Informatik 11
Embedded Software

RWTH AACHEN UNIVERSITY

Praktikum Systemprogrammierung

Versuch 3

Heap / Schedulingstrategien

Lehrstuhl Informatik 11 - RWTH Aachen

6. April 2023

Commit: b3175525

Inhaltsverzeichnis

Dieses Dokument ist Teil der begleitenden Unterlagen zum *Praktikum Systemprogrammierung*. Alle zu diesem Praktikum benötigten Unterlagen stehen im Moodle-Lernraum unter https://moodle.rwth-aachen.de zum Download bereit. Folgende E-Mail-Adresse ist für Kritik, Anregungen oder Verbesserungsvorschläge verfügbar:

support.psp@embedded.rwth-aachen.de

3 Heap / Schedulingstrategien

In diesem Praktikumsversuch werden verschiedene Speicherarten des ATmega 644 erläutert und die Verwaltung des privaten Speichers für SPOS implementiert. Der hier eingeführte private dynamische Speicher stellt in Ergänzung zum Stack eine zusätzliche Möglichkeit für Prozesse dar, zur Laufzeit Speicher allozieren bzw. deallozieren zu können.

3.1 Versuchsinhalte

Nach der Durchführung des vorangegangenen Versuchs unterstützt SPOS die (implizite) Allokation von Stackspeicher für Anwendungsprogramme. Dieser Stackspeicher wird z. B. für lokale Variablen und Funktionsaufrufe benötigt. Die meisten Betriebssysteme bieten dem Entwickler die Möglichkeit, zusätzlichen eigenen Speicher explizit zu allozieren. Dieser explizit allozierte Speicher wird in Form eines sogenannten *Heaps* verwaltet, der im Rahmen dieses Versuchs implementiert wird. Bei Programmiersprachen ohne automatische Garbage Collection muss der Entwickler selbst auf die Freigabe des von ihm angeforderten Speichers achten.
In den nächsten Abschnitten werden zunächst die theoretischen Grundlagen und die Besonderheiten des ATmega 644 erklärt. In den Abschnitten 3.3.1 und 3.3.2 werden unterschiedliche Speicherarten des ATmega 644 vorgestellt und der Unterschied eines dynamischen Speichers zum Stack gezeigt. Kapitel 3.4 beschreibt die einzelnen Aufgabenstellungen und die konkrete Implementierung der vorgestellten Funktionalitäten.

3.2 Lernziel

Das Lernziel dieses Versuchs ist das Verständnis der folgenden Zusammenhänge:

- Verschiedene Speicherarten des ATmega 644
- Dynamischer Speicher (Heap)

3.3 Grundlagen

In diesem Abschnitt werden die Grundlagen der dynamischen Speicherverwaltung eingeführt. Hierzu werden die verschiedenen Speicherarten des ATmega 644 vorgestellt, sowie eine Einführung in die dynamische Allokation und Freigabe von Speicher gegeben.

3.3.1 Speicher des ATmega 644

Aktuelle Mikrocontroller verfügen über verschiedene Arten von Speicher, die sich im internen Aufbau und der typischen Verwendung unterscheiden. Der ATmega 644 bietet seinem Benutzer mehrere Speicherarten, die für unterschiedliche Zwecke genutzt werden können: 64 kByte Flash-Speicher und 4 kByte SRAM. Die Merkmale der einzelnen Speicherarten werden in den folgenden Abschnitten genauer erläutert.

Flash-ROM

Flash-ROMs sind nichtflüchtige Datenspeicher, die elektrisch lösch- und beschreibbar sind. Sie können beliebig oft ausgelesen, aber nicht beliebig oft beschrieben werden. Im Fall des ATmega 644 liegt die Grenze bei etwa 10.000 Schreib- und Löschzyklen.
In einem Flash-ROM können nicht einzelne Bytes gelöscht und neu beschrieben werden, sondern nur ganze Sektoren (Blöcke). Der Flash-ROM des ATmega 644 dient als Programmspeicher, in dem die kompilierten Programme abgelegt werden. Die Programmdaten werden z. B. über die JTAG-Schnittstelle oder mithilfe eines Bootloaders in den Flash-Speicher geschrieben und während der Programmausführung daraus gelesen und ausgeführt. Der Flash-ROM kann zusätzlich zur Speicherung von Daten, die sich während der Programmausführung nicht ändern, etwa Strings für das LCD, verwendet werden.

