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. Außerdem wird eine Vorgehensweise für die Einrichtung einer Entwicklungsumgebung vorgestellt.

1. GRUNDLAGEN ECHTZEITSYSTEME

1.1 Echtzeitsysteme und Echtzeitbetriebssysteme

Mit der steigenden Leistungsfähigkeit von modernen μProzessoren steigen auch die Anforderungen an die Software, die auf diesen Systemen aufsetzt. Viele dieser Systeme fordern, trotz ihrer Komplexität, dass Teile des Programmablaufs in bestimmten zeitlichen Grenzen ausgeführt werden und somit vorhersehbar und deterministisch[3] sind. Systeme die solchen Anforderungen unterliegen werden Echtzeitsysteme genannt. Bezogen auf ihre Zuverlässigkeit unterliegen Echtzeitsysteme einer weiteren Unterteilung:

- Echtzeitsysteme mit weicher Echtzeitanforderung (soft realtime systems)
- Echtzeitsysteme mit harter Echtzeitanforderung (hard realtime systems)

Ein weiches Echtzeitsystem soll eine Aufgabe in den vorgegebenen 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 muss die gestellte Aufgabe in den vorgegebenen Grenzen ausführen. Durch eine Überschreitung wird das System unbrauchbar und dies führt in der Folge 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 sind in Tabelle 1 aufgeführt. Um die grundsätzliche Funktionalität eines Echtzeitbetriebssystems zu erläutern, werden zuerst die Grundmodelle für den Programmablauf eingebetteter Systeme beschrieben. Der Programmablauf lässt sich auf drei Modelle zurückführen[7] (Abbildung 1). Eingebettete Anwendungen können in einer endlosen Hauptschleife laufen (mit und ohne Interruptunterbrechungen), in der die

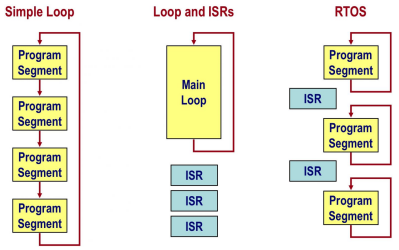

Abbildung 1. Übersicht Programmabläufe in embedded Anwendungen. Unterscheidung von zwei Hauptkategorien: Schleifen-gesteurte Anwendungen und Event-gesteurte Anwendungen. Bild-Quelle [7]

Anweisungen sequenziell abgearbeitet werden oder aber in event-gesteuerten nebenläufigen eigenständigen Programmabschnitten (Thread oder Task[1]). Die nebenläufige Ausführung der unterschiedlichen Programmsegmente ist nur durch einen Scheduler, welcher Teil eines RTOS Kernels ist, zu erreichen. Der RTOS Kernel abstrahiert Timinginformationen[1] und stellt durch den Scheduler sicher, dass der nächste Task rechtzeitig ausgeführt wird. Der Entwickler ist dafür verantwortlich, dass der Task die gewünschte Aufgabe im zeitlichen Rahmen ausführt. Für viele kleine Anwendungen kann die Ausführung in einer endlosen Hauptschleife durchaus sinnvoll sein, wenn beispielsweise die Ressourcen so knapp sind, dass ein Overhead durch zusätzliche Verwaltungsfunktionen ausgeschlossen werden muss. Ein großer Nachteil der „endlosen einschleifigen 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, muss durch den Entwickler sichergestellt werden, dass das Gesamtsystem alle Anweisungen ausgeführt hat und bereit ist schlafen zu gehen. Bei komplexen Anwendungen mit vielen Abhängigkeiten kann es so zu erheblichen Implementierungsaufwand kommen. Besonders bei akkubetriebenen Geräten wie IoT Devices oder Mobiltelefonen wird sehr genau auf die Energieaufnahme geachtet, sodass die Nutzung einer einzigen Hauptschleife hier nicht effektiv genug ist. Ein RTOS Kernel arbeitet mit einem Event gesteuerten Programmablauf, ein „CPU hogging“ kann somit vermieden werden. Des Weiteren bieten viele RTOS Kernel sehr einfache Lösungen zur effektiven Nutzung von Energiesparmodi. Dies wird in Abschnitt 2.9 am Beispiel von FreeRTOS und einem ARM μProzessor demonstriert. Neben der Echtzeitfähigkeit gibt es aber noch viele weitere Vorzüge für den Einsatz eines Echtzeitbetriebssystems. Durch das Herunterbrechen der Anwendungen in Tasks entstehen viele klei-

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

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 von Echtzeitsystemen und deren Auswirkung beim Über- oder Unterschreiten der Anforderungsgrenzen

ne Module, die jeweils eine kleine Teilaufgabe des Gesamtsystems übernehmen. Durch ein sauber definiertes Interface zur Kommunikation zwischen den Tasks, lässt sich die Entwicklungsarbeit gut 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 Tasks ist somit wesentlich einfacher, da andere Tasks 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 Entwicklerteam unter Führung des Gründers Richard Barry konzentriert 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 (OpenRTOS) können solche Änderungen als closed source vertrieben werden. Kunden mit einer kommerziellen Lizenz bietet Real Time Engineers Ltd. Unterstützung bei der Entwicklung von Projekten und Treibern. Des Weiteren werden entsprechende Garantien für die Echtzeitfähigkeit von OpenRTOS gegeben. 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
- TLS/SSL Implementierungen

Abschließend sei auch die Variante SafeRTOS erwähnt, welche durch den TÜV Süd gegen die Richtlinien IEC 61508 SIL 3 vorzertifiziert wurde. Die Erkenntnisse aus dieser Zertifizierung wurden in die Versionen FreeRTOS und OpenRTOS teilweise zurückgeführt.

2.2 Entwicklungsumgebung

FreeRTOS ist grundsätzlich nicht an eine spezielle Entwicklungsumgebung gebunden. Dies liegt vor allem daran, dass FreeRTOS in Form von C-Quellcodedateien zur Verfügung gestellt und vergleichbar einer dynamischen Bibliothek in die zu entwickelnde Software integriert wird. Die verwendete Entwicklungsumgebung muss 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. Ein einfacherer Zugriff auf RTOS Objekte und Eigenschaften ist somit möglich. Das Debugging mittels der RTOS-awareness und die Funktionalitäten, die einem Entwickler bereitstehen, werden 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. Ein Gegenspieler zu den proprietären IDEs ist Eclipse CDT. Es ist komplett frei in der Verwendung und hat keine Beschränkungen. Der Nachteil ist, 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 meist von Grund auf selbst konfiguriert und installiert werden. Da im Laufe dieser Arbeit Eclipse CDT für alle Beispiele verwendet wird, ist in Abschnitt 2.4 das Aufsetzen einer Basiskonfiguration erklärt.

2.3 Zielsystem STM32F4 (ARM Cortex M4)

Der STM32F4 ist ein von STMicroelectronics entwickelter 32 Bit μController, basierend auf einem ARM Cortex M4 Kern. Der STM32F4 arbeitet mit maximal 168 Mhz. Neben seinen unzähligen Schnittstellen (4x UART, SPI, I2C, Ethernet) bietet der STM32F4 mehrere Energiespar-Modi, die ihn für den Einsatz in energieeffizienten Anwendungen, wie IOT Devices interessant machen. Für die Verwendung von FreeRTOS eignet sich der μController be-

[2] Integrated Development Environment

sonders gut, da speziell für diesen μController viele Hardware-Funktionalitäten in den FreeRTOS Kernel integriert wurden. Der STM32F4 ist seit 2012 auf dem Markt und erfährt durch eine große Anzahl an Online-Beispielen eine hohe Beliebtheit in der Entwickler-Community. Für den Zugriff auf μController Funktionen stellt STM den Hardware Abstraction Layer (Abbildung 2), kurz HAL, zur Verfügung.

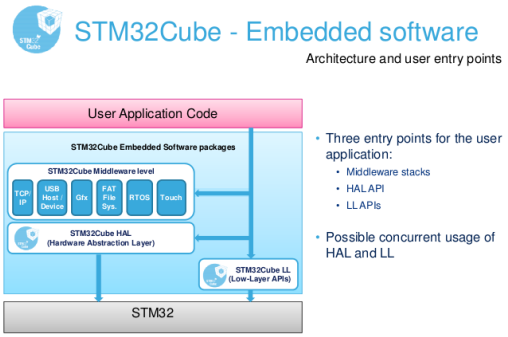

Abbildung 2. Aufbau der zur Verfügung stehenden STM Bibliotheken

Der HAL ermöglicht eine einfache Verwendung der Hardware ohne großen Konfigurationsaufwand. Wie spezielle Hardware-Funktionen des STM32F4 durch FreeRTOS genutzt werden, wird in Abschnitt 2.9 und Abschnitt 2.5.2 gezeigt.

