RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

LEHRSTUHL
INFORMATIK 11
SOFTWARE FÜR
EINGEBETTETE
SYSTEME

Praktikum Systemprogrammierung

Versuch 2

Scheduler / Stack

Lehrstuhl für Informatik 11 - RWTH Aachen

19. April 2011

Inhaltsverzeichnis

Dieses Dokument ist Teil der begleitenden Unterlagen zum *Praktikum Systemprogrammierung*. Alle zu diesem Praktikum benötigten Unterlagen stehen im L^{2}P-Lernraum unter http://www.elearning.rwth-aachen.de zum Download bereit.

Folgende Emailadresse ist für Kritik, Anregungen oder Verbesserungsvorschläge verfügbar:

psp@embedded.rwth-aachen.de

2 Scheduler / Stack

Beinahe jeder heutzutage verkaufte PC wird mit einem Multitasking Betriebssystem ausgeliefert, welches die Schnittstelle zwischen Anwendungsprogrammen und Hardware bildet. Die Kernaufgaben eines Multitasking Betriebssystems sind die Verwaltung von Anwendungsprogrammen und Prozessen, das Scheduling und die Verwaltung des verfügbaren Speichers und der Betriebsmittel. Diese Aufgaben werden von den Komponenten Scheduler, Taskmanager und Speicherverwaltung des Betriebssystems übernommen.

Als Scheduler wird diejenige Komponente des Betriebssystems bezeichnet, welche in regelmäßigen Zeitabständen das derzeit ausgeführte Programm unterbricht, um Betriebsmittel wie Prozessorzeit neu zuzuteilen. Der Taskmanager ist die (ggf. grafische) Benutzerschnittstelle, über welches ein Benutzer zur Laufzeit Einfluss auf das System ausüben kann. Die Speicherverwaltung organisiert bestimmte Teile des Arbeitsspeichers und stellt Anwendungen eine API (*Application Programming Interface*) zu Verfügung, über die zur Laufzeit des Systems dynamisch Speicher alloziert und freigegeben werden kann.

In diesem und den folgenden Versuchen wird die Funktionsweise dieser Komponenten erläutert. Dazu wird ein Multitasking Betriebssystem mit dem Namen SPOS für den Mikrocontroller ATmega 644 der Firma Atmel implementiert.

2.1 Versuchsinhalte

In diesem Versuch werden im Wesentlichen die Grundlagen für den Scheduler gelegt und eine damit verbundene Stackverwaltung erstellt. Der Scheduler ermöglicht das quasi gleichzeitige Ausführen mehrerer Prozesse. Weiterhin wird eine Routine zur Behandlung von Fehlern erstellt. Zusätzlich wird erläutert, wie der Speicher des Mikrocontrollers für SPOS aufgeteilt wird. Die dynamische Speicherallokation wird Inhalt späterer Versuche sein.

Achtung

Die folgenden Versuche werden auf den in diesem Versuch erzeugten Code aufbauen. Daher ist es wichtig, sorgfältig zu arbeiten und sich exakt an die Vorgaben aus dieser Versuchsbeschreibung zu halten.

2.2 Lernziel

Das Lernziel dieses Versuchs ist das Verständnis der folgenden Zusammenhänge:

- Funktionsweise eines Schedulers
- Verwaltung von kritischen Bereichen
- Verwaltung von Betriebssystemstacks
- Verwaltung von Anwendungsstacks
- Verwaltung von Prozessorregistern
- Verwaltung von Anwendungsprogrammen und Prozessen
- Fehlerbehandlung
- Erstellen von Anwendungsprogrammen

2.3 Grundlagen

Die Stackverwaltung, der Scheduler sowie die Verwaltung von Anwendungsprogrammen und Prozessen gehören zu den grundlegenden Aufgaben jedes Multitasking Betriebssystems. Diese drei Bereiche greifen ineinander und lassen sich kaum trennen. Es ist daher notwendig vor der Bearbeitung der Hausaufgaben das gesamte Grundlagenkapitel zu lesen.

Im Folgenden werden zunächst die Konzepte „Anwendungsprogramme und Prozesse“, „Scheduler“, „kritische Bereiche“ und „Stack“ vorgestellt. Nach der allgemeinen Darstellung dieser Konzepte wird die spezifische Umsetzung im Rahmen von SPOS diskutiert.

2.3.1 Anwendungsprogramme und Prozesse

Ein Betriebssystem unterscheidet zwischen Anwendungsprogrammen und Prozessen. Ein Anwendungsprogramm ist eine endliche Folge von Operationen. Ein Prozess ist eine laufende Instanz eines Anwendungsprogramms, zusammen mit einigen Verwaltungsdaten für das Betriebssystem. Es ist in Multitasking Betriebssystemen möglich, ein Anwendungsprogramm mehrfach zu starten. In diesem Fall werden mehrere voneinander unabhängige Prozesse mit eigenen Verwaltungsdaten angelegt.

In diesem Versuch werden nur einfache Arten von Anwendungsprogrammen unterschieden, deren Prozesse in einer Endlosschleife laufen und nicht terminieren. Die Behandlung von terminierenden Prozessen wird in Versuch 3 (*Heap / Schedulingstrategien*) erläutert.

2.3.2 Scheduler

Der Scheduler hat in Multitasking Betriebssystemen die Aufgabe, das Betriebsmittel Prozessorzeit zu verwalten und den Wechsel zwischen Prozessen und ihren Anwendungsstacks transparent durchzuführen. Jedem Prozess sollen Prozessorzeit und Stackspeicher idealerweise so zur Verfügung stehen, dass der Prozess nicht unterscheiden kann, ob er exklusiven Zugriff auf das System hat oder nicht.

Wenn ein Prozess vom Scheduler unterbrochen wird, müssen einige Verwaltungsdaten über diesen Prozess gespeichert werden. Sobald der Prozess zu einem späteren Zeitpunkt wieder aktiviert wird und ihm Rechenzeit zugewiesen werden soll, müssen diese Verwaltungsdaten wiederhergestellt werden. Weitere Informationen zum Scheduler in SPOS werden in Abschnitt 2.4.3 gegeben.

2.3.3 Kritische Bereiche

Ein Multitaskingbetriebssystem besitzt, wie jedes andere Betriebssystem, neben den Prozessen auch eine Vielzahl unterschiedlicher Betriebsmittel. Diese Betriebsmittel können Datenstrukturen, Schnittstellen (USART, USB, ...) oder andere Geräte wie Drucker oder ein Bildschirm sein. Die Prozesse teilen sich den Zugriff auf die Betriebsmittel. Es gibt häufig Situationen, in denen ein Prozess ein Betriebsmittel für eine Zeitspanne exklusiv nutzen möchte, was bedeutet, dass kein anderer Prozess in dieser Zeit auf dieses zugreifen

$t \cdot f_{OSC}$	Prozess 1	Prozess 2
0	`lcd_writeChar('H')`	
1	`lcd_writeChar('a')`	
2		`lcd_writeChar('W')`
3		`lcd_writeChar('e')`
4	`lcd_writeChar('l')`	
5	`lcd_writeChar('l')`	
6		`lcd_writeChar('l')`
7		`lcd_writeChar('t')`
8	`lcd_writeChar('o')`	

(a) Prozessablauf ohne kritische Bereiche.
Ausgabe: `HaWelllto`

$t \cdot f_{OSC}$	Prozess 1	Prozess 2
0	`enterCSection("lcd")`	
1	`lcd_writeChar('H')`	
2	`lcd_writeChar('a')`	
3	`lcd_writeChar('l')`	
4	`lcd_writeChar('l')`	
5	`lcd_writeChar('o')`	
6	`leaveCSection("lcd")`	
7		`enterCSection("lcd")`
8		`lcd_writeChar('W')`
9		`lcd_writeChar('e')`
10		`lcd_writeChar('l')`
11		`lcd_writeChar('t')`
12		`leaveCSection("lcd")`

(b) Prozessablauf mit kritischen Bereichen.
Augabe: `HalloWelt`

Abbildung 2.1: Anwendung kritischer Bereiche zur Vermeidung einer Prozessverzahnung.

darf. Eine derartige Nutzung, bzw. die Dauer dieser Nutzung, wird als kritischer Bereich bezeichnet.

Typischerweise werden kritische Bereiche verwendet, falls aus der gleichzeitigen Nutzung eines Betriebsmittels durch mehrere Prozesse, ein inkonsistenter Zustand des Systems resultierte.

Abbildung 2.1 zeigt ein simples Beispiel für die Anwendung und Notwendigkeit von kritischen Bereichen. Im Prozessablauf der Abbildung 2.1(a) wollen beide Prozesse das Betriebsmittel LCD nutzen, um Text auszugeben. Um das Beispiel anschaulich zu halten wird angenommen, dass der Scheduler den aktiven Prozess immer nach der Ausgabe von 2 Buchstaben wechselt. Die resultierende Ausgabe ist in dem Sinne inkonsistent, als dass eine unerwartete und ungewollte *Verzahnung* beider Zeichenketten erfolgt. In Abbildung 2.1(b) wird vor der Ausgabe des Textes von beiden Prozessen jeweils ein kritischer Bereich betreten, der das Betriebsmittel LCD für alle anderen Prozesse sperrt und erst wieder freigibt, wenn der jeweilige kritische Bereich verlassen wird. Nähere Informationen über die Implementierung von kritischen Bereichen in SPOS befinden sich in den Erläuterungen zu SPOS in Abschnitt 2.4.6.

2.3.4 Stack

Ein Stack (Keller- oder Stapelspeicher) speichert Daten nach dem LIFO-Prinzip (Last-In-First-Out). Das bedeutet, dass das Datum, welches zuletzt abgespeichert wurde, als erstes wieder gelesen wird. Auf dem Stack wird für gewöhnlich mit den Operationen „push“ (Ablegen eines neuen Datums oben auf dem Stack) und „pop“ (Auslesen und Herunternehmen des obersten Datums vom Stack) gearbeitet.