[S]RAM

Der [S]RAM *([Static] Random Access Memory)* ist flüchtiger Speicher mit *wahlfreiem Zugriff*. Wahlfreier Zugriff bedeutet, dass auf die einzelnen Bytes in konstanter Zeit lesend und schreibend zugegriffen werden kann. Im Gegensatz dazu hängt die Zugriffszeit bei blockorientiertem Zugriff von der Position des Bytes im Block ab. Da es sich bei RAM um einen flüchtigen Speicher handelt, gehen die Daten nach dem Ausschalten des Mikrocontrollers verloren. Darüber hinaus kann er sehr schnell und beliebig oft ausgelesen und beschrieben werden. Aus diesem Grund eignet sich dieser besonders zur Speicherung von Daten, auf die häufig zugegriffen wird oder die zur Laufzeit des Programms verändert werden.

3.3.2 Dynamischer Speicher

Der dynamische Speicher, auch als *Heap* bezeichnet, ist ein Speicherbereich im RAM, aus dem Prozesse zur Laufzeit zusammenhängende Blöcke beliebiger Größe für eigene Daten anfordern können. Ein wichtiger Unterschied zum Stack besteht darin, dass sich

die Datenblöcke aller Prozesse in einem gemeinsamen Speicherbereich befinden. Der zur Verfügung stehende Speicherplatz wird also je nach Bedarf auf die einzelnen Prozesse verteilt. Prozesse mit einem hohen Speicherbedarf können somit einen großen Anteil des insgesamt verfügbaren dynamischen Speichers nutzen, während andere Prozesse nur wenig oder gar keinen Speicher benötigen.
Da Prozesse Speicherbereiche in unterschiedlichen Größen anfordern können, muss das Betriebssystem über eine Speicherverwaltung verfügen, die einzelne Allokationen und Freigaben überwacht und verarbeitet. Die Freigabe kann sowohl manuell, als auch mit Hilfe einer automatischen Speicherbereinigung (engl. *Garbage Collection*) erfolgen. Ohne eine automatische Bereinigung kann es zu ungenutzten, besitzerlosen Speicherblöcken kommen, die von einem Prozess nicht explizit freigegeben wurden. Diese Speicherblöcke können nicht wiederverwendet werden und reduzieren somit die Größe des nutzbaren Speichers, was als *Memory Leak* bezeichnet wird.

Allokationsstrategien

Der vom Betriebssystem zu verwaltende dynamische Speicher kann als großes Array von Bytes aufgefasst werden. Fragt ein Benutzer dynamischen Speicher an, muss das Betriebssystem in diesem Array einen Abschnitt finden, der unbelegt ist und mindestens die angeforderte Größe hat. Da nach mehreren Allokationen und Freigaben der Speicher typischerweise in mehrere unterschiedlich große Stücke und Lücken unterteilt ist (*Fragmentierung*), ist das Auswählen eines günstigen Abschnitts nicht trivial. Abbildung 3.1 zeigt exemplarisch einen 25 Byte großen dynamischen Speicher, in dem 9 Byte in Form von 6 Abschnitten belegt (gelb) und noch 16 Byte frei (weiß) sind.

Abbildung 3.1: Fragmentierter Speicher mit 25 Byte Größe

Die einfachste Allokationsstrategie *First-Fit* beginnt die Suche nach einem passenden Abschnitt bei der ersten Adresse des dynamischen Speichers. Dabei wird der erste freie Speicherbereich ausgewählt, der größer oder gleich der angeforderten Größe ist.
Abbildung 3.2 zeigt den Speicher aus Abbildung 3.1 nach dem Anfordern von zwei Byte (blau).

