in Abschnitt 2.9 am Beispiel 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 Anwendungen in Tasks entstehen viele kleine Module, die jeweils eine kleine Teilaufgabe des Gesamtsystems übernehmen. Durch ein sauber definiertes In-

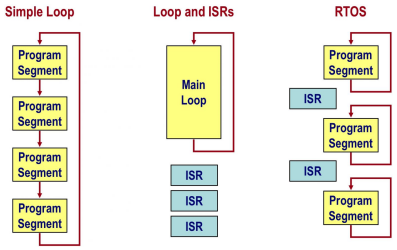

Abbildung 1. Übersicht Programmabläufe in embedded Anwendungen. Quelle [5]

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

terface zur Kommunikation der Tasks lässt sich die Entwicklungsarbeit leicht auf mehrere Teams verteilen. Dies ermöglicht auch den Einsatz von agilen Entwicklungsmethoden wie Scrum in der Entwicklung von eingebetteten Systemen. Ein weiterer großer Vorteil ist die Erweiterbarkeit von RTOS Anwendungen. Bei Änderungen von Anwendungen die in einer Schleife laufen, ist oft der gesamte Code von dieser Änderungen betroffen. Ein RTOS hat durch die Interprozesskommunikation eine natürliche Lose-Kopplung zwischen den einzelnen Programmfunktionalitäten. Das Ändern oder Hinzufügen von Task sind somit wesentlich einfacher, da andere Task nicht unmittelbar durch diese Änderung betroffen sind.

2. FREERTOS

2.1 Geschichte

FreeRTOS wird seit etwa 10 Jahren von der Firma Real Time Engineers Ltd. in Zusammenarbeit mit verschiedenen Chipherstellern entwickelt. Derzeit unterstützt es 35 Architekturen und wurde mehr als 113000 mal heruntergeladen. Das Entwickler Team unter Führung des Gründers Richard Barry, konzentrieren sich bei der Entwicklung darauf sowohl ein geeignetes Qualitätsmanagement umzusetzen, als auch die Verfügbarkeit der verschiedenen Dateiversionen zu gewährleisten. FreeRTOS wird in zwei verschiedenen Lizenzmodellen angeboten, die eine Anpassung der originären GNU General Public Licence darstellen. Die Open Source Lizenz (FreeRTOS) erhält keine Garantien und keinen direkten Support. Entwickler, die diese freie Lizenz verwenden und Änderungen am RTOS Kernel vornehmen müssen den Quellcode ihrer Änderungen für die Community offenlegen. In der kommerziellen Lizenz (SafeRTOS) kann selbst entwickelter Code als closed source vertrieben werden. Ebenso unterstützt Real Time Engineers Ltd. bei der Entwicklung und bietet entsprechende Garantie für die Echtzeitfähigkeit von FreeRTOS. Real Time Engineers bietet zu FreeRTOS diverse Erweiterungen wie Treiber und Tools. Geführt werden diese Erweiterungen unter dem Namen FreeRTOS Ecosystem, dazu gehören unter anderem ein FAT Dateisystem, TCP/UDP Stacks, sowie TLS/SSL Implementierungen.

2.2 Entwicklungsumgebung

FreeRTOS ist im Prinzip nicht an eine spezielle Entwicklungsumgebung gebunden. Dies liegt vor allem daran, dass FreeRTOS in Form von C-Quellcodedateien zur Verfügung gestellt wird und wie eine Art Bibliothek in die zu entwickelnde Software integriert wird. Die verwendete Entwicklungsumgebung muss lediglich einen geeigneten Compiler für das Zielsystem zur Verfügung stellen. Vor dem Start eines Entwicklungsprojektes ist es dennoch ratsam sich einen Überblick über die verfügbaren IDEs[2] zu machen. Der wichtigste Punkt der hierbei zu berücksichtigen 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 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 3.1 aufgezeigt. Ein weiterer Punkt der bei der Auswahl der IDE betrachtet werden muss sind die Kosten. Bei proprietären IDEs können oft mehrere tausend Euro Lizenzkosten anfallen. Diese bieten aber den Vorteil der nahtlosen Einbindungen von μProzessoren und Echtzeitbetriebssystemen (RTOS awareness). Bei der Entwicklung von ARM μProzessoren 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. der maximalen Codegröße. Auf der nicht proprietären Seiten steht Eclipse CDT. Es ist komplett frei in der Verwendung und hat keine Beschränkungen. Der 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 sind die fehlenden Beispielprojekte für Eclipse CDT in der Kombination mit FreeRTOS. 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.