Der Zugriff auf einen Stack erfolgt über einen Pointer, der auf den ersten freien Platz eines zusammenhängenden Speichers zeigt, einen so genannten Stackpointer. Wird ein neues Datum auf den Stack gelegt, so wird es in der Speicherzelle abgelegt, auf die der Stackpointer zeigt. Danach wird der Stackpointer auf die nächste freie Speicherzelle

verschoben. Je nach Implementierung kann dies der Speicherblock mit der nächstgrößeren oder -kleineren Speicheradresse sein. Die Adresse des zuletzt abgelegten Datums ist dementsprechend um eins kleiner oder größer als der Stackpointer.

Abbildung 2.2 zeigt exemplarisch einen Stack mit einer maximalen Kapazität von 8 Datenblöcken und einem aktuellen Füllstand von 4 Datenblöcken, welcher in Richtung kleinerer Speicheradressen wächst.

Anwendungsstacks

In einem Multitasking Betriebssystem, welches mehrere gleichzeitig laufende Prozesse unterstützt, werden Stacks zur Zwischenspeicherung von Laufzeitkontexten, Rücksprungadressen beim Aufruf von Unterfunktionen und für die Speicherung lokaler Variablen benutzt. Diese speziellen Stacks werden *Anwendungsstacks* genannt und vom Betriebssystem verwaltet. Die Befehle zur Verwendung von Stacks werden im Normalfall vollständig vom Compiler in den Maschinencode der Anwendungsprogramme eingefügt. Jeder laufende Prozess erhält vom Betriebssystem seinen eigenen Anwendungsstack, der typischerweise von hohen zu niedrigen Speicheradressen wächst.

Lokale Variablen, die ein Prozess benutzt, sowie Rücksprungadressen aus Unterfunktionen, werden automatisch auf seinem Anwendungsstack abgelegt. Wird ein Prozess durch den Scheduler unterbrochen, darf kein anderer Prozess dessen Daten überschreiben. Aus diesem Grund wird für jeden Prozess ein eigener Anwendungsstack mit eigenem Stackpointer an einer festgelegten Speicheradresse zur Verfügung gestellt.

Anwendungsstack-Verwendung bei Unterfunktionsaufrufen

Wenn im Programmablauf eine Unterfunktion aufgerufen wird, müssen verschiedene Daten auf dem Anwendungsstack gespeichert werden, um den Prozess nach dem Abarbeiten der Unterfunktion korrekt fortsetzen zu können. Die Sicherung und Wiederherstellung der Daten wird im Allgemeinen durch den Compiler implementiert. Abbildung 2.3 zeigt einen Anwendungsstack nach dem Aufruf einer Unterfunktion. Die für den Unterfunktionsaufruf gespeicherten Daten, welche im Folgenden erläutert werden, liegen oben auf dem Anwendungsstack.

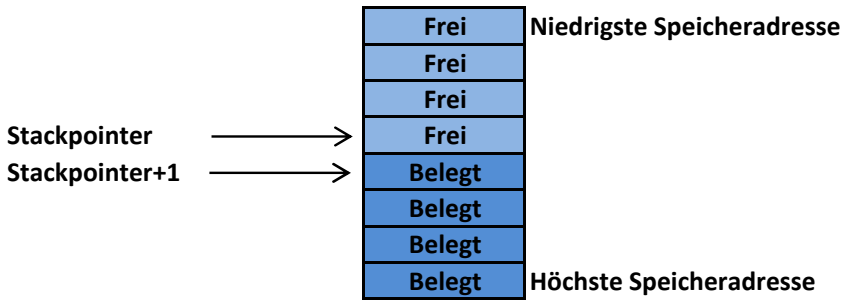

Abbildung 2.2: Beispiel für einen Stack.

Um das korrekte Fortsetzen des Prozesses zu ermöglichen, wird bei jedem Unterfunktionsaufruf als erstes eine Rücksprungadresse auf den Anwendungsstack gelegt. Die Rücksprungadresse bezeichnet die Stelle im Programmspeicher, an welcher der Prozess, nach Rückkehr aus der Unterfunktion, fortgesetzt werden muss. Diese Adresse ist beim ATmega 644 zwei Byte groß.

Bei der Abarbeitung einer Unterfunktion werden für gewöhnlich einige Prozessorregister geändert. Nachdem die Unterfunktion abgearbeitet wurde, sollen die Prozessorregister die Werte enthalten, welche sie vor dem Aufruf hatten. Daher werden die Werte der verwendeten Prozessorregister, die sich bei der Abarbeitung einer Unterfunktion ändern könnten, ebenfalls auf dem Anwendungsstack gesichert. Sobald die Unterfunktion abgearbeitet wurde, werden die Registerinhalte automatisch wiederhergestellt und es wird ein Sprung an die auf dem Anwendungsstack gespeicherte Rücksprungadresse ausgeführt. Diese Funktionalität wird üblicherweise ohne explizite Angabe durch den Compiler sichergestellt.

Stack Overflows

Es ist möglich, dass der Anwendungsstack eines Prozesses durch eine große Anzahl ineinander verschachtelter Unterfunktionen oder das Definieren einer großen Anzahl lokaler Variablen über die vorgesehene Größe hinaus anwächst. Die Werte, die dabei auf den Anwendungsstack geschrieben werden, überschreiben möglicherweise den Anwendungsstack eines anderen Prozesses oder einen anderen Datenbereich. Dies wird als „Stack Overflow“ bezeichnet.

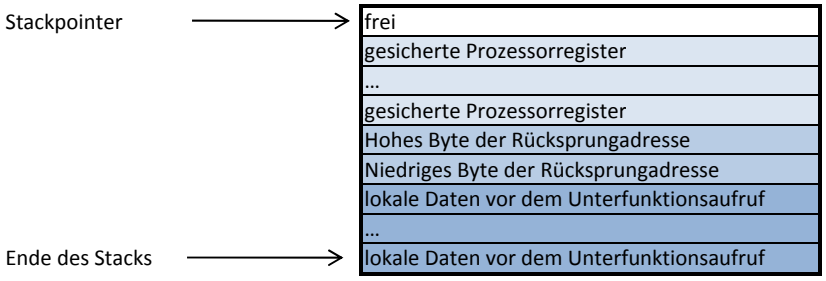

Abbildung 2.3: Der Anwendungsstack nach dem Aufruf einer Unterfunktion

Lernerfolgsfragen

- Wie werden Daten auf einem Stack gespeichert?
- Was bedeutet LIFO?
- Warum ist es nötig, jedem Prozess seinen eigenen Anwendungsstack zur Verfügung zu stellen, anstatt einen Stack für alle Prozesse zu verwenden?
- Welches (Fehl-)Verhalten des Programms ist möglich, wenn ein Stack Overflow aufgetreten ist?

Verwenden einer Heuristik zur Erkennung von Stackinkonsistenzen

Die Konsistenz eines Anwendungsstacks kann mit Hilfe einer Prüfsumme (*Hashwert*) bestimmt werden: Wenn ein Prozess einen Stack Overflow verursacht, und damit Daten auf dem Anwendungsstack eines anderen Prozesses überschreibt, so ändert sich mit hoher Wahrscheinlichkeit die Prüfsumme der überschriebenen Daten.

Um die Konsistenz eines Anwendungsstacks zu gewährleisten, kann dessen Hashwert bei der Unterbrechung des zugehörigen Prozesses durch den Scheduler gebildet werden. Bevor der Scheduler dem zugehörigen Prozess erneut Rechenzeit zuweist, wird für seinen Anwendungsstack ein neuer Hashwert bestimmt. Ist der Hashwert mit dem Wert identisch, der bei der letzten Unterbrechung errechnet wurde, so wurde der Anwendungsstack mit hoher Wahrscheinlichkeit nicht durch einen anderen Prozess überschrieben. Wenn der Hashwert nicht dem zuvor errechneten Wert entspricht, so wurden die Daten auf dem Anwendungsstack von einem anderen Prozess überschrieben, der Anwendungsstack ist somit inkonsistent.

Lernerfolgsfragen

- Wann tritt ein Stack Overflow auf?
- Was bedeutet *inkonsistent* im Zusammenhang mit Anwendungsstacks?
- Wie kann ein Stack Overflow erkannt werden?
- Können erkannte Stackinkonsistenzen behoben werden?

2.4 Hausaufgaben

Bisher wurden die Betriebssystemkonzepte allgemein behandelt. In diesem Abschnitt werden Informationen zur Implementierung dieser Konzepte in einem realen System gegeben. Im Rahmen des Praktikums Systemprogrammierung wird ein eigenes Betriebssystem mit dem Namen SPOS entwickelt, welches auf dem Mikrocontroller ATmega 644 ausgeführt wird. Dieses wird nach dem Abschluss dieses Versuchs die bereits vorgestellten Komponenten Programm- und Prozessverwaltung, Scheduler und Stackverwaltung beinhalten. In den folgenden Versuchen werden weitere Komponenten hinzugefügt. Zur Reduktion des Implementierungsaufwandes steht für diesen Versuch ein Grundgerüst zur Verfügung, in welchem bereits einige grundlegenden Funktionen implementiert sind und welches Ansätze für die in diesem Versuch vorgestellten Komponenten enthält. Dieses Grundgerüst soll im Rahmen der Versuchsvorbereitung als Hausaufgabe vervollständigt werden.