2.4 Einrichten und Konfiguration

Dieser Abschnitt beschreibt die Einrichtung eines FreeRTOS Projektes für den STM32F4. Die Beschreibung dabei ist nicht vollständig und versteht sich eher als eine Art Leitfaden. Alle anwendungsspezifischen Konfigurationen können in den zur Verfügung gestellten Links nachgelesen werden. Hierbei ist besonders die Seite des GNU ARM Plugins hervorzuheben, da diese einen guten Einstieg für die Erstellung eines ARM Projekts bietet. Als Entwicklungsumgebung wird Eclipse CDT verwendet, welches bereits in Abschnitt 2.2 beschrieben wurde. Nach der Installation von Eclipse muss das GNU ARM Plugin für Eclipse CDT installiert werden. Dieses ist entweder über den Pluginmanager oder über den folgenden Link erhältlich:

http://gnuarmeclipse.github.io/

Das Plugin ermöglicht die Einbindung und die Konfiguration von ARM Cross Compilern. Des Weiteren stellt es einige Beispielprojekte für ARM μController zur Verfügung. Nach der Installation des ARM Plugins, müssen die GCC ARM Toolchain und die GNU Build Tools installiert werden. Diese können unter den folgenden Webadressen heruntergeladen werden:

https://launchpad.net/gcc-arm-embedded
https://github.com/gnuarmeclipse/windows-build-tools/

Die Toolchain und die Buildtools stellen Anwendungen bereit, die zum Kompilieren und Debuggen der C und C++ Dateien benötigt werden. Zur Toolchain gehören unter anderem GCC als Cross Compiler und GDB (GNU Debugger) zum Debuggen der Anwendung auf der Zielplattform. Die GNU Buildtools beinhalten make und rm, die zum Organisieren des Builds benötigt werden. Nach der Installation müssen die Verzeichnisse der Toolchain und der Buildtools im Plugin konfiguriert werden. Mit dieser Konfiguration ist das System nun in der Lage C und C++ Dateien für die Zielplattform zu kompilieren und als Binary File (.elf) bereitzustellen. Zum Übertragen und Debuggen der Anwendung auf dem Zielsystem wird ein ISP-Programmer für ARM benötigt.

Folgende ISP-Programmer werden häufig verwendet, dabei ist Liste weder vollständig, noch stellt sie eine Empfehlung dar.

- Segger J-Link:
  https://www.segger.com/jlink-debug-probes.html
- Keil Ulink:
  http://www2.keil.com/mdk5/ulink
- STM ST-Link/VL:
  http://www.st.com/en/development-tools/st-link-v2.html

Zur Nutzung des ISP müssen die benötigten Treiber und der On-Chip Debugger des Herstellers installiert werden. Eine Alternative zum On-Chip Debugger des Herstellers ist OpenOCD. Details dazu findet man unter:

http://openocd.org/

Nachdem die Konfiguration der Entwicklungstools abgeschlossen ist, kann nun ein Basis Projekt erstellt werden. Hierfür empfiehlt sich ein Templateprojekt des GNU ARM Plugins, siehe Abbildung 3. Das Templateprojekt beinhal-

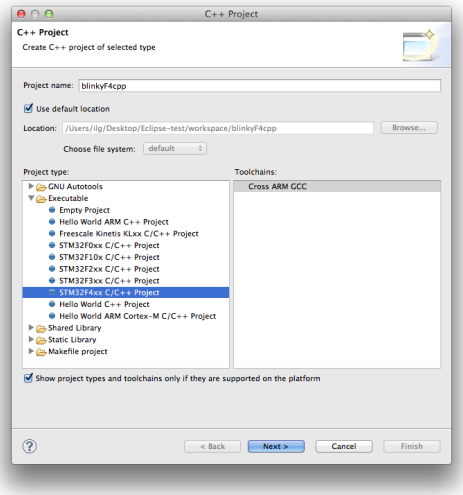

Abbildung 3. Erstellung eines Basisprojekts für den STM32F4 durch das GNU ARM Plugin. Ein Wizzard Konfigurator führt den Anwender durch alle nötigen Einstellungen. Nach Abschluss erhält man ein fertiges C / C++ ARM Projekt welches passend zum Zielsystem konfiguriert ist.

tet bereits alle benötigten Hardware Bibliotheken (Abbildung 4) wie den STM HAL (siehe Abschnitt 2.3) oder das ARM CMSIS (Cortex Microcontroller Software Interface Standard). Das Templateprojekt sollte jetzt kompilieren und mittels ISP-Programmer auf dem Zielsystem ausgeführt werden können. Im nächsten Schritt wird FreeRTOS in das Templateprojekt eingebunden. Hierfür kann auf www.freertos.org die gepackte Variante der Demoprojekte heruntergeladen werden. Für den STM32F4 stehen

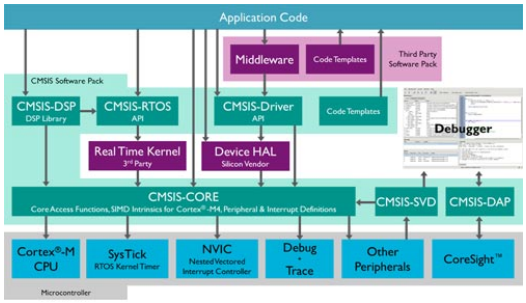

Abbildung 4. ARM embedded Anwendungen setzten auf unterschiedliche Abstraktionsschichten auf. Die untersten beiden Abstraktionsschichten sind CMSIS, welches Funktionen für ARM Devices bereitstellt, und der HAL des μController Herstellers. CMSIS steht für alle ARM Systeme zur Verfügung, wohingegen der HAL μController spezifisch ist.

spezielle Cortex M4 Portierungen zur Verfügung. Die Verzeichnisstruktur sollte dabei der Abbildung 5 ähneln. Nach

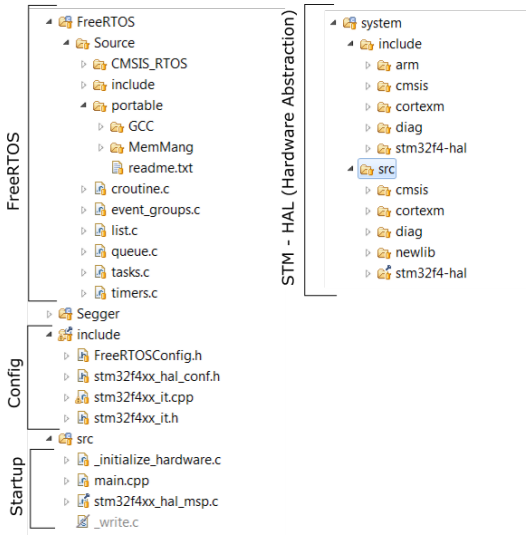

Abbildung 5. Die Basiskonfiguration, die durch das GNU ARM Plugin zur Verfügung gestellt wird, besteht aus den Verzeichnissen: Startup, Config und HAL. Startup beinhaltet alle Files zum Starten der Hardware und die main-Funktion. Im Config Verzeichnis befinden sich alle Dateien zur Konfiguration des μControllers und die FreeRTOS Config. HAL und CMSIS bilden die Grundlage des Systems und werden ebenfalls durch das GNU ARM Plugin eingefügt. FreeRTOS wird danach manuell dem Projekt hinzugefügt.

der Anpassung der Include Pfade für den GCC Compiler und der Konfiguration der FreeRTOS Config ist die Einrichtung abgeschlossen.

