HOCHSCHULE
KONSTANZ
TECHNIK, WIRTSCHAFT
UND GESTALTUNG

AVR OS

Angewandte Systemprogrammierung

Name: Andreas Reinhardt
Matthias Weis
Betreuer: Daniel Urbanietz
Bearbeitungszeit: Wintersemester 15/16

Abstract

Im Rahmen der Lehrveranstaltung angewandte Systemprogrammierung wurde im Wintersemester 15/16 das hier beschriebene Betriebssystem für Microcontroller entwickelt. Außer den funktionalen Vorgaben gab es keine weiteren Bedingungen oder vorgefertigte Codestücke. Für das minimalistische Betriebssystem wurden Module wie der Dispatcher, Scheduler und Synchronisierungsmechanismen unter Betracht gezogen. Zudem sollten zwei zusätzliche, frei wählbare Features implementiert werden. Am Ende stand ein agiles, multitasking-fähiges Betriebssystem.

Inhaltsverzeichnis

Kapitel 1

Systemübersicht

Für ein minimalistisches Betriebssystem werden einige Grundbausteine benötigt. Zunächst muss die Hardware bzw. der Chip initialisiert werden, indem verschiedene Zeitquellen, Timer und die globale Interruptverwaltung konfiguriert werden (s. 1.1). Bevor das eigentliche Betriebssystem gestartet werden kann, müssen zudem diverse Strukturen für die Tasks aufgestellt werden, sodass diese unabhängig voneinander lauffähig sind (s. 1.2). Zum Starten des Systems wird mittels eines modifizierten Aufrufs der Dispatcher (s. 1.3) erstmalig durchlaufen und die erste Task zum Ausführen vorbereitet. Dabei wird auch der Scheduler aufgerufen, der, abhängig vom ausgewählten Algorithmus, die nächste Task bestimmt (s. 1.4).

Die soeben genannten Module würden für ein multitasking-fähiges Betriebssystem bereits ausreichen. Jedoch wäre sein Funktionsumfang sehr beschränkt. Bislang besteht noch keine Möglichkeit einer sicheren Interaktion zwischen zwei oder mehreren Tasks. Hierfür werden z.B. kritische Bereiche oder Signale benötigt, welche das Modul Synchronisation (s. 1.5) bereitstellt. Die Anwendung von Wartezeiten innerhalb Tasks setzt eine Systemzeit vorraus. Das Modul Zeiten (s. 1.6) verwendet hierfür Trigger der Hardware. Eine große Erleichterung im Entwicklungsprozess (und auch im späteren Anwendungsfall) ist die Verwendung von Collections (s. 1.7), welche Datenpakete hantierbar macht. Gleichzeitig verbessert sich mit ihnen die Lesbarkeit des Programmcodes.

1.1 Hardware

Als Zielplattform wurde die weit verbreitete AVR-Serie von Atmel verwendet. Jedoch wurde während der Umsetzung darauf geachtet, dass bei einem Plattformwechsel möglichst wenig Code angepasst werden muss. Als Entwicklungstool wurde der AVR-Simulator verwendet, welcher innherhalb des AVR-Studios von Atmel bereitgestellt wird. Als Zielbaustein wurde der ATxmega128A1 gewählt.

Die minimale Vorraussetzung an die Hardware ist ein Timer und ein Interruptsystem. Die Periode des Timers muss bekannt oder definierbar sein. Sind neben dem als Systemtimer verwendeten Timer noch weitere Timer verfügbar (wie das beim ATxmega128A1 der Fall ist), können diese für andere Zwecke (sei es Anwendung oder Betriebssystem) frei verwendet werden, unter der Bedingung, dass der Systemtimer nicht beeinflusst wird.

1.1.1 Zeitquellen

Zuallererst wird der interne Oszillator auf eine Systemfrequenz von $f_S = 32\text{MHz}$ konfiguriert. Dadurch können durch nachgeschaltete Teiler diverse Frequenzen an weiterer Peripherie erzeugt werden.