Lösen Sie alle hier vorgestellten Aufgaben zu Hause mit Hilfe des AVR Studio und schicken Sie die dabei erstellten funktionsfähigen Programme, einschließlich des AVR Studio-Projekts, über das im L^{2}P-Lernraum verlinkte Webformular ein. Beachten Sie bei der Bearbeitung der Aufgaben die angegebenen Hinweise zur Implementierung! Ihr Code muss ohne Fehler und ohne Warnungen kompilieren.

Achtung

Verwenden Sie zur Prüfung auf Warnungen den Befehl „Rebuild All“ im „Build“-Menü des AVR Studio. Die in der grafischen Oberfläche angezeigten Buttons „Build Active Configuration“, „Build and Run“, sowie „Build Active File “ führen nur ein inkrementelles Kompilieren aus; d.h. es werden nur geänderte Dateien neu kompiliert. Warnungen und Fehlermeldungen in ungeänderten Dateien werden dabei nicht ausgegeben.

2.4.1 Berechnung der Größe der Anwendungsstacks

Rechnen Sie aus, wie viel Speicher jedem Prozess als Anwendungsstack zur Verfügung steht. Passen Sie anhand dieser Ergebnisse die entsprechenden Definitionen (Listing 2.1) an. Die in dieser Aufgabe zu implementierenden bzw. zu ergänzenden Punkte sind im Einzelnen:

- Werte und Kenngrößen in `defines.h`

Hinweise zur Implementierung

Der SRAM-Speicher des Mikrocontrollers wird in mehrere Abschnitte unterteilt, welche in Abb. 2.4 in einer Übersicht dargestellt sind. Auf dem Speicher werden globale Variablen, ein Heap sowie die System- und Anwendungsstacks abgelegt. Für die Stacks soll

die Hälfte des SRAM verwendet werden, die andere Hälfte wird zwischen den globalen Variablen und dem Heap aufgeteilt. Der Heap wird in Versuch 3 (*Heap / Schedulingstrategien*) behandelt, während im Folgenden die genaue Aufteilung des Speicherbereichs für die Stacks beschrieben wird.

Der Stack-Bereich des SRAM teilt sich auf in den Bereich für die Systemstacks (Stack der Main-Funktion und der Scheduler-ISR) und den Bereich für die Anwendungsstacks. Zunächst müssen die Größen der System-Stacks bestimmt werden. Zur ungefähren Bestimmung dieser Werte können in der Main-Funktion und im Scheduler Breakpoints verwendet werden. Wenn an einem Breakpoint gehalten wird, kann die Größe des Stackpointers in AVR Studio ausgelesen werden. Der tatsächlich benötigte Platz muss, unter Zuhilfename des abgelesenen Wertes, nach oben abgeschätzt werden. Durch das Abschätzen ergibt sich ein Sicherheitspuffer, der verhindert, dass ein Stack Overflow auftritt.

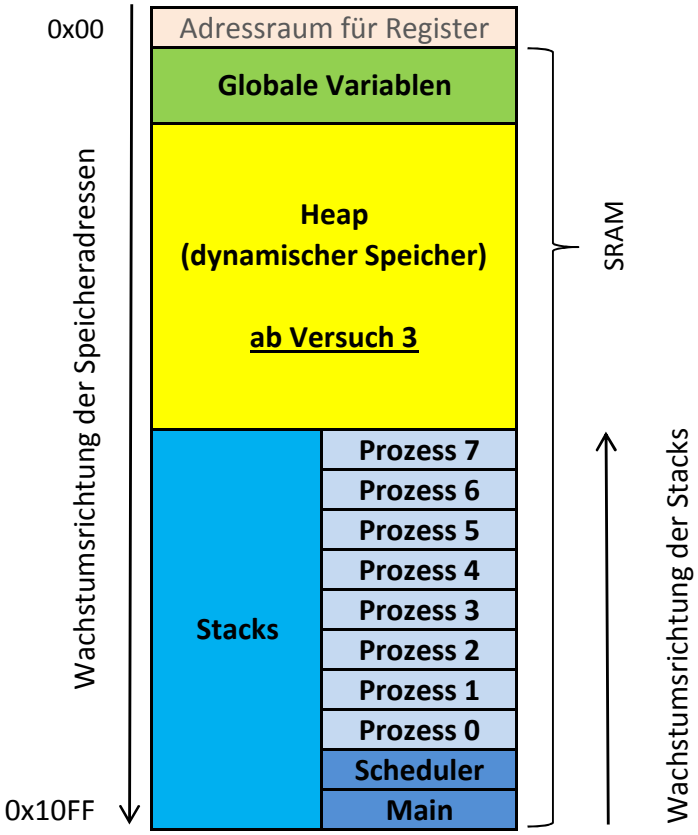

Abbildung 2.4: Speicheraufteilung

Hinweis

Der für den Stack der `main`-Funktion reservierte Speicher, wird nach der Startphase von SPOS nie wieder verwendet. Es ist möglich, beispielsweise den ISR- und Main-Stack zusammenzulegen, so dass kein Speicherbereich ungenutzt bleibt. Dies macht die Implementierung allerdings komplizierter und schafft viele Möglichkeiten für Fehler, so dass davon abgeraten wird.

Die Größe des Bereichs für die Anwendungsstacks ergibt sich, indem die Größe der Systemstacks von der Größe des Bereichs, der für alle Stacks zur Verfügung steht, abgezogen wird. Der Bereich für die Anwendungsstacks wird gleichmäßig in so viele einzelne Anwendungsstacks aufgeteilt, wie die maximale Anzahl gleichzeitig laufender Prozesse (`MAX_NUMBER_OF_PROCESSES`) vorgibt.

Aus der Größe der einzelnen Speicherbereiche lassen sich die Startadressen der einzelnen Stacks berechnen.

Achtung

Der SRAM des Mikrocontrollers wird **nicht** bei 0x0000 beginnend adressiert. Er teilt sich den Adressraum mit den Prozessor- und I/O-Registern. Um den Stack richtig planen zu können, ist es nötig, sich über die Adressierung im Datenblatt zu informieren.

Die ermittelten Werte sollen in der `defines.h` als Präprozessordefinitionen gespeichert werden. Zu den gespeicherten Informationen gehören mindestens die folgenden Werte:

- Größe der Betriebssystemstacks
- Maximale Größe für den Stack eines einzelnen Prozesses
- Startadresse des Stacks des nullten Prozesses

Zum Spezifizieren der Werte wird die `#define`-Direktive verwendet. Nähere Informationen zu dieser Direktive sind im begleitenden Dokument zum Praktikum Systemprogrammierung im L^{2}P-Lernraum zu finden.

Die Präprozessordirektiven aus Listing 2.1 sind bereits im Grundgerüst vorgegeben, und sind den Berechnungen entsprechend anzupassen. Die Namen der Direktiven sind beizubehalten.

```
1  //! The stack size available for initialization and
       globals
2  #define STACK_SIZE_MAIN ?
3  //! The scheduler's stack size
4  #define STACK_SIZE_ISR ?
5  //! The stack size of a process
6  #define STACK_SIZE_PROC ?
7
8  //! The bottom of the main stack. That is the highest
       address.
9  #define BOTTOM_OF_MAIN_STACK ?
10 //! The bottom of the scheduler-stack. That is the
       highest address.
11 #define BOTTOM_OF_ISR_STACK ?
12 //! The bottom of the memory chunks for all process
       stacks. That is the highest address.
13 #define BOTTOM_OF_PROCS_STACK ?
```

Listing 2.1: Präprozessordefinitionen zur Speicheraufteilung

Die Startadressen aller Anwendungsstacks können ausgehend von der Startadresse des ersten Anwendungsstacks und der maximalen Gesamtgröße der Anwendungsstacks berechnet werden. Bei der Berechnung dieser Größen ist zu beachten, dass die maximale Anzahl ausführbarer Prozesse durch ein `define` einstellbar ist. Es ist sinnvoll, bei Bedarf zusätzliche Defines anzulegen.

Hinweis

Verwenden Sie bei der Berechnung von `defines` vorrangegangene `defines`. Beispielsweise kann `STACK_SIZE_PROC` durch `STACK_SIZE_MAIN`, `STACK_SIZE_ISR` und `MAX_NUMBER_OF_PROCESSES` ausgedrückt werden. So können Sie den durch Änderungen eingehenden Korrekturaufwand verringern.

2.4.2 Implementierung der Verwaltung von Anwendungsprogrammen und Prozessen

Implementieren bzw. ergänzen Sie die Funktionen, Datentypen und Variablen zur Verwaltung von Anwendungsprogrammen und Prozessen. Hinweise zur Implementierung folgen im Anschluss an die folgende Aufzählung. Beachten Sie, dass Sie zur Bearbeitung dieser Aufgabe das gesamte Grundlagenkapitel gelesen haben müssen, da es zahlreiche Verflechtungen zwischen der Verwaltung von Anwendungsprogrammen und Prozessen und den Kapiteln Stack und Scheduler gibt! Die zu implementierenden, bzw. ergänzen-

den Punkte sind im Einzelnen:

- Datentypen:
  - `struct Process`
  - `enum ProcessState`
- Typbenennungen (`typedefs`):
  - `Process`, als Typbenennung für `struct Process`
  - `ProcessState`, als Typbenennung für `enum ProcessState`
  - `ProcessID`, als Typbenennung für `uint8_t`
  - `ProgramID`, als Typbenennung für `uint8_t`
  - `Program`, als Typbenennung für `void(void)`
    (Vgl. „Begleitendes Dokument zum Praktikum Systemprogrammierung“ Abschnitt Funktionspointer)
- Variablen, Arrays und Defines:
  - `MAX_NUMBER_OF_PROCESSES`
  - `MAX_NUMBER_OF_PROGRAMS`
  - `Program* os_programs[MAX_NUMBER_OF_PROGRAMS]`
  - `Process os_processes[MAX_NUMBER_OF_PROCESSES]`
- Funktionen:
  - `ProgramID os_registerProgram(Program* program)`
  - `void os_initScheduler(void)`
  - `ProcessID os_exec(ProgramID programID)`
  - `ProcessID os_getCurrentProc(void)`

Hinweise zur Implementierung

Damit das Betriebssystem mit Anwendungsprogrammen und Prozessen arbeiten kann, ist es nötig, zu jedem Anwendungsprogramm und zu jedem Prozess einige Verwaltungsinformationen bereitzuhalten, welche in einer speziell für diesen Zweck entwickelten, erweiterbaren Datenstruktur gespeichert werden. Diese Datenstruktur wird mit zusätzlichen anderen Informationen in den folgenden Abschnitten erläutert.