2.5 Memory Allocation

Beim Erzeugen von RTOS Objekten wie Tasks, Queues oder Semaphore wird Speicher im RAM benötigt. Für die dynamische Speicherverwaltung werden 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[1] 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 wird am Ende dieses Abschnitts betrachtet. 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 6. Diese stellen prinzipiell schon die geläufigsten Imple-

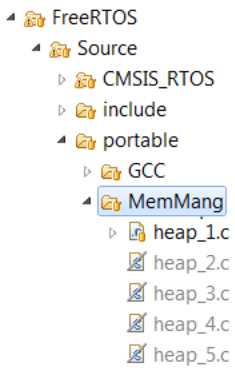

Abbildung 6. Einbindung des Speicheralgorithmus Heap1 in Eclipse CDT. Die Algortihmen Heap2 bis Heap5 sind vom Build ausgeschlossen

mentierungen zur Speicherverwaltung dar. 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 [1] und [2] nachgelesen werden. Wie schon am Anfang dieses Abschnitts beschrieben, wird für alle RTOS Objekte Speicher benötigt. Der Speicher für Objekte wie Semaphore und Tasks wird automatisch in den statischen Erzeugerfunktionen der RTOS API alloziert, indem intern die Funktion *pvPortMalloc()* aufgerufen wird. Die Erzeugerfunktion xTaskCreate() beispielsweise erzeugt eine FreeRTOS Task. Listing 3 zeigt wie xTaskCreate() die Funktion pvPortMalloc() verwendet, 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 7. Durch

[3]Heap3 stellt hier eine Ausnahme dar

```
1 void *pvPortMalloc( size_t xWantedSize )
2 {
3 void *pvReturn = NULL;
4 static uint8_t *pucAlignedHeap = NULL;
5  #if( portBYTE_ALIGNMENT != 1 ){
6   if( xWantedSize & portBYTE_ALIGNMENT_MASK ) {
7    /* Byte alignment required. */
8    xWantedSize += ( portBYTE_ALIGNMENT - (
       xWantedSize & portBYTE_ALIGNMENT_MASK ) );
9   }
10  }
11  #endif
12  vTaskSuspendAll();
13  if( pucAlignedHeap == NULL ){
14   pucAlignedHeap = ( uint8_t * ) ( ( (
       portPOINTER_SIZE_TYPE ) &ucHeap[
       portBYTE_ALIGNMENT ] ) & ( ~( (
       portPOINTER_SIZE_TYPE )
       portBYTE_ALIGNMENT_MASK ) ) );
15  }
16  /* Check there is enough room left for the
       allocation. */
17  if( ( ( xNextFreeByte + xWantedSize ) <
       configADJUSTED_HEAP_SIZE ) &&
18   ( ( xNextFreeByte + xWantedSize ) >
       xNextFreeByte ) ) {
19   pvReturn = pucAlignedHeap + xNextFreeByte;
20   xNextFreeByte += xWantedSize;
21  }
22  xTaskResumeAll();
23  return pvReturn;
24 }
```

Listing 1. FreeRTOS Source von pvPortMalloc() aus Heap1.c. Zuerst wird sichergestellt, dass die Startspeicheradresse dem byte-Alignment des μProzessors entspricht. Der STM32F4 ist ein 32Bit μ-Prozessor und hat ein byte-Alignment von 4, sodass die Startadresse immer eine Potenz von 4 sein muss. Danach wird der Scheduler deaktiviert und geprüft, ob genug Speicher zur Verfügung steht. Abschließend wird der Speicher im ucHeap reserviert.

```
1 void vPortFree( void *pv )
2 {
3  /* Memory cannot be freed using this scheme.
       */
4  ( void ) pv;
5  configASSERT( pv == NULL );
6 }
```

Listing 2. FreeRTOS Source von vPortFree() aus Heap1.c . Da eine Speicherfreigabe in Heap1 nicht vorgesehen ist, ist diese Funktion leer.

den gemeinsamen Adressraum ist es bei fehlerhaften Verwendungen von Speicherzugriffsfunktionen möglich aus einer Task auf die Variablen einer anderen Task zuzugreifen.
In Abschnitt 2.5.2 wird gezeigt welche Möglichkeit der STM32F4 und FreeRTOS bieten um Speicherzugriffe sicherer zu gestalten.

```
1 StackType_t *pxStack;
2 pxStack = ( StackType_t * ) pvPortMalloc(( ( (
       size_t ) usStackDepth )
3 * sizeof( StackType_t ) ) );
4 if( pxStack != NULL )
5 {
6  pxNewTCB = ( TCB_t * ) pvPortMalloc( sizeof(
       TCB_t ) );
7  if( pxNewTCB != NULL )
8  {
9   pxNewTCB->pxStack = pxStack;
10 }
11
12 }
```

Listing 3. FreeRTOS Source von xTaskCreate() aus Task.c. Jede Task besitzt einen Stack und einen Task Control Block, beide werden beim Aufruf von xTaskCreate (Zeile 5 und Zeile 11) erstellt.

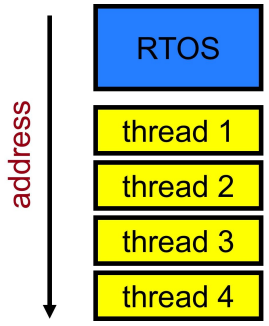

Abbildung 7. Task und Kernel teilen sich in FreeRTOS einen gemeinsamen Adressraum. Dies stellt eine potentielle Fehlerquelle dar. Bild-Quelle [7]

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.

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

Die Größe des Heaps wird durch das Präprozessor-Define configTOTAL_HEAP_SIZE (FreeRTOS_config.h) konfiguriert. Die Gesamtgröße berechnet sich wie folgt:

$$\text{MaxHeapSize} = \text{configTOTAL\_HEAP\_SIZE} * \text{Wortbreite}^4$$

Die Speicherverwaltung durch Heap1 ist sehr einfach. Heap1 deklariert lediglich die Funktion pvPortMalloc(). Die Funktion pvPortFree() wird nicht ausimplementiert. Abbildung 8 zeigt wie 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 und in der Folge allozierter Speicher nicht freigegeben werden kann, 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 Entwickler 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. Eine spätere Freigabe von belegten Ressourcen ist nicht nötig, da die Objekte über

[4] Beim STM32F4 ist die Wortbreite 32 bit

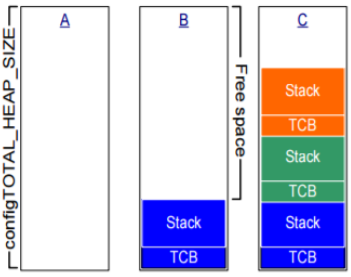

Abbildung 8. Beispiel Speicherbelegung nach drei Instanziierung von Tasks durch die Erzeugerfunktion xTaskCreate() unter Verwendung des Speicheralgorithmus Heap1. Bild-Quelle [1]

die gesamte Laufzeit des Programms bestehen sollen. Genau für solche Anwendungen steht Heap1 zur Verfügung. Nachfolgend ein Kurzüberblick über die nicht beschriebenen Speicheralgorithmen.

- Heap2 - Ähnlicher Algorithmus wie bei 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 - Ähnlicher Algorithmus wie bei Heap1 und Heap2. Verwendet First Fit Algorithmus zur Speicherallozierung. Verbindet mehrere kleinere Speicherblöcke zu einem Großen. Minimiert Speicherfragmentierung.
- Heap5 - Gleicher Algorithmus wie Heap4. Es können mehrere Memory Pools erzeugt werden.

2.5.2 Memory Protection

Embedded Softwaresysteme können durch den Einsatz einer Memory Protection Unit (MPU) eine weitere Steigerung der Zuverlässigkeit erreichen. Die MPU bietet eine hardwarebasierende Lösung zur Detektion von ungewollten Speicherzugriffen (Abbildung 9). Für die MPU

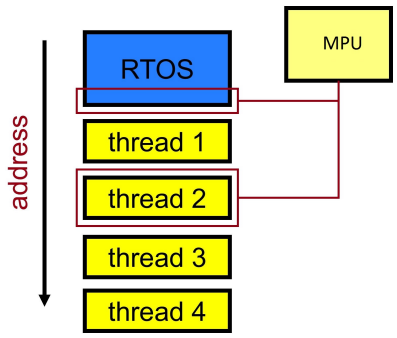

Abbildung 9. Zugriffsrechte für eine Restricted Task werden durch den RTOS Kernel in der MPU konfiguriert. Der Speicherzugriff wird automatisch durch MPU überprüft und im Fehlerfall an den Kernel gemeldet. Bild-Quelle [7]

des STM32F4 μProzessors steht eine spezielle API Portierung von FreeRTOS zur Verfügung (FreeRTOS-MPU). Zur Erzeugung von Tasks, die die MPU nutzen sollen, muss die Erzeugerfunktion xTaskCreateRestricted() verwendet werden. Beim Aufruf der Erzeugerfunktion wird dem Kernel die Stackadresse der Task mitgeteilt, damit dieser die entsprechenden Zugriffsberechtigungen der Speicheradressen konfigurieren kann. Die so erzeugten Tasks werden Ristricted Tasks genannt. Der Zugriff aus einer Restricted Task auf den Speicher (Task-Stack) einer anderen Restricted Task, ist nicht erlaubt. Bei einem nicht erlaubten Speicherzugriff wird automatisch die entsprechende Hook-Funktion aufgerufen. Dem System wird so ermöglicht entsprechend zu reagieren. Restricted Tasks können sich in einem der folgenden Modis befinden:

- User Mode
- Priviliged Mode

Im User Mode ist es einer Restricted Task nicht erlaubt auf den Speicher des FreeRTOS Kernels zuzugreifen. So wird verhindert, dass der Kernel ungewollt modifiziert wird. Nur einer Restricted Task, 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.