Abbildung 3.2: Allokation von zwei Byte mit First-Fit

Lernerfolgsfragen

- Welche von den im vorherigen Abschnitt vorgestellten Speicherarten wird für den dynamischen Speicher verwendet? Warum?
- Wie unterscheiden sich Heap und Stack?
- Was passiert mit einem Speicherblock, der von einem Prozess angefordert und nicht freigegeben wurde?
- Was versteht man unter Fragmentierung? Wie entsteht Fragmentierung?
- Wie funktioniert die First-Fit Allokationsstrategie?

3.4 Hausaufgaben

Achtung

In der Datei `defines.h` im Codegerüst finden Sie das `define VERSUCH`. Passen Sie dieses auf die aktuelle Versuchsnummer an.

Implementieren Sie die in den nächsten Abschnitten beschriebenen Funktionalitäten. Halten Sie sich an die hier verwendeten Namen und Bezeichnungen für Variablen, Funktionen und Definitionen.
Lösen Sie alle hier vorgestellten Aufgaben zu Hause mithilfe von Microchip Studio 7 und schicken Sie die dabei erstellte und funktionsfähige Implementierung über Moodle ein. Ihre Abgabe soll dabei die `.atsln`-Datei, das Makefile, sowie den Unterordner mit den `.c/.h`-Dateien inklusive der `.xml/.cproj`-Dateien enthalten. Beachten Sie bei der Bearbeitung der Aufgaben die angegebenen Hinweise zur Implementierung! Ihr Code muss ohne Fehler und ohne Warnungen kompilieren sowie die Testtasks mit aktivierten Compileroptimierungen bestehen. Wie Sie die Optimierungen einschalten ist dem begleitenden Dokument in Abschnitt *6.3.2 Probleme bei der Speicherüberwachung* zu entnehmen.

Achtung

Verwenden Sie zur Prüfung auf Warnungen den Befehl „Rebuild Solution“ im „Build“-Menü des Microchip Studio 7. Die übrigen in der grafischen Oberfläche angezeigten Buttons führen nur ein inkrementelles Kompilieren aus, d.h. es werden nur geänderte Dateien neu kompiliert. Warnungen und Fehlermeldungen in unveränderten Dateien werden dabei nicht ausgegeben.

3.4.1 Implementierung der Schedulingstrategien

Erweitern Sie den in Versuch 2 entwickelten Scheduler so, dass alle in Versuch 2 vorgestellten Schedulingstrategien genutzt werden können. Dies betrifft die Strategien *RoundRobin*, *InactiveAging* und *RunToCompletion* und deren Verwaltungsdaten. Bei jedem Aufruf des Schedulers muss anhand der gewählten Schedulingstrategie entschieden werden, ob der aktive Prozess fortgesetzt wird oder ein anderer Prozess seine Ausführung beginnen darf.

Verwaltungsdaten Zur Implementierung der Strategien *RoundRobin* und *InactiveAging* müssen das Alter eines Prozesses und der Wert der aktuellen Zeitscheibe gespeichert werden. Legen Sie hierzu in der Datei `os_scheduling_strategies.h` eine Struktur `struct SchedulingInformation` an, welche den Wert `timeSlice` mit geeignetem Wertebereich sowie ein Array `age` mit Werten vom Typ `Age` enthalten soll. Das Array muss Platz für genau `MAX_NUMBER_OF_PROCESSES` Elemente bieten. Benennen Sie die Struktur mittels `typedef` als `SchedulingInformation` und legen Sie eine globale Variable `schedulingInfo` vom Typ `SchedulingInformation` in `os_scheduling_strategies.c` an.

Initialisierung Implementieren Sie die folgenden Funktionen in der Datei `os_scheduling_strategies.c`, um `schedulingInfo` für *RoundRobin* und *InactiveAging* korrekt zu initialisieren.

`void os_resetProcessSchedulingInformation(ProcessID id):`
Diese Funktion soll das Alter des Prozesses mit Prozess-ID `id` auf `0` setzen. Fügen Sie an geeigneter Stelle in `os_exec` einen Aufruf dieser Funktion ein. Dadurch soll sichergestellt werden, dass ein neuer Prozess, dem der Index eines vorigen Prozesses zugewiesen wurde, nicht auch dessen Alter übernimmt.