2.3 Zielsysteme STM32F4 (ARM Cortex M3)

32 bit Prozessor - Funktionsübersicht, Hinweis Port Teil von FreeRTOS

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

[2] Integrated Development Environment

[3] Heap3 stellt hier eine Ausnahme dar

Beispiel	Echtzeit Typ	Auswirkung
Tastatur Controller	Soft Realtime	kurzfristig verzögerte Ausgabe
Echtzeit Media Streaming	Soft Realtime	Bild und Ton kurzfristig asynchron
Computer Numerical Control (CNC)	Hard Realtime	Fehler bei der Fertigung des Teils
Airbag System	Hard Realtime	möglicher Personenschaden

Tabelle 1. Beispiele Echzeitsystem

- 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 );
```

ersetzt. Dies hat den Vorteil, dass die Implementierung dieser Funktionen an die jeweilige Anwendung angepasst werden kann. FreeRTOS stellt dem Entwickler fünf unterschiedliche Implementierungen von Speicheralgorithmen (Heap_1.c bis Heap_5.c) zur Verfügung, siehe Abbildung 2. Diese stellen prinzipiell schon die geläufigsten Imple-

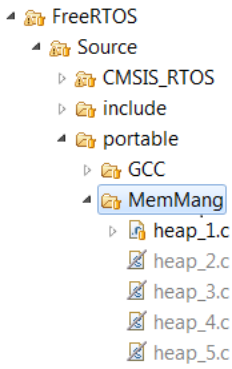

Abbildung 2. Einbindung von Heap1 in Eclipse CDT. Heap2 bis Heap5 sind vom Build ausgeschlossen

mentierungen zur Speicherverwaltung. Es bleibt aber auch weiterhin die Möglichkeit eine eigene Speicherverwaltung zu implementieren. In dieser Arbeit werden wir Heap1 etwas genauer betrachten um ein grundsätzliches Verständnis für die FreeRTOS Speicherverwaltung zu bekommen. Heap2 - Heap 5 werden nur kurz beschrieben und können im Detail in [2] und [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 Erzeugerfunktion xTaskCreate() beispielsweise, erzeugt eine FreeRTOS Task. Listing 3 zeigt wie xTaskCreate() die Funktion pvPortMalloc() verwendet (Zeile 5, 11) um Speicher für den Stack und den

```
1 //**MALLOC**
2 void *pvPortMalloc( size_t xWantedSize )
3 {
4 void *pvReturn = NULL;
5 static uint8_t *pucAlignedHeap = NULL;
6 /* Ensure that blocks are always aligned to
     the required number of bytes. */
7 #if( portBYTE_ALIGNMENT != 1 ){
8  if( xWantedSize & portBYTE_ALIGNMENT_MASK ) {
9   /* Byte alignment required. */
10   xWantedSize += ( portBYTE_ALIGNMENT - (
      xWantedSize & portBYTE_ALIGNMENT_MASK ) );
11  }
12 }
13 #endif
14 vTaskSuspendAll();
15 if( pucAlignedHeap == NULL ){
16  /* Ensure the heap starts on a correctly
     aligned boundary. */
17  pucAlignedHeap = ( uint8_t * ) ( ( (
     portPOINTER_SIZE_TYPE ) &ucHeap[
     portBYTE_ALIGNMENT ] ) & ( ~( (
     portPOINTER_SIZE_TYPE )
     portBYTE_ALIGNMENT_MASK ) ) );
18 }
19 /* Check there is enough room left for the
     allocation. */
20 if( ( ( xNextFreeByte + xWantedSize ) <
     configADJUSTED_HEAP_SIZE ) &&
21  ( ( xNextFreeByte + xWantedSize ) >
     xNextFreeByte ) ) {
22  /* Return the next free byte then increment
     the index past this
23  block. */
24  pvReturn = pucAlignedHeap + xNextFreeByte;
25  xNextFreeByte += xWantedSize;
26 }
27 xTaskResumeAll();
28 return pvReturn;
29 }
```

Listing 1. Implementierung von malloc() in Heap1.c

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 Algorithmen zur Speicherverwaltung

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 (FreeRTOS_config.h) konfiguriert. Die Gesamtgröße berechnet sich wie folgt:

MaxHeapSize = configTOTAL_HEAP_SIZE * Wortbrei-