1.1.2 Dispatcher Timer

Damit nach Ablauf der Zeitscheibe einer Task der Dispatcher aufgerufen wird, muss ein Dispatcher Timer initialisiert werden. Dieser löst, unabhängig vom Ausführungszustand [1] der aktuellen Task, einen Interrupt aus, in dessen Interrupt Service Routine (ISR) der Dispatcher aufgerufen wird. Außerdem dient der Timer zur Berechnung der aktuellen Systemzeit.

```
1 void initDispatcherTimer(){
2         TCF0.CTRLB = TC_WGMODE_NORMAL_gc;
3         TCF0.PER = 0x7D00;
4         TCF0.INTCTRLA = TC_OVFINTLVL_HI_gc;
5 }
```

Code 1.1: Initialisierung Dispatcher Timer

In dieser Konfiguration zählt der Timer aufwärts bis zum Wert der Periode (`TCF0.PER`) und wird auf 0 zurückgesetzt. Gleichzeitig löst er beim Zurücksetzen einen Überlauf-Interrupt aus.

Wird als Zeitquelle die Systemfrequenz f_S ohne Vorteiler gewählt, so ergibt sich folgendes Intervall:

$$t_{per} = \frac{7\text{D}00_{16}}{32\text{MHz}} = 1\text{ms} \tag{1.1}$$

In der Annahme, dass keine Interruptsperren verwendet werden wird somit in Abständen von t_{per} ein Kontextwechsel herbeigeführt. Sollten Interruptsperren Anwendung finden liegt es am Entwickler, diese möglichst kurz zu halten. Solange sie deutlich unter t_{per} liegen gibt es keine Probleme. Bedenklich wird es, wenn die Ausführung des kritischen Bereichs länger als t_{per} dauert, da hierbei ein oder im schlimmsten Fall gleich mehrere Interruptsignale verloren gehen und dadurch u.a. die Systemzeit nicht mehr präzise ist.

1.1.3 Interrupts

Auch die Konfiguration des Interruptsystems ist nicht besonders aufwändig. Auf dem ATxmega128A1 muss jediglich das dem Dispatcher Timer entsprechende Interuptlevel scharf gestellt und das globale Interruptenable gesetzt werden.

[1] Es sei denn, die Task befindet sich in einem kritischen (Interrupt gesperrten) Abschnitt

Da die Funktionen zum Löschen oder Setzen des Interrupts auch mit dem Betreten oder Verlassen eines kritischen Abschnitts gleichzusetzen sind, werden sie plattformunabhängig exportiert. Wichtig hierbei ist nur, dass die Funktionen inline aufgerufen werden, sodass im Anwendungsfall der Stack nicht manipuliert wird. Dies wird beim AVR-Compiler durch das Attribut `always_inline` verwirklicht.

```
1 static void __attribute__((always_inline)) enableInterrupts(){
2         sei();
3 }
4 static void __attribute__((always_inline)) disableInterrupts(){
5         cli();
6 }
```

Code 1.2: Export der Interruptfunktionen

1.2 Strukturen

In einem multitaskingfähigen Betriebssystem müssen neben dem Stack für jede Task auch ein sogenannter Task-Control-Block (TCB) initialisiert werden. Dieser enthält Informationen über den aktuellen Zustand, die Priorität, den Stack-/pointer und Zeitangaben (s. 1.6).

1.2.1 Task-Control-Block

Als mögliche Taskzustände wurden die aus der Literatur und existierenden Betriebssystemen bekannten Zustände `READY`, `RUNNING`, `WAITING` und `KILLED` definiert. Bei der Priorität bedeutet eine kleinere Zahl einen höheren Vorrang (mit 0 als höchsten Wert).

```
1  typedef volatile struct strucTCB{
2          volatile uint8_t prio;
3          volatile uint8_t id;
4          volatile uint8_t* stackPointer;
5          volatile uint8_t* stackBeginn;
6          volatile uint16_t stackSize;
7          volatile uint32_t waitUntil;
8          volatile uint8_t tmpPrio;
9          volatile taskstate state;
10 }taskControlBlock;
```

Code 1.3: Task-Control-Block

1.2.2 Stack und Stackpointer

Zur Initialisierung eines TCB gehört auch der eigentliche Stack sowie dessen Pointer. Daher muss vor dem Start des Betriebsystems auch der Speicherbereich des Stacks reserviert und vorbereitet werden.