`void os_resetSchedulingInformation(SchedulingStrategy strategy):`
Diese Funktion soll die zu der Strategie `strategy` gespeicherten Daten zurücksetzen. Für *RoundRobin* bedeutet dies, die Zeitscheibe auf die Priorität des derzeit ausgewählten

Prozesses zu setzen. Im Fall von *InactiveAging* soll die Routine das Alter aller Prozesse auf `0` setzen. Rufen Sie diese Funktion in `os_setSchedulingStrategy` mit der neu gesetzten Strategie als Parameter auf.

Hinweise zur Implementierung Legen Sie neue Schedulingstrategien ausschließlich in der Datei `os_scheduling_strategies.c` und der zugehörigen Headerdatei an. Bei der Implementierung der Schedulingstrategien können Sie davon ausgehen, dass der Leerlaufprozess mit der Prozess-ID `0` immer aktiv ist und ausgewählt werden kann, falls kein anderer Prozess existiert oder verfügbar ist. Stellen Sie ebenfalls sicher, dass der Leerlaufprozess niemals ausgewählt wird, wenn Anwendungsprozesse auf ihre Ausführung warten. Da in diesem Versuch Prozesse terminieren können, muss bezüglich der *RunToCompletion* Strategie beachtet werden, dass nach der Terminierung eines aktiven Prozesses ein neuer Prozess – beispielsweise mithilfe der *Even*-Strategie – ausgewählt wird. Für *RoundRobin* verringern Sie die Zeitscheibe des aktiven Prozesses bei jedem Scheduleraufruf um Eins. Die Zeitscheibe gilt als abgelaufen, wenn sie den Wert `0` erreicht.

3.4.2 Dynamischer Speicher in SPOS

Der SRAM des ATmega 644 soll im Rahmen dieses Praktikums in zwei große Bereiche aufgeteilt werden: Den dynamischen Speicher (Heap) und den Stack, dessen Verwaltung bereits in Versuch 2 implementiert wurde. Das Ablegen von privaten Daten (z. B. lokaler Variablen) auf dem Stack wird für alle Prozesse vollständig vom Compiler organisiert. Der dynamische Speicher soll in Ergänzung dazu den Anwendungsprozessen eine zusätzliche Möglichkeit zur Verfügung stellen, ihren Speicherbedarf zu decken. Die dafür verwendeten Funktionen, `os_malloc` und `os_free`, werden an die entsprechenden `stdlib.h` Funktionen `malloc` und `free` angelehnt. Abbildung 3.3 zeigt eine erweiterte Aufteilung des SRAM aus Versuch 2.

Die Allokationstabelle

Um die Zuordnung von allozierten Speicherbereichen zu Prozessen zu ermöglichen, soll ein Teil des Speichers für eine *Allokationstabelle* (engl. *Heapmap*) reserviert werden. Diese Tabelle besteht aus Einträgen zu je vier Bit (sog. Nibbles oder Halbbytes), die den Besitzer der zu diesem Eintrag gehörenden Nutzdaten angeben. Ein Byte in der Tabelle verwaltet folglich zwei Byte Nutzdaten. Daraus resultiert, dass der Nutzdatenbereich genau doppelt so groß ist wie die Allokationstabelle.
Jedem Byte Nutzdaten können durch die vier Bit $2^4 = 16$ verschiedene Zustände in der Allokationstabelle zugeordnet werden. Die Zuordnung der Nibbles der Allokationstabelle zu den Bytes der Nutzdaten ordnet dem höherwertigen Nibble des Eintrags in der Allokationstabelle das Byte mit der niedrigeren Adresse in den Nutzdaten zu. Diese Konvention erleichtert die Verständlichkeit und hilft, einzelne Speicherblöcke zu identifizieren (vgl. Abbildung 3.4).