```
1 //**FREE**
2 void vPortFree( void *pv )
3 {
4  /* Memory cannot be freed using this scheme.
       */
5  ( void ) pv;
6  /* Force an assert as it is invalid to call
       this function. */
7  configASSERT( pv == NULL );
8 }
```

Listing 2. Implementierung von free in Heap1.c

```
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 3. xTaskCreate() memory allocation. Aus Task.c

te[4]

Die Speicherverwaltung durch Heap1 ist sehr einfach. Heap1 deklariert lediglich die Funktion pvPortMalloc(). Die Funktion pvPortFree() wird nicht ausimplementiert. 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 erzeugt werden. Ein so einfacher Speicheralgorithmus wie Heap1 hat durchaus seine Berechtigung. Bei vielen embedded Anwendungen wird der Speicher für die benötigten Objekte vor dem Start des Schedulers erzeugt.

[4]Beim STM32F4 ist die Wortbreite 32 bit

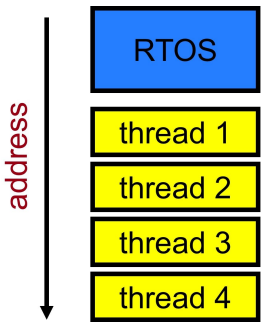

Abbildung 3. Adressraum FreeRTOS und Tasks. Quelle [5]

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

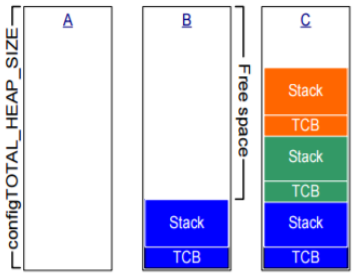

Abbildung 4. Beispiel Speicherbelegung nach drei Instanziierung von Tasks. Quelle [2]

Eine spätere Freigabe von belegten Ressourcen ist nicht nötig, da die Objekte über die gesamte Laufzeit des Programms bestehen sollen. Genau für solche Anwendungen steht Heap1 zur Verfügung. Nachfolgend ein Kurzüberblick über die nicht beschriebenen Speicheralgortihmen.

- Heap2 - Ähnlich Heap1. Erlaubt allerdings Speicherfreigabe durch vPortFree(). 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 mehrere kleinere Speicherblöcke zu einem Großen. Minimiert Speicherfragmentierung.
- Heap5 - Gleicher Algorithmus wie Heap4 allerdings können mehrere Memory Pools erzeugt werden.

2.5.2 Memory Protection

Embedded Softwaresysteme können einen weitere Steigerung der Zuverlässigkeit erreichen durch den Einsatz einer Memory Protection Unit (MPU). Die MPU bietet eine hardwarebasierende Lösung zur Detektion von ungewollten Speicherzugriffen. Für die MPU des STM32F4 uProzessors steht eine spezielle API Portierung von FreeRTOS zur Verfügung (FreeRTOS-MPU). Zur Erzeugung von Task die die MPU nutzen muss die Erzeugerfunktion xTaskCreateRestricted() genutzt werden. Beim Erzeugen der Task wird die Stackadresse der Task mitgeteilt, damit der Kernel die entsprecheden Zugriffsberechtigung der Speicheradressen konfigurieren kann. Die so erzeugten Task werden Ristricted Task genannt. Der Zugriff aus Restricted Task auf den Speicher (Task-Stack) einer anderen Restricted Task ist nicht erlaubt. Bei einem nicht erlaubten Speicherzugriff wird automatisch die entsprechende HookFunktion aufgerufen. Restricted Task können sich in einem der folgenden Modis befinden:

- User Mode
- Priviliged Mode

Im User Mode ist es einer Task nicht erlaubt auf den Speicher des freeRTOS Kernels zuzugreifen, so wird verhindert das der Kernel nicht ungewollt modifiziert wird. Nur einer RestrictedTask die sich im Priviliged Mode befindet

ist ein Zugriff auf den Kernel Speicher erlaubt. Dabei geschieht der Wechsel vom User Mode in den Priviliged Mode implizit durch den Aufruf einer freeRTOS API Funktion. Ein Wechsel durch die Task selbst in den Priviliged Mode ist nicht möglich.

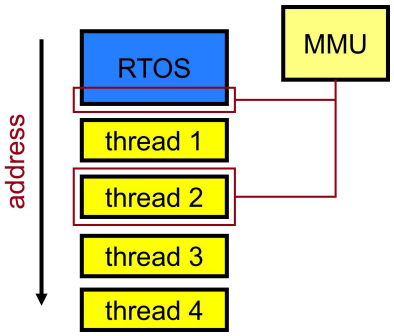

Abbildung 5. Zugriffsrechte für Restricted Task auf gewisse Speicherbereiche wird durch den RTOS Kernel in der MPU konfiguriert. Der Zugriff wir automatisch durch MPU/ MMU überprüft und im Fehlerfall an den Kernel gemeldet. Bild-Quelle [5]

2.5.3 Static Memory Allocation

Die statische Speicherverwaltung wird durch das Präprozessor-Define configSUPPORT_STATIC_ALLOCATION 1 in der FreeRTOS_config aktiviert. Für die statische Objekterzeugung können die dynamischen Erzeugerfunktionen nicht mehr verwendet werden. Daher stehen spezielle Erzeugerfuntkionen für die statische Speicherallozierung zur Verfügung, wie xTaskCreateStatic() statt xTaskCreate() oder xSemaphoreCreateBinaryStatic() statt xSemaphoreCreateBinary(). Der Vorteil der statischen Speicherverwaltung ist, dass der Belegte Speicher im RAM schon zur Übersetzungszeit bekannt ist und die potenzielle Fehlerquelle der dynamischen Speicherverwaltung vermieden wird. Nachteil ist, dass mehr RAM verwendet wird als bei den meisten Heap Implementierungen. Heap1 stellt eine geeignete Alternative in der dynamischen Speicherverwaltung, da es die Risiken der dynamischen Speicherverwaltung auf ein Minimum reduziert.

2.6 Scheduling

Der Scheduler ist die Kernkomponente jedes Echtzeitbetriebssystem Kernels, da er eine quasi parallele Ausführung von Tasks ermöglicht. Eine Task stellt dabei ein eigenständige lauffähige Programmeinheit dar und wird gewöhnlich in einer Schleife ausgeführt. Listing 4 zeigt ein minimal Beispiel einer Task und den Start des Schedulers durch vTaskStartScheduler() in der main function. Folgende Zustände kann eine FreeRTOS Task annehmen:

- Running
- Blocked
- Ready
- Suspended

Abbildung 6 zeigt alle Transitionsübergänge einer FreeRTOS Task. Auf einem uProzessor mit einem Kern kann sich immer nur eine Task im Running Zustand befinden. Alle Tasks im Ready Zustand sind bereit und warten auf ihre Ausführung durch den Scheduler. Tasks die sich im

```
1  void main( void )
2  {
3  //Task werden oft vor dem Start des Schedulers
       erzeugt.
4  xTaskCreate( vTaskCode,
5          "NAME",
6          STACK_SIZE,
7          NULL,
8          tskIDLE_PRIORITY,
9          NULL );
10   // Scheduler wird gestartet
11   vTaskStartScheduler();
12   // Hier sollten wir nicht hinkommen, da der
      Scheduler laeuft.
13 }
14
15 void vTaskCode( void * pvParameters )
16 {
17     for( ;; ){
18         /* Task code wird hier Implementiert
19      z.B. warten auf eine Nachricht*/
20     }
21  /* Hier sollten wir nicht hinkommen*/
22  vTaskDelete( NULL );
23 }
```

Listing 4. Minimal Beispiel für die Definition eine Task.