Zur Vorbereitung eines Stacks wird an seiner ersten und letzten Speicherstelle ein sogenannter Canary-Wert[2] eingefügt. Mit diesem soll erkannt werden, ob es zu einem Stacküberlauf gekommen ist (s. 1.3). Zudem wird an die vorletzte Stelle die Rücksprungadresse des Programmcodes kopiert, welcher in der jeweiligen Task ausgeführt werden soll. Der Task-Stackpointer wird daraufhin auf die Adresse x stellen überhalb der letzten Adresse gesetzt. Das x entspricht hierbei der Anzahl der CPU-Register.

Zum Starten des Systems wird bei der Starttask der Stackpointer nochmals verändert: Um gleich zur Rücksprungadresse bzw. Startadresse des Task-Programmcodes zu gelangen, wird der Stackpointer direkt überhalb dieser Adresse gesetzt und mittels `RET`-Befehl zu selbiger gesprungen.

```
1  void AVROS_start(taskControlBlock* firstTask, void (*scheduler_init)(void)){
2          currentTask = firstTask;
3          scheduler_init();
4          time_init();
5          firstTask->state = RUNNING;
6          mainSP = SP;
7          //set stack pointer of starting task next to taskaddress
8          SP = firstTask->stackBeginn+firstTask->stackSize-progcntOffset;
9          //start task
10         asm volatile ("ret");
11 }
```

Code 1.4: Systemstart

1.3 Dispatcher

Im späteren Betrieb soll es für jede Tasks so aussehen, als wäre sie die einzige im System. D.h. wird eine Task unterbrochen, müssen nicht nur alle Register und der CPU-Zustand gesichert, sondern auch der Stackpointer gespeichert und versetzt werden. Hierfür ist der Dispatcher zuständig. Wichtig hierbei ist, dass der Compiler keine eigenen, unkontrollierbaren Sicherungsversuche vornimmt, wie es üblicherweise der Fall ist. Um dies zu verhindern wird die ISR des Dispatchers für den AVR-Compiler mit dem Attribut `ISR_NAKED` versehen.

Der erste Schritt des Dispatchers besteht darin, sämtliche CPU-Register auf dem Stack der aktuellen Task zu sichern. Die Reihenfolge wie die Register auf den Stack gepushed werden ist dabei nicht maßgebend, jedoch muss beim Widerherstellen der Task in umgekehrter Reihenfolge vorgegangen werden. Nach der Sicherung wird der aktuelle Stackpointer im TCB (s. 1.2.1) abgelegt.

Der Dispatcher kann in zwei unterschiedlichen Situationen aufgerufen werden: Entweder die Zeitscheibe der Task ist abgelaufen und ein Dispatcher-Timer Interrupt wird ausgelöst oder die Task gibt die CPU freiwillig ab (s. Code 1.5). Beide Fälle müssen ab dieser Stelle unterschiedlich behandelt werden. Wurde der Dispatcher vom Timer ausgeführt (`hardwareISR = 1`), muss die

[2] z.B. $AA_{16} = 10101010_2$

Systemzeit erhöht werden und gleichzeitig alle bis zu diesem Zeitpunkt schlafenden Tasks aufgeweckt (s. 1.6) werden. Im anderen Fall (`hardwareISR = 0`) darf dieser Schritt nicht ausgeführt werden. Weiterhin muss unterschieden werden, in welchem Zustand sich die Task gerade befindet. Wurde sie blockiert (`state = WAITING`), muss nichts weiter unternommen werden. Ist sie jedoch noch aktiv (`state = RUNNING`) muss ihr Zustand auf `state = READY` gesetzt und im Scheduler neu eingereiht werden (s. 1.4).