Anzahl gleichzeitig laufender Prozesse Die maximale Anzahl gleichzeitig laufender Prozesse, die SPOS akzeptieren soll, wird in der Datei `defines.h` als `MAX_NUMBER_OF_PROCESSES` festgelegt. Das Betriebssystem soll derart implementiert werden, dass bis zu acht gleichzeitig laufende Prozesse unterstützt werden. Es soll möglich sein, diesen Wert zur Entwurfszeit beliebig zwischen 1 und 8 zu ändern. Dieses Define wird später zur Skalierung verschiedener Betriebssystemkomponenten genutzt.

Anzahl verfügbarer Programme Analog zu der maximalen Anzahl gleichzeitig laufender Prozesse, soll die Anzahl verfügbarer Programme durch das `define MAX_NUMBER_OF_PROGRAMS` limitiert werden. Hier empfiehlt sich, den Wert von 20 nicht zu überschreiten.

Repräsentation von Anwendungsprogrammen und Prozessen in SPOS Anwendungsprogramme werden in SPOS als parameterlose C-Funktionen ohne Rückgabewert implementiert. Um SPOS ein Anwendungsprogramm zur Verfügung zu stellen, SPOS also mit diesem bekannt zu machen, muss ein Funktionspointer auf diese Funktion im Array `os_programs` gespeichert werden. Dieser Vorgang heißt *Registrieren* des Anwendungsprogramms. Das Array `os_programs` soll genau einen Funktionspointer auf jedes registrierte Anwendungsprogramm enthalten. Es muss vor der Verwendung in der Datei `os_scheduler.c` angelegt werden. Legen Sie zunächst eine Typbenennung `typedef void Program(void)` in `os_process.h` an. Anschließend können Sie ein Array mit Pointern auf „`Program`“ anlegen.

Die Funktion `os_registerProgram` hat die Aufgabe, Anwendungsprogramme für das Betriebssystem zu registrieren. Diese Funktion ist im Grundgerüst vorhanden, muss jedoch ergänzt werden. `os_registerProgram` soll jeden Funktionspointer nur einmal abspeichern, es muss also geprüft werden, ob ein zu registrierendes Anwendungsprogramm bereits in `os_programs` vorhanden ist. Als Rückgabewert soll die Funktion den Array-Index angeben, an dem der Funktionspointer für das Anwendungsprogramm gespeichert ist.

Prozesse, also Instanzen von Anwendungsprogrammen, haben mehrere Eigenschaften, die zur Verwaltung gespeichert werden müssen. Prozesse werden in einem Multitasking Betriebssystem durch den Scheduler beim Prozesswechsel unterbrochen und zu einem späteren Zeitpunkt fortgesetzt. Jeder Prozess hat einen eigenen Speicher, welcher derzeit noch ausschließlich aus einem Anwendungsstack besteht (Abschnitt 2.3.4). Damit das Betriebssystem Prozesse verwalten kann, müssen für jeden laufenden Prozess folgende Daten gesichert werden.

1. *Programm-ID*: Identifikationsnummer und gleichzeitig Position (Index) für das Array `os_programs`. Der Prozess ist eine Instanz des Anwendungsprogramms, welches in `os_programs[`Programm-ID`]` gespeichert ist
2. *Programmzähler*: Die Adresse des aktuell ausgeführten Befehls
3. *Laufzeitkontext*: Der aktuelle Inhalt der Prozessorregister und des Statusregisters `SREG`
4. *Prozesszustand*: Der Bereitschaftsstatus des Prozesses (z. B. Ready, Running, . . .).
5. *Stackpointer*: Pointer auf die nächste freie Speicherzelle des Anwendungsstacks des Prozesses

Hinweis

Aufgrund der Interaktion mit dem Taskmanager, müssen für die Attribute dieses Datentyps bestimmte Namen gewählt werden. Benennen Sie das Attribut mit der Programm-ID als `progID` und das Attribut mit dem Prozesszustand als `state`.

Es werden im weiteren Verlauf dieses Versuchs und in späteren Versuchen zusätzliche Informationen gespeichert werden. Verwenden Sie `ProcessState`, dass Sie zuvor als `typedef` für `enum ProcessState` benannt haben, als Datentyp für den Prozesszustand. Das `enum ProcessState` ist im Grundgerüst enthalten, muss jedoch vervollständigt werden. Füllen Sie die Struktur mit den Zuständen `OS_PS_UNUSED` für einen unbenutzten Slot, `OS_PS_READY` für einen auf Fortsetzung/Ausführung wartenden Prozess und `OS_PS_RUNNING` für einen gerade ausgeführten Prozess. Es ist verpflichtend diese Bezeichner zu wählen, damit der Code für die Betreuung wartbar bleibt.

Es ist eine Struktur `struct Process` zu modellieren, die einen Teil der zuvor genannten Prozesseigenschaften speichert. Diese Struktur ist im Grundgerüst, in der Datei `os_process.h`, enthalten, muss jedoch ergänzt werden. Denken Sie beim Anlegen daran, es sofort mittels `typedef` als `Process` zu benennen, um das `struct` später lesbarer und komfortabler verwenden zu können. In dieser Struktur werden folgende Werte gespeichert: Programm-ID, Prozesszustand und Stackpointer. Der Programmzähler und der Laufzeitkontext werden nicht in dieser Datenstruktur gespeichert. Informationen zur Speicherung dieser Daten sind in Abschnitt 2.4.2 unter „Anwendungsstack vorbereiten“ enthalten.

Um dem Scheduler den Zugriff auf alle Prozesse zu ermöglichen, muss das global verfügbare Array `os_processes` des Typs `Process` in der Datei `os_scheduler.c` angelegt werden. Dieses Array enthält für jeden Prozess je einen Eintrag des Typs `Process*`, also einem Zeiger auf den `Process`-Typ.

Die Funktion `void os_initScheduler(void)` initialisiert die unmittelbar zum Scheduling benötigten Daten. Zuerst muss das Array `os_processes` so gefüllt werden, dass das Datenfeld `state` aller Einträge `OS_PS_UNUSED` enthält. Außerdem muss ergänzt werden, dass die Funktion das automatische Starten von Prozessen verwaltet. Dazu wird überprüft, ob Funktionspointereinträge im Array `os_programs` vorhanden sind. Gleichzeitig muss überprüft werden, ob für das jeweilige Programm das Flag zum automatischen Starten gesetzt ist. Dazu kann die vorgegebene Funktion `os_checkAutostart Program(ProgramID prog)` mit gleichem Programmindex verwendet werden. Ist ein Programm zum automatischen Starten markiert, soll es an dieser Stelle mit `os_exec(...)` gestartet werden.

Der derzeit laufende Prozess muss jederzeit durch einen Aufruf der Funktion `ProcessID os_getCurrentProc(void)` abfragbar sein.

Starten von Anwendungsprogrammen / Erzeugen von Prozessen Der Start eines Anwendungsprogramms ist ähnlich dem Erzeugen eines neuen Prozesses aus dem Anwen-

dungsprogramm. Dazu wird die Funktion `os_exec` verwendet, welche im Grundgerüst angelegt ist und ergänzt werden muss. Diese Funktion hat die Aufgabe, im nächsten freien Slot des Arrays `os_processes` die notwendigen Informationen über den neu erzeugten Prozess abzulegen und führt dazu die folgenden Schritte durch:

1. Einen kritischen Bereich betreten
2. Freien Platz im Array `os_processes` finden
3. Den zu dem angegebenen Programmindex gehörigen Funktionspointer aus dem Array `os_programs` auf den Anwendungsstack kopieren
4. Programmindex und Prozesszustand speichern
5. Anwendungsstack vorbereiten
6. Den kritischen Bereich verlassen

Sobald alle diese Schritte durchgeführt wurden, steht der neue Prozess dem Scheduler (Abschnitt 2.4.3) zur Verfügung.

Einen kritischen Bereich betreten/verlassen Die Funktion `os_exec` muss atomar abgearbeitet werden, um zu garantieren, dass die gespeicherten Informationen für einen Prozess korrekt und vollständig sind. Deshalb darf sie während der Ausführung nicht unterbrochen werden. Hierzu muss ein kritischer Bereich mit der Funktion `os_enterCriticalSection` betreten und vor Ende von `os_exec` mit der Funktion `os_leaveCriticalSection` verlassen werden. Verwenden Sie an dieser Stelle zunächst einfach den jeweils genannten Funktionsaufruf ohne weitere Implementierung. Die eigentliche Funktionalität der Methoden für kritische Bereiche wird in Aufgabe 2.4.6 realisiert.

Funktionspointer aus dem Array `os_programs` laden Zum Laden des Funktionspointers soll die Funktion `Program* os_lookupProgramFunction(ProgramID programID)` verwendet werden, die im Fehlerfall, etwa wenn ein ungültiger Programmindex übergeben wurde, `INVALID_PROGRAM` zurückgibt. Dies ist eine Präprozessordefinition, die einen ungültigen Indexwert definiert. Der Funktionsrumpf ist bereits im Grundgerüst enthalten, muss jedoch ergänzt werden. Wenn `INVALID_PRORGAM` zurückgegeben wurde, soll die Funktion `os_exec` abgebrochen werden. Dieser wiederum muss im Fall eines Abbruchs `INVALID_PROCESS` als Rückgabewert liefern.