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 Erzeugerfunktionen für die statische Speicherallozierung zur Verfügung. Beispiele hierfür sind: 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. Der Nachteil besteht darin, dass mehr RAM verwendet wird, als bei den meisten Heap Implementierungen. Heap1 stellt eine geeignete Alternative in der dynamischen Speicherverwaltung dar, da es die Risiken der dynamischen Speicherverwaltung auf ein Minimum reduziert.

2.6 Scheduling

Der Scheduler ist die Kernkomponente eines Echtzeitbetriebssystem Kernels, da er eine quasi parallele Ausführung von Tasks ermöglicht. Ein Task stellt dabei eine eigenständige lauffähige Programmeinheit dar und wird gewöhnlich in einer endlosen Schleife ausgeführt. Abhängig vom aktuellen Zustand der Tasks und dem gewählten Schedulingalgorithmus wählt der Scheduler den nächsten Task, der ausgeführt werden soll. Auf einem μProzessor mit einem Kern kann dabei immer nur ein Task in einer Zeiteinheit ausgeführt werden. Der Vorgang des Task-Wechsels durch den Scheduler wird Kontextwechsel oder Contextswitch genannt. Der Kontextwechsel beeinflusst nicht die Instruktionsfolge des Tasks. Zum Zeitpunkt der Unterbrechung wird durch den Scheduler eine Art Schnappschuss des Tasks erstellt. Alle Register und der Stack des Tasks werden gesichert. Nachdem der Scheduler den verdrängten Task wieder zur Ausführung ausgewählt hat, werden alle Register und der Stack wiederhergestellt und in die entsprechenden μProzessorregister geladen. Die Task wird danach ab der letzten Instruktion fortgeführt. Abbildung 10 zeigt wie ein Task während seiner Ausführung unterbrochen wird.
Neben den User-Tasks, die durch den Entwickler erstellt

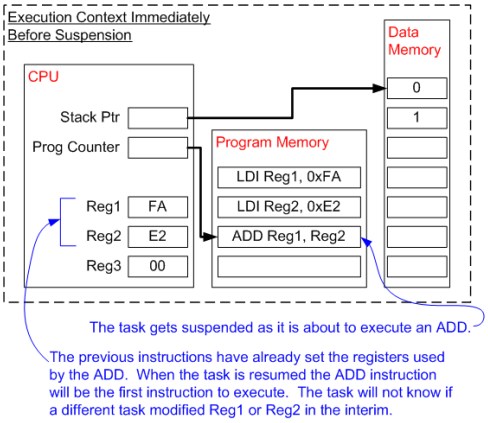

Abbildung 10. Der Kontextwechsel eines Tasks findet mitten in der Ausführung statt. Alle Register, die für die weitere Ausführung benötigt werden, werden durch den Scheduler gesichert. Bild-Quelle [1]

werden, gibt es noch den Idle Task. Dieser wird automatisch beim Start des Schedulers erstellt. Der Idle Task hat immer die niedrigste Priorität (0) und wird immer dann ausgeführt, wenn kein User-Task zur Ausführung bereit steht. Der Idle Task ist ein Indikator für überschüssige Prozessorzeit. Mittels der Idle-Hook Funktion kann dem Idle Task Funktionalität durch den Entwickler hinzugefügt werden. Wie der Idle Task zum Energiesparen genutzt werden kann wird in Abschnitt 2.9 beschrieben. Folgende Zustände kann ein FreeRTOS Task im Scheduling annehmen:

- Running: Der Task wird zur Zeit vom Scheduler ausgeführt
- Blocked: Der Task ist nicht bereit und wartet auf ein Synchronisations- oder ein Timer Event
- Ready: Der Task ist bereit und wartet auf seine Ausführung durch den Scheduler
- Suspended: Der Task hat vTaskSuspend() aufgerufen und wurde vom Schedulingvorgang ausgeschlossen

Abbildung 11 zeigt ein vollständiges Zustandsdiagramm eines FreeRTOS Tasks.

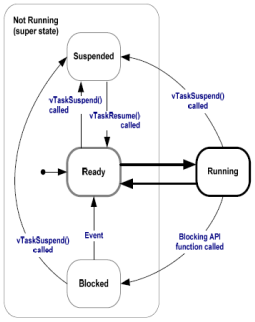

Abbildung 11. Übersicht aller Task-Zustandstransitionen in FreeRTOS. Der Zustandswechsel findet entweder durch den Aufruf einer FreeRTOS API Funktion statt oder aber durch Events z.B. Interrupts, Timer-Events. Der Wechsel in den Zustand Running wird durch den Scheduler bestimmt und ist durch den Schedulingalgorithmus definiert. Bild-Quelle [1]

Die Grundlage aller zur Verfügung stehenden Schedulingalgorithmen ist das Round Robin Verfahren[6]. Dabei werden alle lauffähigen Tasks (Ready) gleicher Priorität in einer Liste verwaltet. Jeder Task in der Liste erhält ein gewisses Zeitquantum[5], welches bestimmt, wie lange einem Task der Prozessor zugeteilt wird. Nach Ablauf des Zeitquantums wird ein Kontextwechsel durchgeführt. Der nächste Task in der Liste erhält Prozessorzeit und der ausgelaufene Task wird durch den Scheduler automatisch hinten an die Liste angefügt. Da jedem Task in FreeRTOS eine gewisse Priorität zugewiesen wird, ist auch für jede Priorität eine eigene Round Robin-Liste nötig. Dieses Verfahren wird auch Priority Scheduling [6] genannt. Abbildung 12 veranschaulicht den Aufbau dieser Listen und in Listing 4 wird gezeigt wie das Priority Scheduling im FreeRTOS Source Code umgesetzt wird. Dem Entwickler

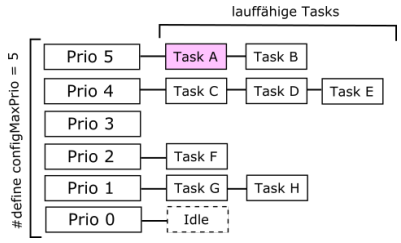

Abbildung 12. Aufbau der Prioritätenliste nach Round Robin in FreeRTOS. Alle aufgeführten Tasks sind bereit zur Ausführung. Task A wird aktuell durch den Scheduler ausgeführt. Nach dem Ablauf des Zeitquantums wird A hinter B einsortiert. Die Maximale Priorität wird durch configMaxPrio bestimmt. Der Idle Task wird automatisch durch den Kernel erzeugt und hat immer die niedrigste Priorität.

```
1#define taskSELECT_HIGHEST_PRIORITY_TASK(){
2 UBaseType_t uxTopPriority = uxTopReadyPriority
    ;
3  /* Find the highest priority queue that
     contains ready tasks. */
4  while(listLIST_IS_EMPTY(&(pxReadyTasksLists[
     uxTopPriority ]))){
5   configASSERT( uxTopPriority );
6   --uxTopPriority;
7  }
8  /* listGET_OWNER_OF_NEXT_ENTRY indexes
     through the list, so the tasks of
9  the same priority get an equal share of the
     processor time. */
10  listGET_OWNER_OF_NEXT_ENTRY(pxCurrentTCB, &(
     pxReadyTasksLists[uxTopPriority]));
11  uxTopReadyPriority = uxTopPriority;
12 } /* taskSELECT_HIGHEST_PRIORITY_TASK */
```