Bevor der Scheduler aufgerufen wird, überprüft der Dispatcher mittels Magicnumber den Stack auf einen Überlauf. Sollte dies der Fall sein, wird ein `kernelPanic`-Fehler ausgelöst (s. 1.8).

1.4 Scheduler

Abhängig vom ausgewählten Algorithmus wählt der Scheduler die nächste auszuführende Task aus. Außerdem verwaltet er die anstehenden, lauffähigen Tasks. Zum Systemstart kann der gewünschte Scheduleralgorithmus ausgewählt werden(s. Code 1.4). Zudem kann er sogar in Laufzeit geändert werden. Dazu muss einfach die Initialisierungsfunktion des neuen Schedulers aufgerufen werden. Darin werden im System verwendete Funktionspointer auf die jeweiligen Algorithmenfunktionen gesetzt. Daher benötigt jede Implementierung eines Scheduleralgorithmus folgende Funktionen:

- `void scheduler_NextTask(void)`
- `void scheduler_enqueueTask(task)`
- `void scheduler_blockedByRessourceRequest(ressource)`
- `void scheduler_ressourceReleased(ressource)`

Die Funktion `scheduler_NextTask` wird bei jedem Dispatcherdurchlauf aufgerufen. Sie wählt entsprechend dem Algorithmus die nächste Task aus der Menge der lauffähigen Tasks aus.

`scheduler_enqueueTask` wird dann benötigt, wenn eine Task in den Zustand `READY` wechselt, unabhängig davon, welcher Zustand davor herrschte. So findet die Funktion u.a. beim Verdrängen einer Task im Dispatcher oder beim Aufwecken im Zeiten-Modul (s. 1.6) Anwendung.

Die Funktionen `scheduler_blockedByRessourceRequest` und `scheduler_ressourceReleased` werden nur von wenigen Scheduleralgorithmen benötigt. Im AVR OS sind sie ausschließlich im Prioritätenvererbungs-Algorithmus (PIP) zu finden. (s. 2.2) Sollten sie vom Algorithmus nicht verwendet werden müssen sie dennoch mit einem leeren Körper implementiert werden.

1.4.1 Scheduleralgorithmen

AVR OS unterstützt folgende Algorithmen:

- einfaches Round-Robin
- prioritätengesteuertes Round-Robin
- Prioritätenvererbung (PIP, s. 2.2)

Mit Ausnahme von PIP soll nicht näher auf die Implementierung der einzelnen Algorithmen eingegangen werden, da diese weitestgehend bekannt sind.

1.4.2 Idle-Task

Ist keine der Tasks zum gleichen Zeitpunkt im lauffähigen Zustand muss es dennoch eine Möglichkeit geben, dass der Dispatcher immerfort aufgerufen wird, damit neue lauffähige Tasks ausgewählt werden können. Die einfachste Umsetzung hierfür ist die Initialisierung einer sogenannten Idle-Task, welche in einer Dauerschleife "freiwillig" ihre Zeitscheibe beendet und dadurch den Dispatcher aufruft. Wichtig hierbei ist, dass die Idle-Task nie in einen blockierten Zustand versetzt werden darf und ihre Priorität auf das Minimum gesetzt wird. Letzteres verhindert, dass die Idle-Task ausgeführt wird, obwohl andere Tasks lauffähig sind.

```
1  void idleTask(){
2          while(1){
3                  task_yield();
4          }
5  }
6  void task_yield(){
7          disableInterrupts();
8          hardwareISR = 0;
9          DISPISRVEC();
10 }
```

Code 1.5: Idle-Task und yield

1.5 Synchronisation

Für die Interaktion zwischen Tasks und die Verwaltung von Ressourcen werden in Betriebssystemen Synchronisationsmechanismen verwendet. Ihre Funktionalität baut im Grunde auf den kritischen Abschnitt auf, welcher Interrupts sperrt. AVR OS unterstützt neben dem kritischen Abschnitt auch Signale und Semaphore (und damit auch Mutexe).

