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
verweis
globale Interruptverwaltung konfiguriert werden. Bevor das eigentliche Betriebssystem gestartet werden kann, müssen zudem diverse Strukturen für die Tasks aufgestellt werden, sodass diese
verweis
unabhängig voneinander lauffähig sind. Zum Starten des Systems wird mittels eines modifizierten Aufrufs der Dispatcher erstmalig durchlaufen und die erste Task zum Ausführen vorbereitet. Dabei wird auch der Scheduler aufgerufen, der, abhängig vom ausgewählten Algorithmus, die nächste
verweis
Task bestimmt.

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 bereitstellt. Die Anwendung
verweis
von Wartezeiten innerhalb Tasks setzt eine Systemzeit vorraus. Das Modul Zeiten verwendet hierfür Trigger der Hardware. Eine große Erleichterung im Entwicklungsprozess (und auch im späteren Anwendungsfall) ist die Verwendung von Collections, 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 (wie im Fall vom ATxmega128A1) weitere Timer verfügbar, können diese vom Entwickler 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 .

verweis

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.

verweis

```
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 kritische Abschnitte verwendet werden wird somit in Abständen von t_{per} ein Kontextwechsel herbeigeführt. Sollten kritische Abschnitte 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 .

verweis scheduler

verweis schlafen

1.2.1 Task-Control-Block

Als mögliche Zustände einer Task wurden wie üblich `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 Zeiger. Daher muss vor dem Start des Betriebsystems auch der Speicherbereich des Stacks reserviert und vorbereitet werden.

verweis dispatcher

Zur Vorbereitung eines Stacks wird an seiner ersten und letzten Speicherstelle eine Magicnumber[2] eingefügt. Mit dieser soll erkannt werden, ob es zu einem Stacküberlauf gekommen ist. 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.

verweis dispatcher

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.

naming beer??

```
1  void beerOS_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 abgelegt.

verweis

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. Beide Fälle müssen ab dieser Stelle unterschiedlich behandelt werden. Wurde der Dispatcher vom Timer ausgeführt (`hardwareISR = 1`), muss die Systemzeit

s. yield

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

erhöht werden und gleichzeitig alle bis zu diesem Zeitpunkt schlafenden Tasks aufgeweckt 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.

verweis scheduler

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.

verweis exceptions

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. 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:

verweis systemstart

- `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 Anwendung.

verweis

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

name?

verweis PIP

1.4.1 Scheduleralgorithmen

name?

XXOS unterstützt folgende Algorithmen:

- einfaches Round-Robin
- prioritätengesteuertes Round-Robin

verweis

- Prioritätenvererbung (PIP)

Mit Ausnahme von PIP soll nicht näher auf die Implementierung der einzelnen Algorithmen eingegangen, 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 Maximum 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

name?

kritischen Abschnitt auf, welcher Interrupts sperrt. XXOS unterstützt neben dem kritischen Abschnitt auch Signale und Semaphore (und damit auch Mutexe).

1.5.1 Semaphore

1.5.2 Signale

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 verketteten Liste aufsteigend nach `waitUntil` einsortiert. Die Sortierung vereinfacht das spätere Aufwecken.

verweis

Wird nun der Dispatcher 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.

verweis

1.7 Collections

1.8 Exceptions

Kapitel 2

Features

2.1 Speicherverwaltung

2.2 Scheduler: Prioritätenvererbung

Kapitel 3

Tests

Abkürzungsverzeichnis

ISR ..... Interrupt Service Routine

TCB ..... Task-Control-Block

Tabellenverzeichnis

Abbildungsverzeichnis

To-do Liste