Listing 4. FreeRTOS Source zur Priroty Task Selection aus Task.c. Alle lauffähigen Tasks werden in einem Array verwaltet pxReadyTaskLists. Die Listen verwalten sich durch Referenz-Pointer in den TCBs der einzelnen Tasks

stehen zwei Konfigurationsmöglichkeiten des FreeRTOS Scheduler zur Auswahl: Der Scheduler kann entweder im Cooperative Mode oder im Preemptive Mode ausgeführt werden. Welchen Modus der Scheduler als Schedulingalgorithmus verwendet, wird durch das folgende Define in der FreeRTOS config bestimmt.

```
#define configUSE_PREEMPTION
```

Im Preemptive Mode wird ein aktiver Task mit niedriger Priorität sofort von einem Task mit höherer Priorität verdrängt. Ein Kontextwechsel wird durchgeführt. Im Cooperative Mode hingegen wird ein Kontextwechsel erst durchgeführt, wenn ein Task den Prozessor explizit durch die

[5]Round Robin definiert nicht die Länge des Zeitquantums

Funktion xTaskYield() abgibt. Abbildung 13 zeigt den Vergleich beider Modi durch einen beispielhaften Ablauf. Für

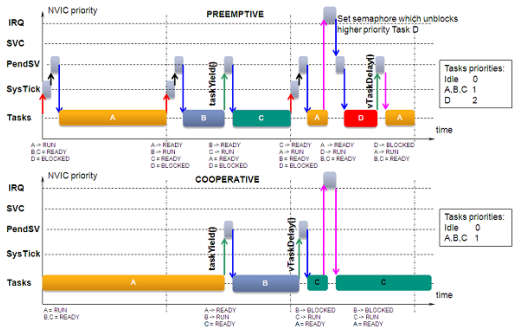

Abbildung 13. Im Cooperative Mode wird der Prozessor von einer Task erst abgegeben, wenn diese explizit taskYield() aufruft. Selbst wenn ein Task mit höherer Priorität in den Ready Zustand wechselt, läuft der Task mit niedrigerer Priorität weiter. Im Gegensatz dazu steht das Pre-Emptive Scheduling (hier mit Time-Slicing). Es unterbricht den laufenden Task mit niedrigerer Priorität sofort, sobald ein Task mit höherer Priorität in den Zustand Ready wechselt. Bild-Quelle [1]

den Preemptive Mode bietet FreeRTOS eine weitere Konfigurationsmöglichkeit. Mit der nachfolgenden Pre-Prozessordire lässt sich ein Zeitschlitzverfahren (time slicing) aktivieren.

```
#define configUSE_TIME_SLICING 1
```

Durch das Zeitschlitzverfahren wird die zugeteilte Prozessorzeit für Tasks gleicher Priorität gleichmäßig aufgeteilt. Dies geschieht durch Einführung eines festen Tick-Interrupt Intervalls. Bei jedem Tick Interrupt wird der FreeRTOS SysTickHandler aufgerufen. Listing 5 zeigt die Implementierung des FreeRTOS SysTicks. Der SysTickHandler ist Bestandteil des Schedulers. Er überprüft bei jeder Ausführung, ob sich ein Task gleicher Priorität im Ready Zustand befindet. Sollte es einen solchen Task geben, wird ein Kontextwechsel durchgeführt und der Task erhält den Prozessor zugeteilt. Des Weiteren kümmert sich der SysTickHandler um die Verwaltung des TickCount, welcher als Referenz für alle RTOS Timingfunktionen dient. Abbildung 14 zeigt diesen Vorgang nochmal im zeitlichen Verlauf.

```
1 void xPortSysTickHandler( void ){
2  portDISABLE_INTERRUPTS();
3  {
4   /* Increment the RTOS tick. */
5   if( xTaskIncrementTick() != pdFALSE )
6   {
7    /* A context switch is required. */
8    portNVIC_INT_CTRL_REG =
       portNVIC_PENDSVSET_BIT;
9   }
10  }
11  portENABLE_INTERRUPTS();
12 }
```

Listing 5. FreeRTOS Source des SysTickHandlers aus Task.c. Der SysTickHandler verwaltet den TickCount. Der TickCount dient allen Timingfunktionen des RTOS Kernels als Zeitreferenz. Des Weiteren wird beim aktiven Time Slicing überprüft ob ein Kontextwechsel nötig ist. Der Kontextwechsel wird dann ggf. durch den PendSVHandler durchgeführt.

Die wohl am häufigsten verwendete Konfiguration ist der Preemptive Mode mit aktivem Zeitschlitzverfahren.

```
#define configUSE_PREEMPTION 1
#define configUSE_TIME_SLICING 1
```

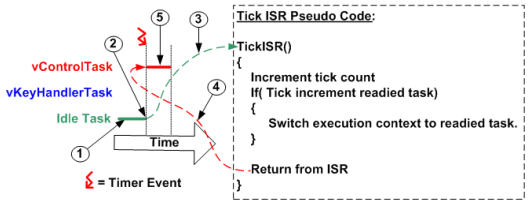

Abbildung 14. Beispielhafter Ablauf eines SysTickInterrupts.(1) kein User Task ist ready, der Idle Task ist aktiv. (2) SysTickInterrupt. (3) SysTickHandler wird aufgerufen. (4) vControlTask ist ready und ein Kontextwechsel wird durchgeführt. vControlTask hat hier die gleiche Priorität wie der IdleTask. (5)vControlTask wird ausgeführt. Bild-Quelle [1]

Diese Einstellung wird üblicherweise Prioritized Pre-emptive Scheduling with Time Slicing genannt. Abbildung 15 zeigt wie sich diese Konfiguration des Schedulers bei mehreren Tasks mit unterschiedlicher Priorität verhält. FreeRTOS ist

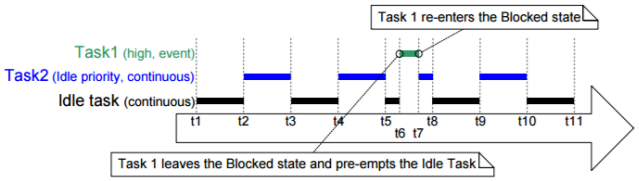

Abbildung 15. Durch das Zeitschlitzverfahren wechseln sich Task1 und Idle Task bei jedem SysTick Interrupt ab, da beide die gleiche Priorität haben. Bei T6 ist Task 1 bereit und verdrängt (preempt) aufgrund seiner höheren Priorität Task2. Nachdem Task 1 blockiert, wird Task 2 fortgeführt. Bild-Quelle [1]

grundsätzlich für weiche Echtzeitanforderungen geeignet, dadurch dass die Auswahl der nächsten Task durch die gegeben Scheduling Algorithmen garantiert wird und somit vorhersehbar ist. Bei harten Echtzeitanforderungen kann man das nicht so pauschal sagen, da es hier auf weitere Faktoren ankommt. Wichtig ist hier beispielsweise die Periode des zugrundeliegendem SysTick Interrupt, der für alle Timings des RTOS Kernels verantwortlich ist. Dieser ist wieder Abhängig von der eingesetzten Hardware. Ein weiterer wichtiger Punkt ist, dass für FreeRTOS keine worst cast Timing Informationen für den Scheduler publiziert werden. Bei weichen Echtzeitsystemen können diese Timings oft vernachlässigt werden, bei harten Echtzeitsystemen müssen diese allerdings bekannt sein, damit diese mit in die Schedulinganalyse mit einbezogen werden können. Um harten Echtzeitanforderungen in FreeRTOS gerecht zu werden, müssten diverse Messungen vorgenommen werden, um die Echtzeit-Performance zu bestimmen, dies wird beispielsweise in [5] gezeigt.