1.5.1 Semaphore und Signale

Dank der Verwendung der LinkedList und Interruptsperren war die Implementierung der Synchronisationsfunktionen vergleichsweise einfach. Als kleine Herausforderung stellten hierbei verschachtelte Interruptsperren dar. Gelöst wurde das Problem, indem vor dem Eintritt in die Interruptsperre der aktuelle Interruptzustand in eine lokale Variable gespeichert wird und anschließend beim Austritt überprüft wird, ob zuvor bereits eine Interruptsperre bestand. Ist dies der Fall, darf die Sperre nicht aufgehoben werden. Falls nicht werden die globalen Interrupts wieder aktiviert.[3]

```
1  uint8_t enterCriticalSection(){
2          uint8_t oldState = GLBINTFLG;
3          disableInterrupts();
4          return oldState;
5  }
6  void leaveCriticalSection(uint8_t oldState){
7          if(oldState){
8                  enableInterrupts();
9          }else{
10                 disableInterrupts();
11         }
12 }
```

Code 1.6: Ein- und Austritt der Interruptsperre mit Schachtelung

1.6 Zeiten

Zur Verzögerung einer Task für eine bestimmte Zeit wird das Zeiten-Modul benötigt. Damit ist es möglich, Tasks in einen schlafenden Zustand (bzw. `WAITING`) in Abhängigkeit der Zeit zu versetzen. In der späteren Anwendung ist das z.B. für periodische Aktionen sehr hilfreich.

Soll eine Task schlafen gelegt werden, wird im TCB (wie bereits erwähnt) ihr Status auf `WAITING` gesetzt. Zudem wird im Attribut `waitUntil` eingetragen bis zu welchem absoluten Systemzeitpunkt sie in diesem Zustand verweilen soll. Danach wird sie in einer LinkedList (s. 1.7) aufsteigend nach `waitUntil` einsortiert. Die Sortierung vereinfacht das spätere Aufwecken.

Wird nun der Dispatcher (s. 1.3) vom Dispatcher-Timer ausgelöst wird die Systemzeit inkrementiert. Aufgrund der Änderung muss überprüft werden, ob es Tasks gibt, die bis zu diesem Zeitpunkt ins Schlafen versetzt wurden. Dazu wird über die o.g. Liste iteriert, die entsprechenden Tasks aufgeweckt und zum Einsortieren in die Menge der lauffähigen Tasks an den Scheduler übergeben.

[3] Die Inspiration hierfür kam von BlueOS (©2013 Moritz Nagel, Daniel Urbanietz)

1.7 Collections

Beim Scheduling und auch bei den synchronisation Methoden wie Signalen und Mutex sind oft Listen von Tasks zu verwalten. Um dies zu vereinfachen wurde eine `LinkedList` implementiert.

Es handelt sich hierbei um eine gewöhnliche einfach verkettete Liste. Neben den Standartfunktionen, wie das Hinzufügen, Entfernen und Suchen eines Elements sind außerdem einige Hilfsfunktionen implementiert die z.B. das erste oder letzte Element einer Liste zurückgeben. Außerdem steht eine Iteratorfunktion zu Verfügung, die in Kombination mit einer `while`-Schleife eine Art `forEach`-Schleife ergibt. Ihre Anwendung ist im nachfolgenden Codebeispiel 1.7 demonstriert.

```
1 int value;
2 while(linkedList_iter(&linkedList, &value)){
3     handleValue(value);
4 }
```

Code 1.7: Iteratorfunktion der `LinkedList`

Es ist allerdings anzumerken, das die Implementierung es nur erlaubt Pointer in der `LinkedList` abzuspeicher, bzw. Datentypen, die die gleiche Größe haben.

1.7.1 Queue

Um die Verwaltung von Taskstrukturen noch weiter zu vereinfachen, wurde neben der `LinkedList` auch eine `Queue` implementiert. Es handelt sich hierbei um einen Wrapper um die `LinkedList`, der die von einer Queue bekannten Funktionen wie `pop` und `push` anbietet. Wenn möglich wurde die `Queue` im AVR OS verwendet. Sollte in der weiteren Entwicklung des Betriebssystems die `Queue` nicht mehr auf der `LinkedList` basieren, um diese etwa zu optimieren, wäre keine Anpassung des restlichen Codes nötig.