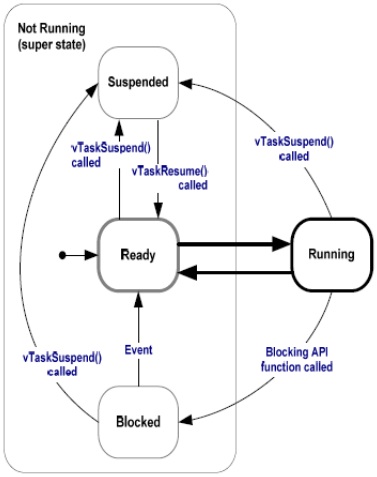

Abbildung 6. Task States. Quelle [2]

Blocked Zustand befinden sind nicht bereit und warten auf ein Synchronisations- oder ein Timer Event. Eine Task die vTaskSuspend() aufruft, wird vom Scheduling Vorgang ausgeschlossen und nimmt den Zustand Suspended an. Erst nach den Aufruf von vTaskResume() verlässt die Task den Suspended Zustand. Welche Task als nächstes vom Zustand Ready in den Zustand Running wechselt, wird durch den Schedulingalgorithmus bestimmt. Der Schedulingalgorithmus des FreeRTOS Kernels basiert auf Round Robin[4]. Alle Tasks gleicher Priorität werden in einer Liste verwaltet. Jede Task in der Liste erhält ein gewisses Zeitquantum, welches bestimmt wie lange einer Task der Prozessor zugeteilt wird. Nach Ablauf des Zeitquantum wird ein Kontextwechsel durchgeführt und die nächste Task in der Liste erhält Prozessorzeit. Die Ausgelaufene Task wird durch den Scheduler automatisch hinten an die Liste angefügt. Da in FreeRTOS jeder Task eine gewisse Priorität zugewiesen wird, gibt es auch für jede Priorität genau eine Liste. Der FreeRTOS Scheduling Algorithmus kann durch Konfigurations Präprozessor-defines angepasst werden. Der Scheduler kann entweder im Cooperative Modus

```
#define taskSELECT_HIGHEST_PRIORITY_TASK(){
 UBaseType_t uxTopPriority = uxTopReadyPriority
    ;
  /* Find the highest priority queue that
     contains ready tasks. */
  while(listLIST_IS_EMPTY(&(pxReadyTasksLists[
     uxTopPriority ]))){
   configASSERT( uxTopPriority );
   --uxTopPriority;
  }
  /* listGET_OWNER_OF_NEXT_ENTRY indexes
     through the list, so the tasks of
     the same priority get an equal share of the
     processor time. */
  listGET_OWNER_OF_NEXT_ENTRY(pxCurrentTCB, &(
     pxReadyTasksLists[uxTopPriority]));
  uxTopReadyPriority = uxTopPriority;
} /* taskSELECT_HIGHEST_PRIORITY_TASK */
```