Prozesszustand speichern Der Prozesszustand gibt den aktuellen Status eines Prozesses an. In diesem Versuch können Prozesse die Zustände *Ready* (der Prozess wartet darauf, dass der Scheduler ihm Rechenzeit zuweist) und *Running* (der Prozess wird zur Zeit ausgeführt) annehmen. Wenn ein neuer Prozess erzeugt wird, soll dieser initial den Zustand *Ready* erhalten. In späteren Versuchen werden weitere Zustände hinzugefügt. Der Prozesszustand hat den Datentyp `ProcessState`. Der `enum`-Typ ist in der Datei `os_process.h` vorgegeben und muss ergänzt und als `ProcessState` benannt werden.

Anwendungsstack vorbereiten Das Starten eines Prozesses wird wie der Rücksprung aus einer Unterfunktion behandelt. Um einen Prozess zu starten, müssen daher entsprechende Daten im Voraus auf dessen Anwendungsstack gespeichert werden. Auf dem Anwendungsstack werden die Prozessinformationen Programmzähler (als Rücksprungadresse) und Laufzeitkontext (als gesicherte Registerinhalte) gespeichert.

Als Programmzähler wird für den Start eines Prozesses, der zu dessen Programm im Array `os_programs` hinterlegte Funktionspointer verwendet, da dieser auf den Beginn der Anwendungsfunktion zeigt. Das niedrige Byte der insgesamt 16 Bit breiten Adresse muss muss hierbei als erstes auf dem Stack gesichert werden. An der nächstkleineren Stelle [1] werden dann die höherwertigen Bits geschrieben. Der Laufzeitkontext besteht aus den 32 Prozessorregistern und dem Statusregister `SREG` des ATmega 644, die alle mit `0` initialisiert werden müssen. Daher werden 33 Nullen nach dem Funktionspointer auf den Anwendungsstack geschrieben. Unter der Rücksprungadresse sollen auf dem Anwendungsstack des neuen Prozesses keine Daten liegen.

Der Compiler trifft bei Optimierungen bestimmte Annahmen. Eine dieser Annahmen ist, dass bestimmte Register, welche mit `0` initialisiert werden, höchstens vom Compiler selbst verändert werden. Falls solche Annahmen verletzt werden, z.B. indem die Inhalte der Prozessorregister durch Werte aus uninitialisierten Anwendungsstacks ersetzt werden, kann dies in falschem Maschinencode resultieren. Dies ist der Grund, warum 32 Nullen auf den Stack geschrieben werden.

Liegen alle benötigten Werte auf dem Anwendungsstack, muss der Stackpointer des neuen Prozesses auf die erste freie Speicherstelle des Anwendungsstacks gesetzt werden.

2.4.3 Implementierung eines Schedulers

Erweitern Sie die im Grundgerüst vorhandene Scheduler-ISR `TIMER2_COMPA_vect` um die Fähigkeit, zwischen verschiedenen Prozessen zu wechseln. Dazu ist außerdem die Implementierung einer Schedulingstrategie notwendig. Die Strategie Random wurde bereits als Beispiel im Grundgerüst vorgegeben. Die in dieser Aufgabe zu entwickelnden Teile des Betriebssystems sind also:

- Datentypen:
  - `enum SchedulingStrategy`
- Typbenennungen (`typedefs`):
  - `SchedulingStrategy`, als Typbenennung für `enum SchedulingStrategy`
- Funktionen:
  - `TIMER2_COMPA_vect(...)`
  - `ProcessID os_Scheduler_Even(...)`
  - `void os_setSchedulingStrategy(...)`
  - `SchedulingStrategy os_getSchedulingStrategy(void)`

[1] Beachten Sie: Der Stack wächst in Richtung „kleinerer“ Adressen, siehe Abbildung 2.4.

Hinweise zur Implementierung

Der Scheduler in SPOS hat die Aufgabe, Rechenzeit zwischen gestarteten Prozessen zuzuteilen. Er muss regelmäßig aufgerufen werden, um dem Benutzer den Eindruck zu vermitteln, dass Prozesse gleichzeitig ausgeführt werden.

Zur Erfüllung dieser Anforderungen, unterbricht der Scheduler regelmäßig die Abarbeitung von Prozessen und setzt diese zu einem späteren Zeitpunkt wieder fort. Bei der Unterbrechung müssen einige Daten gespeichert werden, damit jeder Prozess nach dem Fortsetzen denselben Laufzeitkontext zur Verfügung hat, wie vor der Unterbrechung. Diese Daten müssen also gesichert werden, wenn ein Prozess unterbrochen wird, bzw. wiederhergestellt werden, wenn er fortgesetzt wird. Ebenso muss der aktuelle Programmzähler durch den Scheduler gespeichert werden. Ab dem Programmzähler wird ein unterbrochener Prozess später fortgesetzt.

Hinweis

Während der Abarbeitung benötigt die Schedulerfunktion einen eigenen Stack. Würde sie den Anwendungsstack eines Prozesses benutzen, könnte es zu einem Stack Overflow kommen. Nähere Informationen zur Speicheraufteilung sind in Abschnitt 2.4.1 zu finden.

Wechsel zwischen Anwendungsstacks in SPOS Der vom Prozessor verwendete Stackpointer ist im Register `SP` gespeichert. Das Register `SP` enthält die Adresse des ersten freien Speicherbereichs des aktuell verwendeten Anwendungsstacks. Um zwischen zwei Anwendungsstacks zu wechseln, wird zunächst der Inhalt des Registers `SP` als Stackpointer des aktuellen Prozesses gespeichert. Hierfür existiert bereits ein `union`, das als `StackPointer` benannt wurde und im Grundgerüst vorgegeben ist. Anschließend wird das Register `SP` auf den Stackpointer des nächsten Prozesses gesetzt. Nachfolgende Stackoperationen beziehen sich anschließend auf den Anwendungsstack des nächsten Prozesses.

Implementierung des Schedulers als Interrupt Service Routine eines Timers Damit der Scheduler regelmäßig aufgerufen wird, wird er als Interrupt Service Routine (ISR) eines Timers implementiert. Die Ausführung der Scheduler-ISR in hinreichend kurzen Abständen ist gewährleistet, wenn Sie die Timer-Einstellungen verwenden, die im Grundgerüst des Projekts bereits vorgegeben sind.

Das Grundgerüst enthält sowohl die (leere) Timer-ISR `TIMER2_COMPA_vect` in der Datei `os_scheduler.c`, welche den Scheduler enthalten soll, wie auch die Konfiguration des zugehörigen Timers in der Datei `os_core.c`. Timer 2 lässt sich so konfigurieren, dass beim Erreichen eines von zwei voreingestellten Werten, A bzw. B, ein entsprechender Interrupt ausgelöst wird. Die zugehörige ISR `TIMER2_COMPA_vect` wird beim Auftreten eines Compare-A-Interrupts von Timer 2 aufgerufen, also wenn der Counter von Timer 2

den Wert A erreicht hat. In der vorgegebenen Konfiguration ist A auf den Wert 60 Timer-Ticks eingestellt. Damit die ISR automatisch aufgerufen wird, müssen die Interrupts global aktiviert sein. Sobald ein Interrupt ausgelöst wird, werden die Interrupts durch den Prozessor automatisch global deaktiviert, sie müssen also zu Beginn der ISR **nicht** explizit deaktiviert werden.

Hinweis

Es ist ebenfalls möglich, die ISR manuell mit dem Befehl `TIMER2_COMPA_vect()` aufzurufen. Der manuelle Aufruf wird vor allem in den folgenden Versuchen von Bedeutung sein.

Weiterführende Informationen zur Interruptbehandlung und zur Verwendung von Timern und Timer-Interrupts stehen unter anderem im begleitenden Dokument zum Praktikum Systemprogrammierung und im Datenblatt zur Verfügung.

Ablauf des Schedulers und Prozesswechsel Der Ablauf des Schedulers ergibt sich wie folgt:

1. Speichern des Programmzählers als Rücksprungadresse für die spätere Fortsetzung des aktuellen Prozesses auf dessen Anwendungsstack. Dies wird implizit durch den Compiler beim Sprung in die ISR erledigt.
2. Sichern des Laufzeitkontextes des aktuellen Prozesses auf dessen Anwendungsstack. Dies erfolgt durch ein Assemblermakro.
3. Sichern des Stackpointers für den Anwendungsstack des aktuellen Prozesses.
4. Setzen des Stackpointers auf den ISR-Stack.
5. Setzen des Prozesszustandes des aktuellen Prozesses auf *Ready*.
6. Auswahl des nächsten fortzusetzenden Prozesses.
7. Setzen des Prozesszustandes des fortzusetzenden Prozesses auf *Running*.
8. Wiederherstellen des Stackpointers für den Anwendungsstack des fortzusetzenden Prozesses.
9. Wiederherstellen des Laufzeitkontextes des fortzusetzenden Prozesses und Rücksprung.

Wenn diese Operationen ausgeführt worden sind, steht dem fortgesetzten Prozess der Laufzeitkontext zur Verfügung, den dieser vor der Unterbrechung durch den Scheduler hatte. Die Ausführung wird an der Stelle fortgesetzt, an der sie zuvor durch den Scheduler unterbrochen wurde. Der Kontext der ISR (und somit der Wert des Stackpointers am

Ende der ISR) muss nicht gespeichert werden, da eine ISR nicht erwartet, auf dem Stack Daten zu finden. Es ist daher vollkommen richtig, den Stackpointer bei jedem Aufruf der ISR wieder auf den Beginn des ISR-Stacks zu setzen.