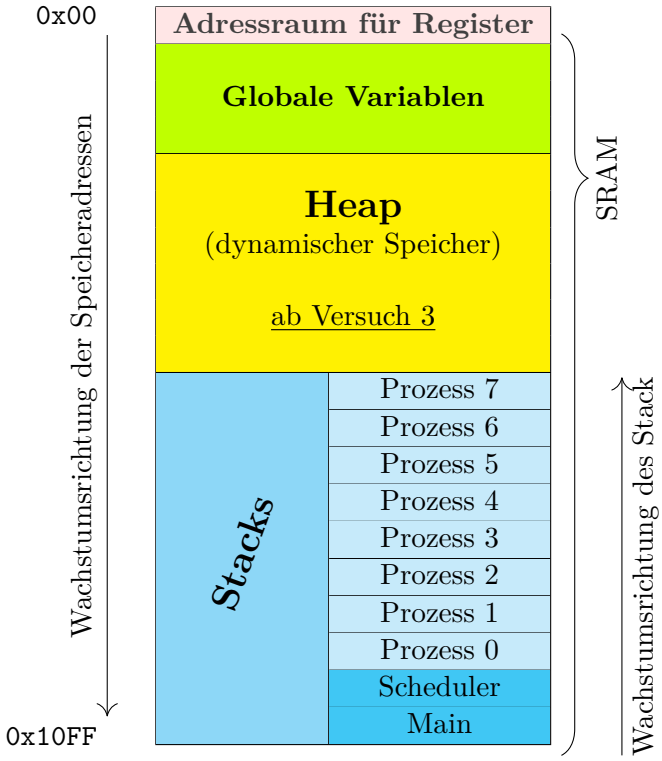

Abbildung 3.3: Speicherbelegung (SRAM) des ATmega 644

Für die Zuordnung von Speicherblöcken zu Prozessen wird ein Protokoll verwendet, welches anhand der Werte der Allokationstabelle jedem Speicherblock eindeutig einen Prozess zuweist. Umgekehrt müssen alle von einem Prozess allozierten Speicherblöcke anhand der Allokationstabelle identifizierbar sein. Die Anzahl der Prozesse ist auf 7+1 (7 Anwendungs- und der Leerlaufprozess) limitiert, wobei nur die 7 Anwendungsprozesse dynamischen Speicher anfordern dürfen.