2.7 Intertask Kommunikation

In Projekten, in denen verschiedene Tasks parallel verarbeitet werden, ist es häufig erforderlich, dass diese Tasks die Möglichkeit besitzen Informationen untereinander auszutauschen. Zum Einen kann ein Task Informationen produzieren, die ein anderer Task für die weitere Verarbeitung benötigt. Zum Anderen können beide Tasks gemeinsame Ressourcen (bspw. Hardwareregister oder Variablen) nutzen. Hierbei muss sichergestellt werden, dass die dort hinterlegten Daten jederzeit korrekt sind. FreeRTOS bietet hierfür verschiedene Funktionen zur Interprozesskommunikation an. Zuerst sind hier die Queues zu nennen. Diese dienen dem klassischen Austausch von Informationen, in-

dem Daten durch einen Task in die Queue hineingeschrieben werden und von einem zweiten Task gelesen werden. Die Größe der Queue wird beim Aufruf der Erzeugerfunktion definiert. In FreeRTOS wird eine Queue mit folgender Funktion erstellt.

```
xQueueCreate(uxQueueLength, uxItemSize)
```

Es werden keine expliziten Queues angeboten, die Pointer speichern. Es ist jedoch möglich Pointer als Datum in einer Queue zu hinterlegen. Um einen Überlauf der Queue zu verhindern, werden Tasks, die in eine volle Queue hineinschreiben wollen, in den Zustand Blocked versetzt. Ebenso werden Tasks behandelt, die versuchen Daten aus einer leeren Queue abzurufen. Um die Echtzeitfähigkeit der Tasks weiter zu gewährleisten, bieten die Funktionen xQueueSendToFront(), xQueueSendToBack() und xQueueReceive() einen Parameter xTicksToWait an. Mit xTicksToWait übergibt ein Task den Zeitwert, in dem der Task eine Antwort erwartet. Erfolgt innerhalb dieser Zeit keine Antwort, so wird der Task mittels Rückgabewert über diesen Timeout informiert. Wenn mehrere Tasks auf eine gemeinsame Queue zugreifen, wird deren Zugriff im o.g. Fall zuerst nach Priorisierung des Tasks und danach durch Wartezeit gesteuert. Je nach Zustand der Queue erhält der Task mit der höchsten Priorität den Zugriff. Existieren zwei Tasks mit der gleichen Priorität, so darf der Task zugreifen, der schon länger auf einen Zugriff wartet.
Neben Queues werden von FreeRTOS Mailboxen angeboten. Diese verhalten sich grundsätzlich wie Queues. Jedoch beinhalten sie nur ein Datenobjekt. Dieses wird nach dem Lesen nicht direkt gelöscht, sondern verbleibt in der Mailbox, bis es von einem datenerzeugenden Task überschrieben wird. Mailboxen sind vor allem in Szenarien interessant, in denen mehrere Tasks lesend auf ein erzeugtes Datum zugreifen sollen. Beispielsweise greift ein Task zur Verarbeitung und ein niedriger priorisierter Task zur Anzeige auf die Mailbox zu.
FreeRTOS bietet außerdem Semaphore an. Üblicherweise werden diese für die Behandlung von Interrupts verwendet. Semaphore werden im Rahmen der Interprozesskommunikation meist zur Synchronisierung der Tasks angewendet. Hierzu wird durch einen Task der Semaphor angefordert, durch den zweiten Task der Semaphor vergleichbar einem Interrupt gesetzt. Semaphore werden detailliert im Abschnitt 2.8 beschrieben.
Die nächste Gruppe der Funktionen zur Interprozesskommunikation sind die Mutexe. Diese bilden eine Sonderform der Semaphore. Mutexe werden benutzt um Zugriffe auf gemeinsam genutzte Ressourcen zu steuern. Wenn ein Task auf eine Ressource zugreifen will, so prüft er vorher, ob er den Mutex erhalten kann. Ist dies nicht der Fall (weil ein anderer Task den Mutex besitzt), so muss der Task warten bis der andere Task den Mutex zurück gibt. Zur Unterstützung von rekursiven Funktionen bietet FreeRTOS rekursive Mutexe an, die von einer Task mehrfach angefordert werden können. Im Rahmen von Echtzeitsystemen müssen jedoch zwei Risiken beim Einsatz von Mutexen berücksichtigt werden. Es besteht ein Risiko darin, dass Deadlocks entstehen. Hierbei versuchen zwei (oder mehr) Tasks zeitgleich auf zwei (oder mehr) Ressourcen zu zugreifen. Beiden Tasks gelingt es mindestens einen Mutex zu erhalten. In der Folge kann keiner der Tasks vollen Zugriff auf die Ressourcen erlangen und wartet jeweils auf den anderen. Der Deadlock kann nur aufgebrochen werden, indem einer der Tasks seine Mutexe zurück gibt und der andere diese erhalten kann. Durch die notwendigen Timeouts kann die Echtzeitfähigkeit des Systems sichergestellt werden. Das andere Risiko besteht darin, dass eine niedrig priorisierte Task einen Mutex erhält und damit eine höher priorisierte Task von der Verwendung der Ressource ausschließt. FreeRTOS bietet hierzu eine Möglichkeit an, die niedrig priorisierte Task kurzzeitig auf die Priorität der hoch priorisierten Task zu setzen, so dass hier eine zeitnahe Abarbeitung stattfinden kann. Es kann jedoch auch hier zu Laufzeitproblemen kommen. Die Entwickler von FreeRTOS verwenden daher Wrapper-Funktionen, auch Gatekeeper genannt. Diese nehmen eine Kapselung der Ressourcen vor und werden über Queues angesprochen. Auf diesem Weg kann auf den Einsatz von Mutexen weitestgehend verzichtet werden. Die dritte und letzte Form der Intertaskkommunikation, die von FreeRTOS angeboten wird, ist die Task Notification. Sie ist die Variante mit dem geringsten Ressourcenaufwand, da anders als bei Queues und Mutexes keine Datenobjekte angelegt werden müssen. Durch das Aktivieren der Task Notifikation innerhalb der FreeRTOSConfig.h (Setzen von configUSE_TASK_NOTIFICATIONS auf 1) wird pro Task ein fester Speicherbereich reserviert, der für die Notification genutzt wird. Task Notifications werden direkt an die Zieltask gesendet. Es findet, anders als bei Queues, kein Zwischenspeichern der Information statt. Wenn ein Task einen anderen Task benachrichtigt, wird er in den Zustand blockiert versetzt, bis der Notification Wert in den hierfür vorgesehen Speicherbereich geschrieben wurde. Darüber hinaus ist bei Notifications sichergestellt, dass die Informationen ausschließlich zwischen den beiden beteiligten Tasks ausgetauscht werden. Ein Zugriff eines dritten Tasks oder einer ISR (Interrupt Service Routine) auf diese Kommunikation ist ausgeschlossen.

2.8 Interrupt Handling

Interrupts können innerhalb von FreeRTOS auf verschiedenen Wegen behandelt werden. Hierbei bilden die Hardware gesteuerte Interrupt Service Routinen (ISR) die Basis. Um die Verarbeitungszeit für einen Interrupt kurz zu halten, führen ISRs gewöhnlich nur wenige Instruktionen aus. Dies geschieht beispielsweise durch das Informieren einer FreeRTOS Task mittels Intertaskkommunikation. Die FreeRTOS Task führt danach die eigentliche Aufgabe aus. Da die normalen API Funktionen für den Aufruf aus einer Task implementiert wurden und spezielle Eigenschaften einer Task verwenden, kann eine normale API Funktion nicht in einer ISR verwendet werden. Beispielsweise versetzen viele Intertask API Funktionen die aufrufende Task in den Blocked Zustand. Dies ist im ISR Kontext natürlich nicht möglich. Damit man diese Funktionen dennoch nutzen kann, stellt FreeRTOS für die meisten API Funktionen, spezielle ISR API Funktionen zur Verfügung. Diese Funktionen haben den postfix FromISR. ISR API Funktionen deaktivieren kurzfristig die Interruptverarbeitung innerhalb der kritischen Zugriffe. FreeRTOS bietet verschiedene Möglichkeiten an, um Tasks einen Zugriff auf Interrupts zu ermöglichen.
Zuerst die binären Semaphore, die mit xSemaphoreCreateBinary(void) erstellt werden. Hierbei handelt es sich um Speichervariablen, die einen binären Wert annehmen können. In dem Moment, in dem die Variable den Wert TRUE annimmt, ändert die Task ihren Zustand von Blocked auf Ready. Die Task kann in den Wartezustand gebracht werden, indem xSemaphoreTake() aufgerufen wird. Der Se-

maphor selbst wird durch eine ISR gesetzt. Binäre Semaphore werden meist zu Synchronisationszwecken zwischen einem Task und einem Interrupt eingesetzt. Da nicht sichergestellt ist, dass die Task innerhalb der Zeitspanne, in der ein weiterer Interrupt auftreten kann, den vorhandenen Interrupt verarbeiteten kann, ist es möglich, dass ein Interrupt bei binären Semaphoren "verloren'"geht.
Abhilfe schaffen hier die Counting Semaphore, die die nächste Zugriffsvariante auf Interrupts darstellen. Die Counting Semaphore werden durch das Setzen der folgenden Pre-Prozessor Direktive in der FreeRTOS Config aktiviert.

```
#define configUSE_COUNTING_SEMAPHORES 1
```