Informationen zur Verwendung des Stacks beim Prozesswechsel Wie in Abschnitt 2.3.4 beschrieben, werden bei jedem Unterfunktionsaufruf bestimmte Daten auf dem Anwendungsstack gesichert. Dies gilt auch für den Aufruf einer ISR, wie z.B. der Scheduler-ISR. Weil der Scheduler zwischen verschiedenen Prozessen wechselt, ist dem Compiler beim Aufruf der Scheduler-ISR nicht bekannt, welche Register geändert werden, bevor der aktuelle Prozess vom Scheduler das nächste Mal fortgesetzt wird. Aus diesem Grund muss das Sichern und Wiederherstellen der Prozessorregister und des Spezialregisters `SREG` explizit erfolgen. Dadurch wird garantiert, dass alle Register gesichert werden. Allerdings ist es nicht empfehlenswert, diese Sicherung in C durchzuführen, da der Zugriff auf ein Register zur Veränderung von Werten in den anderen Registern führen kann. Der Zugriff auf die Register muss deshalb durch einen Umweg über „Inline-Assembler“ erfolgen (siehe begleitendes Dokument zum Praktikum Systemprogrammierung). In der Datei `util.h` ist das bereits funktionsfähige Assemblermakro `saveContext()` enthalten, mit dessen Hilfe der Laufzeitkontext (bestehend aus den 32 Registern r0 bis r31, sowie dem Spezialregister `SREG`) seiteneffektfrei auf den Anwendungstack gesichert wird.

Die in einer Funktion verwendeten Register werden durch den Compiler vor dem Überschreiben implizit auf den Stack gesichert. Da jedoch aus Effizienzgründen nur die tatsächlich benutzten Register gesichert werden, muss die Sicherung beim Betreten der ISR explizit vorgenommen werden.

Um zu verhindern, dass Registerinhalte sowohl explizit als auch implizit durch den Compiler gesichert werden, muss die implizite Sicherung deaktiviert werden. Dazu gibt es die gcc-Erweiterung `naked`, mit der erzwungen werden kann, dass der Compiler lediglich die Rücksprungadresse auf den Stack speichert. Die Verwendung ist in Listing 2.2 dargestellt.

```
1 // .h Datei
2 ISR (TIMER2_COMPA_vect) __attribute__ ( ( naked ) );
3
4 // .c Datei
5 ISR (TIMER2_COMPA_vect) {
6   ...
7 }
```

Listing 2.2: naked bei Header- und C-Datei

Das `naked`-Attribut ist bereits im Grundgerüst enthalten. Es verhindert nicht nur das automatische Sichern, sondern auch das automatische Wiederherstellen der Register und den Rücksprung in den vom Scheduler unterbrochenen Prozess. Das Wiederherstellen der Registerinhalte und der Rücksprung müssen daher ebenfalls manuell erfolgen. Dazu ist das Assemblermakro `restoreContext()` vorgegeben. Dieses Makro nimmt die

Registerinhalte vom aktiven Anwendungsstack und speichert diese in den Prozessorregistern und im Spezialregister. Danach führt es den Assemblerbefehl `reti`[2] aus, welcher die Rücksprungadresse vom aktiven Anwendungsstack lädt, zu dieser Adresse zurückspringt und die Interrupts global einschaltet, da diese beim Auslösen eines Interrupts durch den Prozessor deaktiviert werden.

Schedulingstrategien Um festzustellen, welcher Prozess in einem Multitaskingbetriebssystem bei einem Scheduling die Kontrolle über den Prozessor erhalten soll, wird eine so genannte *Schedulingstrategie* verwendet. Dem Scheduler sollen mehrere verschiedene Schedulingstrategien zur Verfügung stehen, zwischen denen beliebig gewählt werden kann. In diesem Versuch werden zunächst zwei einfache Schedulingstrategien vorgestellt. Eine weitere Strategie wird in der Zusatzaufgabe im Versuchsteil erläutert. In Versuch 3 (*Heap / Schedulingstrategien*) werden weitere Schedulingstrategien behandelt.

Um die Auswahl zwischen mehreren Schedulingstrategien zu ermöglichen, kann eine globale Variable angelegt werden, die speichert, welche Strategie aktuell verwendet wird. Verwenden Sie zur Speicherung der aktuellen Schedulingstrategie den enum-Typ `SchedulingStrategy`, welcher im Grundgerüst bereits enthalten ist und vervollständigt werden muss[3]. Diese Variable soll aus Gründen der Datenkapselung nicht projektweit zur Verfügung stehen. Lese- und Schreibvorgänge müssen daher über die beiden Zugriffsfunktionen `os_getSchedulingStrategy` und `os_setSchedulingStrategy`, geschehen. Diese Funktionen sind im Grundgerüst bereits vorhanden und müssen ergänzt werden.

Als mögliche Strategien werden in diesem Versuch die *Random-Strategie* und die *Even-Strategie* betrachtet. Die Random-Strategie wählt zufällig einen der laufenden Prozesse aus während die Even-Strategie die laufenden Prozesse in einem Ringzähler durchläuft, so dass auf den letzten aktiven Prozess wieder der erste folgt. Alle Schedulingstrategien sollen ausschließlich Prozesse auswählen, welche sich im Zustand *Ready* befinden. Der Leerlaufprozess sollte nur dann ausgewählt werden, wenn kein anderer Prozess verfügbar ist. Eine sehr grundlegende Implementierungsmöglichkeit der Random Strategie wurde exemplarisch vorgegeben. Sie soll dahingehend verbessert werden, dass sie den Leerlaufprozess nur dann auswählt, wenn kein anderer Prozess verfügbar ist. Zum Verständnis der Funktion ist es hilfreich, die Möglichkeiten zur Generierung von Zufallszahlen zu kennen. Informationen dazu befinden sich im begleitenden Dokument zum Praktikum Systemprogrammierung.

2.4.4 Erkennung von Stackinkonsistenzen

Erweitern Sie die zuvor erstellte Scheduler-ISR `TIMER2_COMPA_vect` um die Fähigkeit Stackinkonsistenzen zu entdecken. Dazu ist insbesondere die Erstellung einer Funktion `os_getStackChecksum` notwendig. Die in dieser Aufgabe zu entwickelnden Teile des Betriebssystems sind:

[2] `reti` = **ret**urn from **i**nterrupt. Zum Zurückspringen aus einer normalen Unterfunktion würde `ret` verwendet werden.

[3] Verwenden Sie auch bei diesem `enum` vorgegebene Bezeichner: OS_SS_RANDOM und OS_SS_EVEN.

- Funktion:
  - `StackCheckSum os_getStackChecksum(ProcessID pid)`

Hinweise zur Implementierung

Wie schon in dem entsprechenden Abschnitt des Grundlagenkapitels erläutert, soll in diesem Versuch eine Erkennungsheuristik für Inkonsistenzen der Anwendungsstacks, wie sie etwa bei einem Stack Overflow auftreten, implementiert werden. Diese Heuristik soll mit einer Hashfunktion arbeiten, welche die Prüfsumme eines Anwendungsstacks errechnet. Zur Berechnung der Prüfsumme eines Anwendungsstacks kann prinzipiell jede Hashfunktion verwendet werden. Eine dieser Funktionen erhält man dadurch, dass alle Bytes des Anwendungsstacks mit dem bitweisen XOR verknüpft werden. Diese Hashfunktion hat den Vorteil, dass sie mit wenig Aufwand zu berechnen ist und Aufrufe dieser Funktion daher in relativ kurzer Zeit abgearbeitet werden können. Implementieren Sie in der Datei `os_scheduler.c` die Hashfunktion `StackCheckSum os_getStackChecksum(ProcessID pid)`, welche sich wie folgt verhält: Die Funktion erhält die ID eines Prozesses als Parameter übergeben und berechnet die Prüfsumme des Anwendungsstacks dieses Prozesses. Dazu verknüpft sie die einzelnen Bytes, die auf dem Anwendungsstack liegen, mit dem bitweisen XOR. Das Ergebnis dieser Berechnungen wird als Prüfsumme zurückgegeben. Der Typ `StackCheckSum` soll ein `typedef` für einen Datentyp mit geeignetem Wertebereich sein. Es genügt, die Daten in die Überprüfung einzubeziehen, die zwischen dem Anfang des Anwendungsstacks und dem Stackpointer des entsprechenden Prozesses liegen. Daten, die jenseits dieses Stackpointers liegen, werden vom Prozess nicht genutzt und sind für die Konsistenz irrelevant.

Verwenden Sie die implementierte Hashfunktion in der Scheduler-ISR, um die Konsistenz der Anwendungsstacks zu überwachen. Erweitern Sie dazu die Struktur `struct Process` um ein Attribut, das den Hashwert des Anwendungsstacks speichert. Erweitern Sie nun den Scheduler, so dass er nach dem Sichern des Kontextes eines unterbrochenen Prozesses den Hashwert seines Anwendungsstacks ermittelt und abspeichert. Sorgen Sie dafür, dass dieser Wert mit dem aktuellen Hashwert verglichen wird, bevor der unterbrochene Prozess wieder fortgesetzt wird. Der Prozess darf nur fortgesetzt werden, wenn die beiden Werte übereinstimmen. Sollte sich der Hashwert geändert haben, genügt es, eine Fehlermeldung auszugeben und SPOS anzuhalten, weitere Maßnahmen sind nicht erforderlich. Die Behandlung von Fehlermeldungen wird in einem weiteren Abschnitt erläutert.