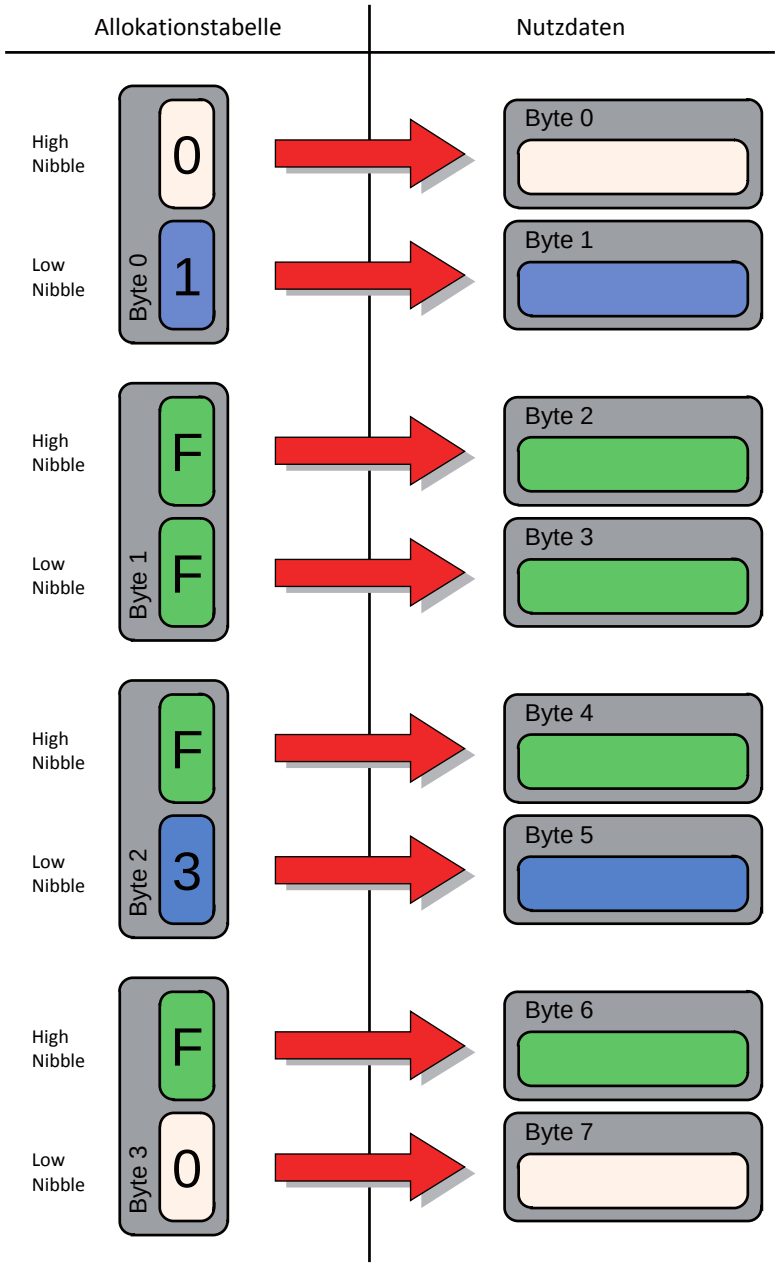

Abbildung 3.4: Zusammenhang zwischen Allokationstabelle und Nutzdaten

Das Beispiel in Abbildung 3.4 zeigt die Verwendung der Allokationstabelle. Eine 0 in einem Nibble der Allokationstabelle bezeichnet eine freie Speicherstelle und eine Zahl $i \in \{1, ..., 7\}$ eine Speicherstelle, welche von Prozess i angefordert wurde. Nachfolgende Nibbles mit dem Wert F_h (hexadezimal 15) symbolisieren das Fortlaufen der Speicherstelle. In der Allokationstabelle in Abbildung 3.4 ist das erste Byte der Nutzdaten frei, da an der entsprechenden Position in der Allokationstabelle eine 0 steht. Die Werte der

Allokationstabelle für die nächsten vier Byte Nutzdaten sind $1FFF_h$, was in diesem Protokoll bedeutet, dass diese vier Byte dem Prozess 1 zugeordnet sind. Analog wurden die nachfolgenden zwei Byte vom dritten Prozess reserviert (in der Allokationstabelle mit $3F_h$ markiert). Das letzte dargestellte Byte ist frei (markiert durch eine 0).

Hinweis

Machen Sie sich mit der Funktionsweise des vorgestellten Protokolls vertraut, bevor Sie es implementieren.

Lernerfolgsfragen

- Wieso ist es sinnvoll festzuhalten, welcher Prozess einen bestimmten Speicherbereich belegt?
- Welches Verhalten sieht das Protokoll vor, falls derselbe Prozess zwei Speicherbereiche der Länge 1 anfordert und diese direkt nebeneinander positioniert werden? Kann es bei der Freigabe zur Verwechslung mit einem Speicherblock der Länge 2 kommen?
- Sieht das Protokoll vor, dass ein allozierter Speicherblock freigegeben werden kann, wenn die übergebene Adresse nicht auf das erste Byte dieses Blocks zeigt?

Wie in Abschnitt 3.3.1 erläutert wurde, stehen dem ATmega 644 insgesamt 4096 Byte Arbeitsspeicher (SRAM) zur Verfügung. Der Mikrocontroller legt verschiedene Daten (z. B. globale Variablen) automatisch am Anfang des SRAM ab. Daher muss zum Anfang des Heaps ein Sicherheitsabstand eingehalten werden, um ein Überschreiben dieser Daten zu verhindern. Teilen Sie den restlichen verfügbaren Speicher im Verhältnis 1:2 in die Allokationstabelle und den nutzbaren Datenbereich auf. Der bereits im SRAM verwendete Speicher kann nach der Kompilierung des Projekts in Microchip Studio 7ausgelesen werden. Abbildung 3.5 zeigt beispielhaft die Ausgabe der relevanten Informationen im *Output*-Fenster. In diesem Fall reicht ein Abstand von 200 Byte aus (179 Byte + zusätzlicher Sicherheitsabstand). Abhängig von Ihrer Implementierung muss dieser Wert angepasst werden.

```
Program Memory Usage: 18808 bytes   28,7 % Full
Data Memory Usage: 179 bytes   4,4 % Full
```