1.8 Exceptions

Im Entwicklungsprozess sind gute Debugging-Tools obligatorisch. Der verwendete AVR-Simulator bietet u.a. Breakpoints, Watches und Stacktracing an. Letzteres funktioniert allerdings nicht zuverlässig, weshalb eine eigene Stacktracing bzw. `kernelPanic`-Funktion implementiert wurde. Diese findet vor allem in den Tests (s. 3) Anwendung.

```
1 void kernelPanic(){
2         disableInterrupts();
3         uint32_t calledAt = __builtin_return_address(0);
4         while(1){
5                 asm volatile ("nop");
6         }
7 }
```

Code 1.8: `kernelPanic`-Exception

Zum Debugging bietet sich an, einen Breakpoint in Zeile 5 zu setzen. In der Variable `calledAt` kann dann die Rücksprungadresse ausgelesen werden. Diese wurde zuvor mithilfe der Compilerfunktion `__builtin_return_address(level)` ermittelt.

Kapitel 2

Features

2.1 Speicherverwaltung

Um dynamische Speicherallokierung zu ermöglichen, was z.B. bei der LinkedList nötig ist, wurde eine Speicherverwaltung implementiert. Die Speicherverwaltung erlaubt es Speicherbereiche beliebiger Größe zu allokieren. Der von der Speicherverwaltung genutzte Heap wird durch das anlegen eines Segment-Arrays reserviert. Für die Segmente ist eine Größe von 4 Byte gewählt worden.

2.1.1 MemoryHead

Zur Verwaltung der auf dem Heap vergebenen Speicherbereiche wird das vor dem jeweiligen Speicherbereich liegende Segment genutzt. Dieses Segment wird in einem solchen Fall MemoryHead genannt und ist folgendermaßen aufgebaut (siehe 2.1).

```
1 typedef struct{
2         uint16_t prev: 12;
3         uint16_t next: 12;
4         uint16_t size: 8;
5 }MemoryHead;
```

Code 2.1: MemoryHead Struct

Die Bitfelder `prev` und `next` geben den Index des vorherigen bzw. nächsten MemoryHead im Heap Array an. Durch die Limitierung auf 12 Bit können maximal 2^{12} Segmente bzw $4 \cdot 2^{12} = 16$kB addresiert werden. Eine Implentierung ohne ein `prev` wäre auch denkbar, also eine einfach verkettete Liste. Es vereinfacht aber das entfernen eines MemoryHeads aus der List wenn ein Speicherbereich freigegeben wird. Das `size` Feld gibt die Größe des zu verwaltenden Speicherbereichs an. Da für `size` 8 Bit bereitstehen können Speicherbereiche mit einer Größe von bis zu $4 \cdot 2^8 = 1$kB allokiert werden.

Es existiert zu jeder Zeit mindestens ein MemoryHead: das erste Segment des Heaps. Dieser dient auch als Einstiegspunkt für den Speicherverwaltungsalgorithmus.

2.1.2 Algorithmus

Die Speicherverwaltung legt keinen bestimmten Algorithmus fest, sonder erlaubt es beliebige Algorithmen zu implementieren. Dazu stellt die Speicherverwaltung zwei Funktionen und eine Hilfsstruktur zur Verfügung. Mit der Funktion `memoryManagement_next` können iterativ alle freien Speicherbereiche erfragt werden, die eine angegebene Größe nicht unterschreiten. Die erwünschte Größe wird als Teil der Struktur `MemoryRequest` übergeben. In dieser Struktur werden von der Funktion Informationen über den nächsten freien Speicherbereich gespeichert. Darunter die Id und die Größe des freien Speicherbereichs. Liefert die Funktion den Wert 0 zurück sind keine weiteren freien Speicherbereich der angegebenen Größe auf dem Heap verfügbar.