Im Anschluss kann die Semaphore mittels

```
xSemaphoreCreateCounting(uxMaxCount,
    uxInitialCount)
```

angelegt werden. Die Counting Semaphore werden hierbei als Queue angelegt, die jedoch eher wie ein Zähler funktionieren. Der Parameter uxMaxCount legt hierbei fest, ab welchem Wert ein Überlauf des Zählers erfolgt. uxInitialCount legt den Wert des Zählers nach der Initialisierung fest. Im Anschluss können die Counting Semaphore wie binäre Semaphore verwendet werden. Der Aufruf von xSemaphoreTake() holt hierbei ein Semaphor-Objekt aus der Queue und versetzt den Task erst in den Blocked Zustand, wenn die Queue leer ist. Eine Möglichkeit um ganze Gruppen von Interrupts zusammenzufassen sind die Eventgroups. Hierbei geschieht die Tasksteuerung über Bitmasken. Innerhalb eines Tasks kann eine „unblock condition" definiert werden. Mit dieser kann definiert werden, ob der Task nur bei einer vollständig identischen Maske in den Zustand Ready wechselt, oder ob es bereits ausreicht, dass ein einzelnes Bit in der Maske gesetzt wird. Die Bedeutung der einzelnen Bits kann durch die Entwickler frei festgelegt werden. Eine Eventgroup wird mit dem Kommando xEventGroupCreate(void) erzeugt. Mittels xEventGroupSetBits(EventGroup,uxBitsToSet) werden die Bits uxBitsToSet innerhalb der Eventgroup gesetzt. Diese Funktion kann auch innerhalb von Tasks aufgerufen werden, beispielsweise zum Zwecke der Tasksynchronisation. Ein Task kann sich in den Blocked Zustand versetzen, indem xEventDroupWaitForBits() aufgerufen wird. Diese Funktion erhält außerdem als Parameter die Eventgroup, sowie die Bits die beobachtet werden. Darüber hinaus wird mittels weiterer Parameter festgelegt, ob die aktuell gesetzten Bits zurückgesetzt werden sollen.

2.9 Low Power Modes auf Stm32F4

Echtzeitbetriebssysteme werden immer häufiger in akkubetriebenen embedded Systemen eingesetzt. Solche Systeme verlangen eine effiziente Nutzung der Energieressourcen, um einen möglichst langen Betrieb zu gewährleisten. Bezogen auf den μProzessor gibt es zwei Wege zur Energieeinsparung:

- Heruntertakten des μProzessors.
- Das System schlafen legen, wenn keine weiteren Aufgaben anstehen.

Das Heruntertakten des μProzessors ist unabhängig vom Einsatz eines RTOS. Aus diesem Grund wird hier nur der zweite Punkt genauer betrachtet, das Schlafenlegen des μ-Prozessors. Abbildung 16 zeigt wie sich die Stromaufnahme beim STM32F4 von 40mA im Normalbetrieb(bei 168

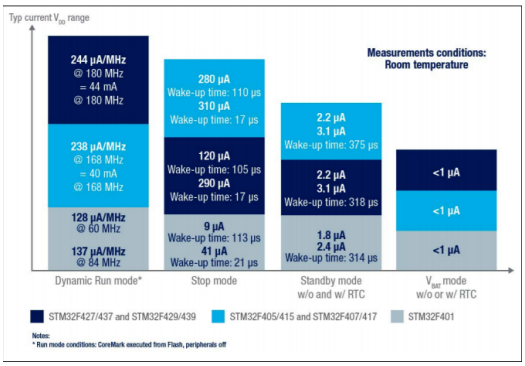

Abbildung 16. Der STM32F4 bietet diverse LowPower Modes. Die Modes haben starke Auswirkung auf die Funktionalität des uControllers während der Schlafphase. Beispielsweise kann im Stop Mode keine UART Schnittstelle benutzt werden. Abhängig von der benötigten Peripherie, wählt der Entwickler einen dieser Modes. Die genutzte Taktfrequenz hat ebenfalls Einfluss auf die Stromaufnahme. Eine Anpassung der Takfrequenz zur Laufzeit ist ebenfalls möglich.

Quelle: STM32F4 - Power Modes

MHz) auf 2,2μA im Tiefschlafmodus reduzieren lässt. In einfachen Anwendungen ist der Zustand, in dem ein Gerät schlafen gehen kann, relativ leicht zu ermitteln.
In komplexen Systemen ist die Ermittlung dieses Zustands aufwendiger, da mehrere Tasks möglicherweise auf unterschiedliche Ressourcen warten. Ein Echtzeitbetriebssystem, wie FreeRTOS, kann hierbei unterstützen. 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 der IDLE Task von FreeRTOS aktiviert, sobald sich alle User-Tasks im Blocked Zustand befinden. Durch konfigurieren des Präprozessor-Defines

```
#define configUSE_IDLE_HOOK 1;
```

```
1 static portTASK_FUNCTION( prvIdleTask,
      pvParameters )
2 {
3  /* Stop warnings. */
4  ( void ) pvParameters;
5  /** THIS IS THE RTOS IDLE TASK - WHICH IS
       CREATED AUTOMATICALLY WHEN THE
6  SCHEDULER IS STARTED. **/
7  for( ;; ) {
8 //skipped some code
9 if ( configUSE_IDLE_HOOK == 1 )
10 {
11  extern void vApplicationIdleHook( void );
12  vApplicationIdleHook();
13 }
14 //guess what.. skipped more code
15 }
```

Listing 6. Aufruf der Idle-Hook Funktion durch den FreeRTOS Idle Task. Aus Task.c