Listing 5. Pre-emptive List selection aus Task.c

oder im Preemption Modus ausgeführt werden. Die Konfiguration wird durch das define configUSE_PREEMPTION geändert. Im Preemtive Modus wird eine aktive Task mit niedriger Priorität sofort von einer Task mit höherer Priorität verdrängt und ein Kontextwechsel wird durchgeführte. Im kooperativen Modus hingegen wird ein Taskwechsel erst durchgeführt, wenn eine Task den Prozessor explizit abgibt z.B. durch TaskYield(). Abbildung 7 zeigt den Vergleich beider Modis durch einen beispielhaften Ablauf. Eine weitere Option die sich über das define configUSE_TIME_SLI aktivieren lässt ist das sogenannte Zeitschlitzverfahren. Durch das Zeitschlitzverfahren wird die zugeteilte Prozessorzeit für Task gleicher Priorität gleichmäßig aufgeteilt. Dies geschieht durch Einführung fester TickInterrupt Intervalle. Bei jedem TickInterrupt überprüft der Scheduler ob sich eine Task gleicher Priorität im Ready Zustand befindet. Sollte es eine solche Task geben wird ein Kontextwechsel durchgeführt und die Task erhält den Prozessor zugeteilt. Die häufigst verwendete Scheduling Algorithmus nennt sich Prioritized Pre-emptive Scheduling with Time Slicing.

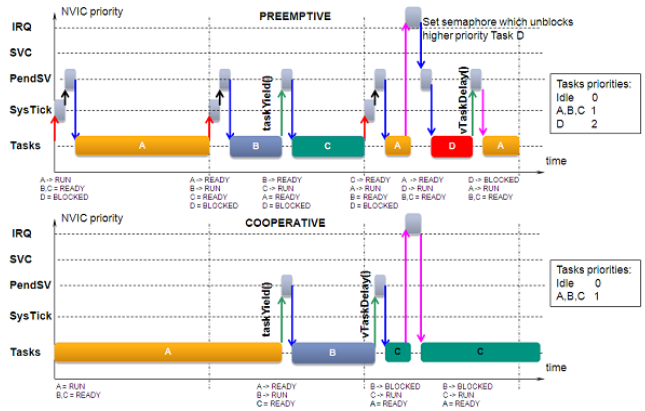

Abbildung 7. Pre-emptive vs. Co-operative. Quelle [2] - Not referenced yet

- Tickcount
- FSM
- IDLE Task
- Priorität nicht durch Scheduler

```
void xPortSysTickHandler( void ){
 /* The SysTick runs at the lowest interrupt
    priority, so when this interrupt
 executes all interrupts must be unmasked.
    There is therefore no need to
 save and then restore the interrupt mask value
    as its value is already
 known. */
 portDISABLE_INTERRUPTS();
 {
  /* Increment the RTOS tick. */
  if( xTaskIncrementTick() != pdFALSE )
  {
   /* A context switch is required. Context
      switching is performed in
    the PendSV interrupt. Pend the PendSV
      interrupt. */
   portNVIC_INT_CTRL_REG =
     portNVIC_PENDSVSET_BIT;
  }
 }
 portENABLE_INTERRUPTS();
}
```

Listing 6. Implementierung von SysTick aus Task.c

```
void vTaskSwitchContext( void )
{
 if( uxSchedulerSuspended != ( UBaseType_t )
    pdFALSE )
 {
  /* The scheduler is currently suspended - do
     not allow a context
   switch. */
  xYieldPending = pdTRUE;
 }
 else
 {
  xYieldPending = pdFALSE;
  /* Check for stack overflow, if configured.
     */
  taskCHECK_FOR_STACK_OVERFLOW();
  /* Select a new task to run using either the
     generic C or port
   optimised asm code. */
  taskSELECT_HIGHEST_PRIORITY_TASK();
  traceTASK_SWITCHED_IN();
 }
}
```

Listing 7. Implementierung von Kontextwechsel aus Task.c

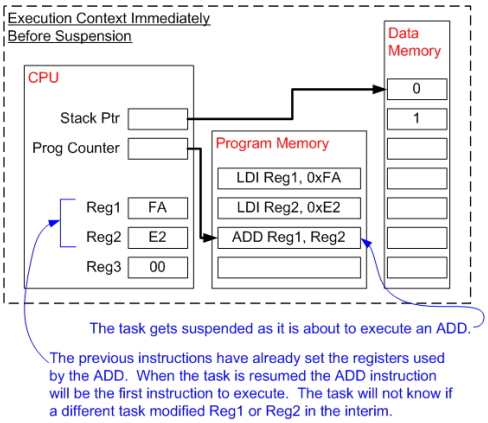

Abbildung 8. FreeRTOS Pseudoimplementierung des Context - Switch. Quelle [2] - Not referenced yet