Um einen Speicherbereich zu belegen steht die Funktion `memoryManagement_alloc` zur Verfügung. Ihr wird beim Aufrufen ebenfalls ein `MemoryRequest` übergeben. Die in der `MemoryRequest` Stuktur enthaltene Id gibt dabei an welcher Speicherbereich belegt werden soll. Der Rückgabewert der Funktion ist die Adresse des allokierten Speicherbereichs.

Im nachfolgenden Codebeispiel 2.2 ist der Worst Fit Algorithmus mithilfe der beiden vorgestellten Funktionen implementiert.

```
1  uint16_t worstFit(uint16_t size){
2          MemoryRequest memoryRequest;
3          memoryManagement_initMemoryRequest(&memoryRequest, size);
4
5      uint16_t memId = 0;
6      uint16_t currentGapSize = MAX_UINT16;
7
8          while(memId = memoryManagement_next(&memoryRequest)){
9          uint16_t nextGapSize = memoryRequest.gapSize;
10
11         if(nextGapSize > currentGapSize){
12             currentGapSize = nextGapSize;
13             memId = memoryRequest.memId;
14         }
15         }
16
17     if(currentGapSize == MAX_UINT16){
18         return NULL;
19     }
20
21     memoryRequest.memId = memId;
22     return memoryManagement_alloc(&memoryRequest);
23 }
```

Code 2.2: Worst Fit Algorithmus

2.2 Scheduler: Prioritätenvererbung

Bei der Umsetzung des Priority-Inheritance-Protocols (PIP) wurde ein flexibler, selbst-lernender Ansatz erarbeitet. Dabei erweitert er im Grunde den prioritätengesteuerten Round-Robin Algorithmus um zwei Punkte:

1. Jede Ressource (darunter fallen auch Teile des Synchronisations-Moduls) erhält jeweils eine eigene Liste. Darin wird vermerkt, welche Tasks die Ressource jemals freigegeben haben.
2. Wird nun eine Task durch eine Ressourcenanforderung blockiert, überprüft der Algorithmus, ob eine Task aus der in 1. genannten Liste für eine Prioritätenvererbung in Frage kommt.

Diese Methode führt dazu, dass nachdem alle Tasks die benötigten Ressourcen mindestens einmal verwendet haben es zu keiner Prioritäteninversion kommen kann. Soll diese Bedingung gleich zu Beginn gelten, so kann optional die Freigabeliste bereits vom Entwickler vorgegeben werden.

2.2.1 Blockierung durch Ressourcenanforderung

Im Fall einer Blockierung wird über die o.g. Freigabeliste iteriert. Als Bedingung, dass eine Prioritätenvererbung stattfindet muss die bevorzugte Task eine niedrigere Priorität haben und lauffähig sein (`state = READY`). Ist beides erfüllt, wird der Task temporär die Priorität der blockierten Task zugewiesen und in die Menge der lauffähigen Tasks neu einsortiert.

```
1  void prioInheritance_blockedByRessourceRequest(LinkedList* resFreedBy){
2          uint8_t length = linkedList_length(resFreedBy);
3          taskControlBlock* nextTask;
4          if(length){
5                  for(uint16_t i = 0; i < length; i++){
6                          linkedList_get(resFreedBy, i, &nextTask);
7                          if(nextTask->state == READY && nextTask->prio > currentTask->
                               prio){
8                                  nextTask->prio = currentTask->prio;
9                                  queue_removeItem(&prioQueue[nextTask->tmpPrio], nextTask
                                       );
10                                 queue_push(&prioQueue[nextTask->prio], nextTask);
11                                 break;
12                         }
13                 }
14         }
15 }
```

Code 2.3: Verwaltung einer blockierten Ressourcenanforderung

2.2.2 Freigabe einer Ressource

Wird bei PIP eine Ressource freigegeben, wird zunächst überprüft, ob die freigebende Task bereits im der Freigabeliste enthalten ist. Ist dies nicht der Fall, wird sie eingetragen. Wichtiger jedoch muss erkannt werden, ob eine Prioritätenvererbung stattgefunden hat. Falls ja, wird die Priorität auf den ursprünglichen Wert (`tmpPrio`) zurückgesetzt und die Task verdrängt, da sie nun die Ressource nicht mehr für sich in Anspruch nimmt (s. Code 2.4 Zeile 7).

```
1 void prioInheritance_ressourceReleased(LinkedList* resFreedBy){
2         if(!linkedList_contains(resFreedBy, currentTask)){
3                 linkedList_append(resFreedBy, currentTask);
4         }
5         if(currentTask->tmpPrio != currentTask->prio){
6                 currentTask->prio = currentTask->tmpPrio;
7                 task_yield();
8         }
9 }
```