kann die Idle-Hook Funktion aktiviert werden. Diese wird immer aufgerufen, sobald der Idle Task in den Zustand Running wechselt. Die Funktionalität der Idle-Hook Funktion kann frei vom Entwickler implementiert werden.
Listing 7 zeigt Pseudocode zu einer beispielhaften Implementierung der Idle-Hook Funktion. Bevor das System schlafen gelegt werden kann, müssen alle GPIOs und IRQs konfiguriert werden, sodass das System nicht unnötigerweise aufwacht. Des Weiteren werden alle nicht benötigten GPIOs auf Analog gestellt um Energie zu sparen. Als einzige

```
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 7. Pseudocode für eine Idle-Hook Funktion

Interrupt-Quelle wird hier eine externe RTC konfiguriert. Mit dem Aufruf von HAL_PWR_EnterSTOPMode() wird der μProzessor in den Schlafmodus versetzt. Die Funktion wird erst wieder verlassen, wenn der externe Interrupt der RTC ausgelöst wurde. Danach werden alle GPIOs auf den ursprünglichen Zustand zurückgesetzt. Auserdem sind die User-Tasks zu informieren, z.B. mittels Notify oder Message, damit das System nicht beim nächsten Tick Interrupt den Idle Task reaktiviert. Nachteil dieser Variante ist, dass die Nutzung von Software Timern nicht mehr möglich ist. Der FreeRTOS Kernel würde in diesem Fall die Idle-Hook Funktion aufrufen und sich schlafen legen, obwohl 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 stellt, der sogenannte Tickless Idle Mode. Der Tickless Idle Mode 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 neben allen Software Timern 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 7 dargestellt, 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 die RTC bestimmt werden. So ist es auch möglich Software Timer für absolute Zeiten zu nutzen. Details und Beispiel Implementierungen hierzu finden sich unter [2].

3. FREERTOS IN DER PRAXIS

3.1 Komplexität durch Nebenläufigkeit - Debugging von Echtzeitsystemen

Durch den 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 oft im HardFault Handler. Der HardFault Handler ist dabei eine Art Endstation und wird aufgerufen, sobald der μProzessor feststellt, dass eine schwerwiegende, fehlerhafte Operation stattgefunden hat. Des Weiteren können natürlich auch gewöhnliche Synchronisationsprobleme auftreten, wie Starvation oder Deadlocks. Dieser Abschnitt soll dabei nicht erklären wie solche Probleme im Detail gelöst werden können, dies wir bereits hinlänglich in der Literatur behandelt z.B. in [6] [4]. Dieser Abschnitt soll zeigen, welche Tools und Möglichkeiten ein Entwickler hat, um solche Probleme ausfindig zum machen. Beim Debuggen einer Anwendung kann der Entwickler gewöhnlich durch die Nutzung eines ISPs auf die aktuellen Registerinhalte und den Stack des μProzessors zugreifen. Das Problem ist, dass eine FreeRTOS Anwendung gewöhnlich aus mehreren Task besteht und jede Task eine eigene Anwendungseinheit darstellt. Der Stack und die Register einer verdrängten Task werden durch das Echtzeitbetriebssystem gesichert und sind für den ISP nicht mehr sichtbar. Daher bieten einige ISP Hersteller spezielle Thread Aware Pakete. Diese ermöglichen das Auslesen von Daten einer blockierten Task, siehe Abbildung 17. Ein weiteres,

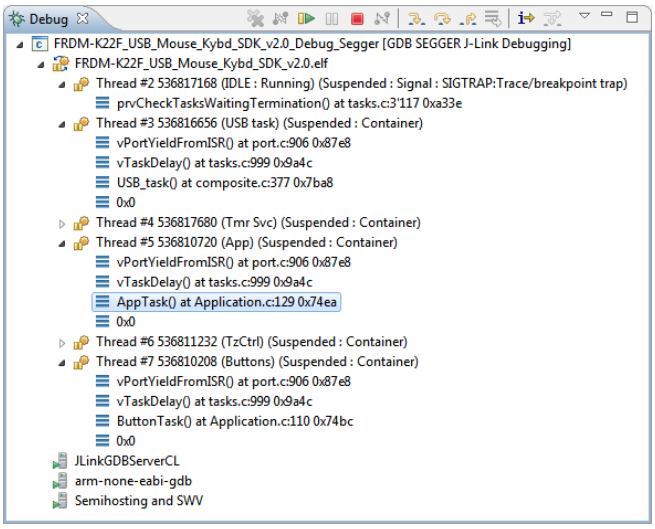

Abbildung 17. In diesem Beispiel ist die IDLE Task running, alle anderen Task blockieren (Hier wird für den Zustand Blocked die Bezeichnung Suspended verwendet). Es kann jedoch durch die Thread Awareness auf den Stacktrace der anderen Task zugegriffen werden

sehr mächtiges Tool zur Analyse von Anwendungen, die auf einem Echtzeitbetriebssystem aufsetzen, sind die sogenannten Trace Tools. Diese ermöglichen die Aufnahme der Scheduling Vorgänge zur Programmlaufzeit, wie in Abbildung 18 dargestellt. Besonders in Anwendungen, in den viele Tasks interagieren, ist der Einsatz eines solchen Tools fast unabdingbar. Viele dieser Trace Tools ermöglichen eine ununterbrochene Aufzeichnung des RTOS Kernels. Besonders Fehler, die erst nach sehr langer Programmlaufzeit auftreten oder aber nur sporadisch stattfinden, können so entdeckt und analysiert werden. Für die Nutzung von Trace Tools werden weitere Bibliotheken benötigt, die eine weitere Schicht zwischen Hardware und Echtzeitbetriebssystem bilden, siehe Abbildung 19. Eine noch bessere Möglichkeit zum Aufzeichnen der Vorgänge auf dem embedded System, sind die sogenannten Trace Recorder. Dabei handelt es sich um ISP Programmer mit integrier-

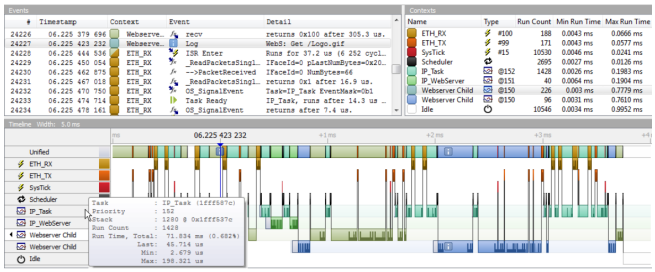

Abbildung 18. Trace Tool Segger Systemview ermöglicht die Aufnahme aller Schedulingvorgänge und stellt diese im zeitlichen Verlauf dar. Dem Entwickler ist es somit möglich alle RTOS Operationen rückblickend zu betrachten.

tem Trace Buffer (z.B. Segger J-Trace). Mit einem Trace Recorder kann der gesamte Programmablauf aufgenommen werden. Der Trace Rekorder nimmt nicht nur FreeRTOS spezifische Abläufe auf, sondern alle Instruktionen des μProzessors. Dadurch ist es möglich rückwärts durch den Instruktionsverlauf zu springen. Man kann die Anwendung also quasi zurückspulen.

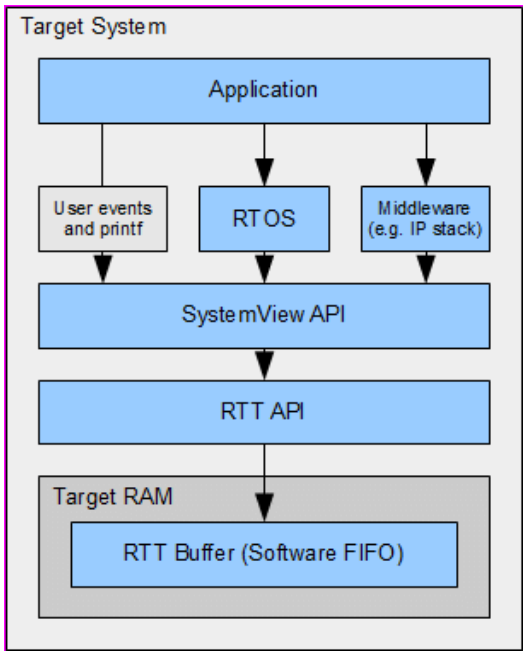

Abbildung 19. Die benötigten Target Files für die Trace Tools bilden eine weitere Middleware Schicht.

4. ZUSAMMENFASSUNG

FreeRTOS kann als freies, professionelles Echtzeitbetriebssystem betrachtet werden. Im Vergleich zu kommerziellen Echtzeitbetriebssystemen wird ein annähernd gleicher Funktionsumfang gewährleistet. Bei den Herstellern von uControllern und ISP ist FreeRTOS eines der Standard Echtzeitbetriebssysteme. Es stehen gewöhnlich viele Beispiele oder Template Projekte für FreeRTOS zur Verfügung. Besonders für Einsteiger ist FreeRTOS sehr zu empfehlen, da es kostenlos und sehr gut dokumentiert ist. FreeRTOS wird als offener Source Code zur Verfügung gestellt. Hierdurch ist es dem Entwickler auch möglich einen Blick in die Implementierung des Echtzeitsystems zu werfen. Dies ist besonders beim Verstehen des Kernels hilfreich. Da FreeRTOS auch in einer kommerziellen Version angeboten wird, kann davon ausgegangen werden, dass der Kernel auch langfristig Support erfährt. Ein Nachteil ist die komplizierte Einrichtung einer freien Entwicklungsumgebung wie Eclipse CDT. Um eine erste Beispielanwendung mit FreeRTOS auf dem Zielsystem zu implementieren, müssen diverse Konfigurationen an der Entwicklungsumgebung vorgenommen werden.

Literatur

1. R. Barry. *Mastering the FreeRtos Real time Kernel*. Real time Engineers Ltd., pre-release 161204 edition edition, 2016.
2. R. Barry. Freertos.org implemenation - advanced, 2017.
3. X. Fan. *Real-Time Embedded Systems*. Elsevier LTD, Oxford, 2015.
4. E. Glatz. *Betriebssysteme*. Dpunkt.Verlag GmbH, 2015.
5. D. B. Stewart. Measuring execution time and real-time performance. In *Embedded System Conference*, 2006.
6. A. S. Tanenbaum. *Moderne Betriebssysteme*. Pearson Studium, 2002.
7. C. Walls. Rtos revealed. *Embedded.com*, 2016-2017.