2.7 Intertask Kommunikation

Queues, Semaphore, Notify, Event Groups

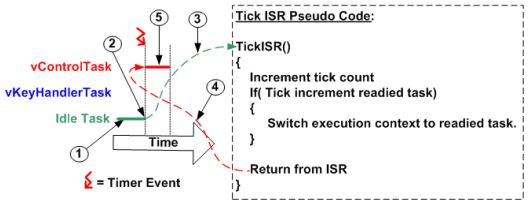

Abbildung 9. FreeRTOS Pseudoimplementierung des Tick Interrupts. Quelle [2] - Not referenced yet

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 10 zeigt wie sich die Stromaufnah-

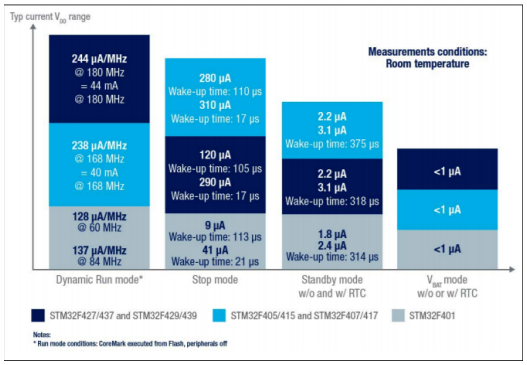

Abbildung 10. Energieaufnahme für STM32F4 in SleepModes
Quelle: STM32F4 - Power Modes

me beim STM32F4 von 40mA im Normalbetrieb(@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

```
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 8. Aufruf der IdleTask Hook Funktion durch die FreeRTOS Idle Task. Aus Task.c

Running wechselt. Die Funktionalität der Idle-Hook Funktion kann frei vom Entwickler implementiert werden. Lis-

```
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 9. Pseudocode für eine Idle Hook Funktion

ting 9 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 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. Der FreeRTOS Kernel 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. Tickless idle kann durch das folgende Define in der FreeRTOSconfig.h aktiviert werden.

```
#define configUSE_TICKLESS_IDLE    1;
```

Im Gegensatz zur gewöhnlichen Idle Hook Funktion berücksichtigt der Tickless Idle Mode alle ausstehenden Timerfunktionen, dazu gehören alle Software Timer aber auch Tasks die nur für eine gewisse Zeit blockiert sind z.B. durch die Funktion xTaskWaitUntil(). Des Weiteren muss der Tickinterrupt nicht wie in Listing 9 explizit abgeschaltet werden, dies wird hier automatisch durch den Kernel gehandelt. Der Tickless Idle Mode bietet durch die Funktion vTaskStepTick() auch die Möglichkeit den TickCount nach dem Aufwachen anzupassen. Die verpassten TickCounts können beispielsweise durch einen externen Timer oder RTC bestimmt werden. So ist es auch möglich Software Timer für absolute Zeiten zu nutzen. Details und Beispiel Implementierungen hierzu findet man unter [1].

3. FREERTOS IN DER PRAXIS

3.1 Komplexität durch Nebenläufigkeit - Debugging von Echtzeitsystemen

Under Construction :D

Durch die Einsatz eines Echtzeitbetriebssystems erhält der Entwickler einige Vorteile die bereits in Abschnitt 1.1 beschrieben wurden. Im Gegenzug entstehen aber durch die Nebenläufigkeit neue mögliche Fehlerquellen. Viele dieser Fehler, lassen sich nicht einfach analysieren und enden beispielsweise im HardFault Handler. Welche Hilfsmittel einem Entwickler speziell bei FreeRTOS (auf STM32F4) zur Verfügung stehen und welche Fehler häufig auftreten ist der Inhalt dieses Abschnitts. Die Arten von Fehler die in einem Echtzeitsystem häufig auftreten lassen sich grob in zwei Kategorien aufteilen:

- Stackoverflow
- Synchronisatiosfehler

Bekannte Probleme detailliert erklärt in [4]

- dining philosopherproblem
- reader and writers problem
- producer consumer problem (starvation)

3.2 Echtzeitanalyse

$$\sum_{i=1}^{m} \frac{C_i}{P_1} \leq 1 \qquad (1)$$

aus [4]

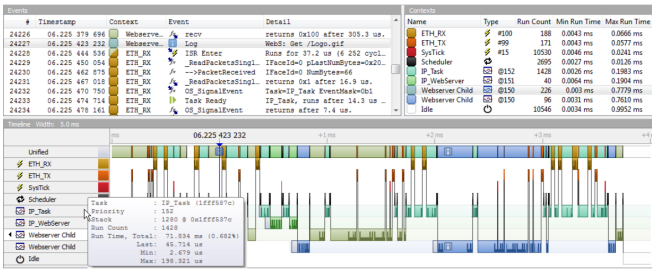

Abbildung 11. Segger Systemview - Not referenced yet

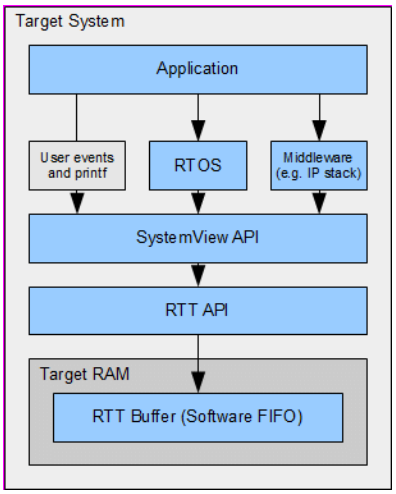

Abbildung 12. Segger Systemview Target - Not referenced yet

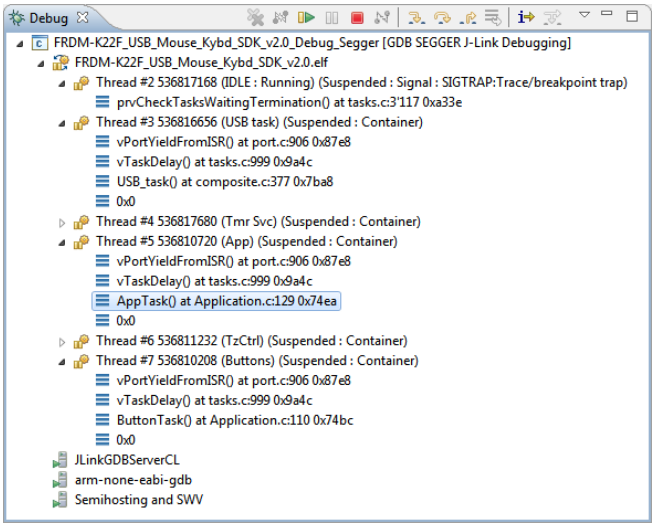

Abbildung 13. Segger Thread Awareness- Not referenced yet

$$\min_{0<1} \leq D_i \left(\sum_{i=1}^{min(i,n_{intr})} \frac{C_j + 2\Delta_{intr}}{t} \left\lceil \frac{t}{T_j} \right\rceil + \sum_{j=n_{intr+1}}^{i} \frac{C_j + 2\Delta_{thr}}{t} \left\lceil \frac{t}{T_j} \right\rceil \right) \leq 1 \qquad (2)$$

aus [3]

4. ZUSAMMENFASSUNG

Literatur

1. R. Barry. Freertos.org implemenation - advanced.
2. R. Barry. *Mastering the FreeRtos Real time Kernel*. Real time Engineers Ltd., pre-release 161204 edition edition, 2016.
3. D. B. Stewart. Measuring execution time and real-time performance. In *Embedded System Conference*, 2006.
4. A. S. Tanenbaum. *Moderne Betriebssysteme*. Pearson Studium.
5. C. Walls. Rtos revealed. *Embedded.com*, 2016-2017.