Hinweis

- Beachten Sie beim Starten von Prozessen, dass der Hashwert geeignet initialisiert werden muss.
- Es muss nicht geprüft werden, ob der Anwendungsstack des Prozesses mit der höchsten ID in den Heap-Bereich des Speichers übergelaufen ist.

2.4.5 Implementierung einer Schnittstelle zur Fehlerbehandlung

Vervollständigen Sie die im Grundgerüst vorhandene Funktion `os_errorPStr` und fügen Sie an geeigneten Stellen in Ihrem Quellcode Prüfungen auf Fehler ein. Im Fehlerfall soll die Funktion `os_errorPStr` mit einer geeigneten Fehlermeldung aufgerufen werden. In dieser Aufgabe ist zu bearbeiten:

- Funktionen:
  - `void os_errorPStr(prog_char const* str)`
  - `void freeze(void)`

Hinweise zur Implementierung An einigen Stellen des Codes ist eine Überprüfung auf Fehlverhalten notwendig. Ein solches Fehlverhalten liegt zum Beispiel vor, wenn kein Anwendungsprogramm registriert wurde oder zu viele Prozesse gestartet werden sollen. Es ist empfehlenswert, weitere sinnvolle Überprüfungen zu implementieren und entsprechende Fehlermeldungen auszugeben.

Zur Umsetzung der Fehlerprüfung ist die Funktion `os_errorPStr` in der Datei `os_core.c` zu implementieren. Diese Funktion soll die übergebene Fehlermeldung auf dem Display ausgeben und das Betriebssystem anhalten. Mit der Tastenkombination "Button 1 + Button 4" wird die Fehlermeldung vom Benutzer quittiert und das Betriebssystem läuft weiter. Verwenden Sie zur Abfrage der Buttons keine Interrupts, sondern fragen Sie den Status der Buttons in einer Schleife ab (Polling). Es sind keine weiteren Maßnahmen bzgl. des gefundenen Fehlers zu ergreifen. Es ist durchaus möglich, dass sich SPOS nach dem manuellen Quittieren eines Fehlers nicht mehr korrekt verhält. Aus diesem Grund sollte die Möglichkeit der manuellen Quittierung nur zu Debuggingzwecken genutzt werden. Die Fehlerbehandlung wird vor allem in den nächsten Versuchen von Bedeutung sein.

Zeichenketten nehmen sehr viel Speicher in Anspruch. Da der SRAM-Speicher des ATmega 644 bereits zu großen Teilen für den Heap und die Anwendungsstacks benutzt wird, ist es nicht sinnvoll, zusätzlich eine große Anzahl von Fehlermeldungen dort zu speichern. Es besteht jedoch die Möglichkeit, Daten im Flash-Speicher des ATmega 644 abzulegen, der deutlich größer ist. Ein im Flash-Speicher abgelegtes Zeichen besitzt den Datentyp `prog_char`. Eine Zeichenkette im Flash-Speicher wird im Rahmen dieses Praktikums

entsprechend `prog_String` genannt; dies ist jedoch kein offizieller Datentyp. Weitere Informationen zu `prog_char` sind im begleitenden Dokument zu finden. Um Zeichenketten im Flash-Speicher abzulegen, kann ein `define` der Bibliothek `<avr/pgmspace.h>` verwendet werden. Die Definition `PSTR(s)` kennzeichnet den angegebenen String `s` für den Compiler zur Ablage im Flash-Speicher. Eine solche Zeichenkette kann mit dem Befehl `lcd_writeProgString` ausgegeben werden. Die Verwendung ist in Listing 2.3 exemplarisch gezeigt.

```
1  void example(void){
2     //Aufruf von os_errorPStr
3     os_errorPStr(PSTR("Dies ist ein Fehler"));
4     //Äquivalenter Aufruf direkt mit os_error
5     os_error("Dies ist Fehler 2");
6  }
7
8  //Bereits vorhandenes Makro:
9  #define os_error(str) os_errorPStr(PSTR(str))
10
11 void os_errorPStr(prog_char const* str){
12    /* Zeigt den String str mittels des Befehls
13     * lcd_writeProgString() aus der LCD-
14     * Bibliothek an und hält das Betriebssystem an,
15     * bis der Fehler quittiert wird.
16     */
17 }
```

Listing 2.3: Verwendung von PSTR(s)

Im Grundgerüst ist bereits das Makro `os_error` vorgegeben, welches erlaubt, Fehler direkt anzugeben, ohne jedes mal `PSTR` vor den eigentlichen Fehler zu schreiben.

Implementieren Sie die Hilfsfunktion `freeze`, welche das gesamte System komplett einfrieren soll. Sie wird später für Fehler benötigt, die eine Fehlerquittierung und Fortsetzung des Betriebssystems unmöglich machen. Legen Sie `freeze` in `util.c` an und bedenken Sie die Deklaration in der zugehörigen Header Datei.

2.4.6 Implementierung von kritischen Bereichen

Implementieren Sie die im Codegerüst enthaltenen Funktionen zur Verwendung kritischer Bereiche.
Überlegen Sie sich mögliche Fehlerquellen, die in den Funktionen `os_enterCriticalSection` und `os_leaveCriticalSection` auftreten können und ermöglichen Sie die Ausgabe von Fehlern mittels geeigneter Meldungen für `os_errorPStr`.
Die in dieser Aufgabe zu bearbeitenden Teile des Betriebssystems sind:

- Variable:
  - `criticalSectionCount` (Wertebereich 0-255)

- Funktionen:
  - `void os_enterCriticalSection(void)`
  - `void os_leaveCriticalSection(void)`
- geeignete Fehlermeldungen für:
  - `void os_enterCriticalSection(void)`
  - `void os_leaveCriticalSection(void)`

Hinweise zur Implementierung

In Abschnitt 2.3.3 wurde das Prinzip von kritischen Bereichen konzeptionell eingeführt. In SPOS soll dieses ebenfalls umgesetzt werden, um das konfliktfreie Nutzen von Betriebsmitteln gewährleisten zu können. Kritische Bereiche sperren einzelne Betriebsmittel für andere Prozesse. Da eine Sperre für einzelne Betriebsmittel relativ aufwändig zu implementieren ist, soll SPOS ein vereinfachtes Verfahren anwenden: Das Betreten eines kritischen Bereichs durch einen Prozess soll alle anderen Prozesse daran hindern, aufgerufen zu werden. Dazu wird der Scheduler während der Verarbeitung eines kritischen Bereiches deaktiviert. Um kritische Bereiche zu verwalten, sollen die Funktionen `void os_enterCriticalSection(void)` und `void os_leaveCriticalSection(void)` implementiert werden. Diese Funktionen sollen einen kritischen Bereich betreten, bzw. verlassen, indem sie den Scheduler aus- bzw. wieder einschalten. Diese Funktionen müssen atomar sein, dürfen also nicht durch Interrupts unterbrochen werden. Um den Scheduler in der Funktion `os_enterCriticalSection` zu deaktivieren, muss das Bit `OCIE2A` im entsprechenden Register von Timer 2 deaktiviert werden. Das Lösches des Bits sorgt dafür, dass es nicht zu einem Timer Interrupt und somit zu einem Scheduleraufruf kommt. Lesen Sie dazu Kapitel 15 im Datenblatt. Da evtl. in Unterfunktionen kritische Bereiche aufgerufen werden, obwohl schon ein kritischer Bereich betreten wurde, müssen Verschachtelungen beachtet werden. Legen Sie dazu eine Variable `criticalSectionCount` mit geeignetem Typ an, um die Verschachtelungstiefe speichern zu können. Diese Variable soll nur in der Datei `os_scheduler.c` sichtbar sein. Der Ablauf beim Betreten von `os_enterCriticalSection` sollte sich an den folgenden Punkten orientieren:

1. Speichern des Global Interrupt Enable Bit aus dem `SREG` in einer lokalen Variablen.
2. Deaktivieren des Global Interrupt Enable Bit, um das atomare Verhalten der Funktion zu gewährleisten.
3. Inkrementieren der Verschachtelungstiefe des kritischen Bereiches.
4. Deaktivieren des Schedulers.
5. Wiederherstellen des (zuvor gespeicherten) Zustandes des Global Interrupt Enable Bit im `SREG`.

`os_leaveCriticalSection` funktioniert analog zu `os_enterCriticalSection`, mit dem Unterschied, dass der Scheduler nur dann wieder aktiviert wird, wenn nach dem Dekrementieren der Verschachtelungstiefe des aktuellen Prozesses diese den Wert Null angenommen hat. Somit wird der Scheduler erst nach dem Verlassen aller ineinander verschachtelten kritischen Bereiche wieder aktiviert.

Es empfiehlt sich an geeigneten Stellen innerhalb dieser Funktionen Fehlerüberprüfungen durchzuführen.

Hinweis

Es wäre denkbar die kritischen Bereiche durch Deaktivieren des globalen Interrupt-Bits zu implementieren. Es sei darauf hingewiesen, dass dies nicht hinreichend ist. Prozesse sollen später durch andere Interrupts, unabhängig von der Scheduler-ISR, unterbrechbar sein. Die kritischen Bereiche sollen ausschließlich den Prozesswechsel vermeiden.