Code 2.4: Ressource wird freigegeben

Kapitel 3

Tests

Um die einzelnen Module des Betriebssystems unabhängig testen zu können wurde für den AVR-Simulator eine eigene Testbench entwickelt. Mit ihr ist es Möglich sämtliche Szenarien durchzutesten und auf Richtigkeit zu überprüfen.

3.1 Neustart

Für jeden Test sollten dabei eine definierte Umgebung vorliegen, welche unabhängig von zuvor ausgeführten Tests sein muss. Dadurch kann eine flexible Wiederholbarkeit erreicht werden. Damit alle Tests automatisch ausgeführt werden können wird ein Betriebssystemneustart benötigt. Dazu wird eine Interruptsperre verhängt, der Dispatcher-Timer gestoppt, der Stack geleert, alle CPU-Register zurückgesetzt und schließlich zur Programm-Adresse 0 gesprungen.

```
1  void beerOS_reboot(){
2          disableInterrupts();
3          stopDispatcherTimer();
4          //clear main stack
5          while(mainSP <= RAMEND){
6                  *mainSP = 0;
7                  mainSP++;
8          }
9          mainCalled--;
10         SP = 0;
11         asm volatile(
12                 "CLR␣R0\n\t"
13                 [...] //clear registers
14                 "BCLR␣7\n\t"
15                 );
16         asm volatile ("jmp␣0");
17 }
```

Code 3.1: Neustart

3.2 `.noinit`-Variablen

Zum Schutz vor einem fehlerhaften mehrfachen Aufrufen der `main`-Funktion wurde ein Zähler implementiert. Da bei einem Neustart ein solcher Aufruf gewollt ist wird die Zählervariable `mainCalled` dekrementiert. Vor dem Start der gewrappten Startfunktion `run` wird abgefragt, ob der `main`-Aufruf gewollt ist.

```
1 int main(void){
2         if(mainCalled){
3                 kernelPanic();
4         }
5         mainCalled++;
6         return run();
7 }
```

Code 3.2: `main`-Funktion

Globale Variablen werden vom Compiler in der Weise behandelt, dass sie beim Systemstart (Durchlauf der Programmadresse 0) initialisiert werden. Jedoch ist dieser Automatismus für zwei Fälle nicht gewünscht:

1. Würde die o.g. Variable `mainCalled` bei jedem Aufruf der Programmadresse 0 initialisiert bzw. zurück gesetzt werden, so wäre die Bedingung in Zeile 2 (Code 3.2) nie gültig, obwohl ein Fehlerfall vorläge.

2. Auch die Funktionspointer-Variable `void (*initNextTest)(void)`, in der der nächste auszuführende Test gespeichert wird, würde zurückgesetzt werden. Automatisiertes Testen wäre somit unmöglich.

Abhilfe hier schafft das Attribut `(section (".noinit"))` welches dem Compiler vermittelt, dass diese Variablen zwar angelegt, jedoch nicht initialisiert werden sollen. [4]

```
1 void (*initNextTest)(void) __attribute__ ((section (".noinit"))) = &initSemaphoreTest;
2 uint8_t mainCalled __attribute__ ((section (".noinit"))) = 0;
```

Code 3.3: `noinit`-Variablen

[4] In gewisser Weise werden sie beim ersten Start des AVR-Simulators auf die angegebenen Werte initialisiert. Daher entsteht beim Compilieren eine Fehlermeldung.

Abkürzungsverzeichnis

ISR ..... Interrupt Service Routine

TCB ..... Task-Control-Block