2.4.7 Implementierung von Anwendungsprogrammen

Erstellen Sie drei Anwendungsprogramme. Ergänzen Sie dazu die bereits vorgegebenen Programmfunktionen im Code-Gerüst, die in einer Endlosschleife jeweils die Buchstaben „A“, „B“ bzw. „C“ auf dem LCD ausgeben sollen und dann einige Millisekunden warten, so dass die Buchstaben auf dem Display gut lesbar sind. Informationen zur Verwendung des LCD finden Sie im begleitenden Dokument zum Praktikum Systemprogrammierung. Von der Aufgabenstellung betroffen ist:

- Funktionen:
  - `PROGRAM(1, ...)`
  - `PROGRAM(2, ...)`
  - `PROGRAM(3, ...)`

Hinweise zur Implementierung

- Ein Ihnen vorgegebenes Präprozessor-Makro legt aus den obigen Definitionen zunächst Programmfunktionen an. Diese werden automatisch im Betriebssystem registriert (das Makro legt den Funktionspointer an der angegebenen Stelle im Array `os_programs` ab). Als zweites Argument muss angegeben werden, ob das Programm automatisch gestartet werden soll (`AUTOSTART`), oder ob dies nicht automatisch geschehen soll (`DONTSTART`). Im letzteren Fall kann das Programm dann dennoch über `os_exec` jederzeit gestartet werden.
- Es ist möglich, Anwendungsprogramme zu schreiben, die ohne dieses Makro registriert und aufgerufen werden. Dazu müsste man selbst `os_registerProgram(...)` und `os_exec(ProgramID programID, Priority priority)` aufrufen.

- `DEFAULT_OUTPUT_DELAY` stellt den Standardwert für die Wartezeit bei Ausgaben dar. Er ist in der Datei `defines.h` deklariert und muss um einen sinnvollen Wert ergänzt werden. Das `define` enthält den Wartewert für Displayausgaben in Millisekunden.

- Die zum Ansteuern des LCD benötigten Funktionen befinden sich in den Dateien `lcd_Lcd.c /.h`. Die Zugriffe auf das LCD sind so implementiert, dass sie atomar abgearbeitet werden, d.h. sie werden nicht vom Scheduler unterbrochen.

- Achten Sie darauf, dass im Projekt mindestens die Compileroptimierung `-O1` eingestellt ist (Project → Configuration Options). Ohne Optimierung (Option `-O0`) wird Ihr Betriebssystem nicht zuverlässig funktionieren, da einige Funktionen, insbesondere die Zugriffsfunktionen für das LCD, ohne Optimierung nicht mehr das notwendige Timing einhalten können.

2.4.8 Implementierung und Einbinden eines Leerlaufprozesses

Fügen Sie der `os_scheduler.c` ein Anwendungsprogramm `PROGRAM(0, AUTOSTART)` hinzu, das in einer Endlosschleife einen Punkt „.“ schreibt und um `DEFAULT_OUTPUT_DELAY` Millisekunden wartet. Dieses Anwendungsprogramm wird *Leerlaufprogramm*, seine laufende Instanz *Leerlaufprozess* genannt. Der Leerlaufprozess ist der erste Prozess, dem Rechenzeit zugeteilt werden soll. Sorgen Sie daher dafür, dass der Leerlaufprozess am Ende der Funktion `os_startScheduler` als erstes gestartet wird.

Hinweise zur Implementierung

- Die in `os_startScheduler` durchzuführenden Schritte sind analog zu den letzten Aktionen in der Scheduler-ISR.

- Nach dem Assemblermakro `restoreContext` kann kein weiterer Befehl stehen!

2.4.9 Aufrufen des Taskmanagers

Der Taskmanager für SPOS wurde Ihnen bereits vorgegeben. Die Dateien `os_taskman.c /.h`, sowie `os_user_privileges.c /.h` sind bereits im Codegerüst enthalten. Der Taskmanager soll aufgerufen werden, wenn Button1 und Button4 des Evaluationsboards gleichzeitig gedrückt werden. Ergänzen Sie den Aufruf des Taskmanagers in Ihrer Implementierung des Betriebssystems.

Hinweise zur Implementierung

- Das Überprüfen der Buttons soll mittels Polling geschehen. Damit der Taskmanager nicht beliebig, sondern immer aus einem definierten Zustand heraus aufgerufen wird, sollen die Buttons in der Scheduler-ISR geprüft werden. Es ist sinnvoll, die Überprüfung der Buttons nach Schritt 4 und vor Schritt 5 des in 2.4.3 vorgestellten Schedulerablaufs vorzunehmen und ggfs. den Taskmanager dann aufzurufen.

- Beachten Sie, dass nicht nur das Drücken der Buttons geprüft werden muss, sondern auch, ob die Taster wieder losgelassen wurden. Ansonsten würde der startende Taskmanager die noch gedrückten Tasten sofort wieder als Input auffassen, was verhindert werden soll.
- Der Taskmanager wurde bereits für alle kommenden Versuche implementiert und ist vollständig im Grundgerüst enthalten. Da er Funktionen enthält, die in diesem Versuch noch nicht verfügbar sind, kann er mit einem `define` an den jeweiligen Versuch angepasst werden. Zu diesem Zweck existiert das `define VERSUCH` in der Datei `defines.h`.

2.4.10 Test und Simulation

Testen Sie Ihre Implementierung ausgiebig. Dazu können Sie auch den im CIP-Pool des RBI (Raum 4U18) stehenden PC mit angeschlossenem Mikrocontrollerboard verwenden. Wenn Sie Ihre Implementierung mittels Simulation testen wollen, beachten Sie das Define `SIMULATION` in der `defines.h`. Dieser Eintrag muss für die Simulation angepasst werden, da in der Simulation die Konfiguration der Timer geändert werden muss. Dazu greifen einige der vorgegebenen Funktionen auf den Eintrag zurück. Entfernen Sie die entsprechende Zeile also nicht, und definieren Sie `SIMULATION` entsprechend als `1` oder `0`.

2.5 Präsenzaufgaben

2.5.1 Testprogramme

Sie erhalten während des Versuchs von den Betreuern eine Sammlung von Testprogrammen, mit denen die Funktionalität Ihres Betriebssystems zum aktuellen Entwicklungsstand getestet werden kann. Wenn Ihre selbst entwickelten Anwendungsprogramme fehlerfrei unterstützt werden, starten Sie die Testprogramme. Wenn es mit diesen Probleme gibt, haben Sie wahrscheinlich bestimmte Anforderungen nicht erfüllt, oder gewisse Sonderfälle nicht bedacht. Diese Sonderfälle können bei der Erweiterung Ihrer Implementierung des Betriebssystems für spätere Versuche zu Folgefehlern führen. Ergänzen oder korrigieren Sie Ihr Projekt, wenn Probleme mit den Testprogrammen auftreten.

Sollten Sie alle zur Verfügung gestellten Testprogramme erfolgreich ausführen können, ist dies kein Garant dafür, dass Ihr Code fehlerfrei ist! Diese Testprogramme decken lediglich einen Teil der möglichen Fehler ab.

2.5.2 Implementierung einer dritten Schedulingstrategie (in V. 2 optional)

Sollten Sie alle anderen Aufgaben vor Ende des Praktikumstermins abgearbeitet haben, implementieren Sie als dritte Schedulingstrategie die RoundRobin-Strategie, welche nach Prozessprioritäten vorgeht:

Bei dieser Strategie erhält jeder Prozess, sobald er an die Reihe kommt, vom Scheduler eine sog. Zeitscheibe zugeteilt, die sich bei jedem Scheduleraufruf um 1 verringert. Diese Zeitscheibe entspricht der Prozesspriorität und soll bei der Vergabe immer mindestens die Größe 1 besitzen. Ein Prozess bleibt solange aktiv, wie die Zeitscheibe dies vorgibt. Wenn die Zeitscheibe des aktuellen Prozesses abgelaufen ist, wechselt der Scheduler wie beim Ringzähler zum nächsten aktiven Prozess und teilt diesem eine neue Zeitscheibe zu. Die abgelaufene Zeitscheibe wird dann auf die Priorität zurückgesetzt, um beim nächsten Aufruf wieder die volle Zeit zur Verfügung zu haben.

Hinweise zur Implementierung Bedenken Sie, dass Sie die Priorität des Prozesses und den Wert der aktuellen Zeitscheibe speichern und initialisieren müssen.

Legen Sie hierzu in der Datei `os_scheduling_strategies.c` eine Struktur `struct SchedulingInformation` an, welche zunächst nur den Wert `timeSlice` mit geeignetem Wertebereich enthalten soll. Die Struktur ist nötig, da in kommenden Versuchen weitere Attribute hinzugefügt werden. Benennen Sie die Struktur mittels `typedef` als `SchedulingInformation`. Die Prozesspriorität wird dagegen in `struct Process` gespeichert, da Sie sich nicht laufend ändert.

Verringern Sie die Zeitscheibe des aktiven Tasks bei jedem Scheduleraufruf um eins. Die Zeitscheibe ist abgelaufen, wenn sie den Wert `0` erreicht. Denken Sie daran, dass einige Funktionen die Priorität als zusätzlichen Parameter erhalten müssen. Diese Funktionen müssen Sie in den Header- und den C-Dateien anpassen, so wie sie in der Doxygen-Dokumentation vorgegeben sind. Dazu wird es notwendig sein, auch für die Priorität erneut eine Typbenennung eines `uint8_t` zu `Priority` vorzunehmen.

2.6 Pinbelegungen

Port	Pin	Belegung
PORTA	1	frei
	2	LCD Pin 6
	3	LCD Pin 5
	4	LCD Pin 4
	5	LCD Pin 3
	6	LCD Pin 2
	7	LCD Pin 1
	8	LCD Pin 0
PORTB	1	frei
	2	frei
	3	frei
	4	frei
	5	frei
	6	frei
	7	frei
	8	frei
PORTC	1	Button 1
	2	Button 2
	3	Reserviert für JTAG
	4	Reserviert für JTAG
	5	Reserviert für JTAG
	6	Reserviert für JTAG
	7	Button 3
	8	Button 4
PORTD	1	frei
	2	frei
	3	frei
	4	frei
	5	frei
	6	frei
	7	frei
	8	frei

Pinbelegung für Versuch 2 (*Scheduler / Stack*).