Abbildung 3.5: Informationen im Output-Fenster von Microchip Studio 7nach der Kompilierung

Achtung

- Berücksichtigen Sie, dass Sie den Abstand ggf. in den nächsten Versuchen anpassen müssen, wenn Sie das Betriebssystem erweitern und z. B. neue globale Variablen anlegen.
- Bedenken Sie, dass der für Sie nutzbare Datenbereich des SRAM bei Adresse 0x100 beginnt, nicht bei 0!
- Achten Sie darauf, wie im *Begleitenden Dokument zum Praktikum Systemprogrammierung* beschrieben, sämtliche Zeichenketten im Flash-Speicher abzulegen.

Der vom Mikrocontroller verwendete Speicher am Anfang des Heaps kann zur Laufzeit ermittelt werden, da dieser nach dem Kompilieren des Projekts feststeht. Die erste freie Speicherstelle des Heaps wird durch das Symbol `__heap_start` markiert und kann im Programmablauf ausgelesen werden. Da dieses Symbol erst nach dem Linken verfügbar ist, muss es an geeigneter Stelle als externe, konstante Variable deklariert werden, um es auslesen zu können (siehe Listing 3.1).

```
extern uint8_t const __heap_start;
```

Listing 3.1: Deklarierung einer Variable, die am Anfang des Heaps liegt

Diese Variable liegt nach dem Kompilieren am Anfang des dynamischen Speichers. Um zu erfahren, ob der Sicherheitsabstand ausreichend groß gewählt wurde, vergleichen Sie die **Speicheradresse** dieser Variable mit Ihrem gewählten Anfang des Heaps. Der Wert der Variable selbst ist unbedeutend. Mehr Informationen zum Schlüsselwort `extern` finden Sie in Kapitel *5.1.5* im *Begleitenden Dokument zum Praktikum Systemprogrammierung*. Erstellen Sie das Define `HEAPOFFSET` in der Datei `defines.h`, um den von Ihnen gewählten Sicherheitsabstand zu setzen. Den Wert des Define können Sie gemäß der obigen Rechnung setzen. Implementieren Sie dann an geeigneter Stelle eine Sicherheitsüberprüfung und benutzen Sie dabei das Symbol `__heap_start`. Im Fehlerfall soll eine Fehlermeldung auf dem LCD erscheinen. Bedenken Sie dabei, dass das LCD bereits initialisiert sein muss und die Überprüfung daher nicht als erster Schritt beim Start des Betriebssystems erfolgen darf.

Lernerfolgsfragen

- Was kann geschehen, wenn Sie den Sicherheitsabstand zum Bereich der globalen Variablen zu gering bzw. zu groß kalkulieren?
- Wieso kann die Größe der globalen Variablen nicht durch ein Define zur Compilezeit festgelegt werden, sondern steht erst zur Linkzeit fest?
- Ab welcher Adresse beginnend wird der SRAM adressiert? Was liegt davor?

Aufteilung in Schichten

Die zu implementierende Speicherverwaltung soll soweit abstrakt sein, dass der Zugriff auf das eigentliche Speichermedium – in diesem Versuch auf den SRAM – für die Anwendungsprozesse transparent bleibt und das Medium beliebig ausgetauscht werden kann. Zu diesem Zweck wird die Speicherverwaltung in zwei Schichten unterteilt: Die untere Schicht stellt einen Treiber für direkte Zugriffe auf das Speichermedium bereit, die obere Schicht bearbeitet die Speicheranfragen der Anwendungsprozesse auf Basis des Treibers in einem vordefinierten Heap auf dem Speicher. Dieses Treibermodell wird eingeführt, da in späteren Versuchen andere Speichermedien als der SRAM genutzt werden und so die Funktionalitäten der oberen Schicht nicht für jedes Speichermedium erneut implementiert werden müssen.
Speichertreiber sind Instanzen der unteren Schicht und enthalten Informationen über das verwendete Speichermedium. Dazu gehören unter anderem die Größe des Speichermediums, verfügbarer Adressraum und Funktionen zur Initialisierung des Mediums sowie zum direkten Schreiben und Lesen von Bytes. Für jedes benutzte Speichermedium muss ein eigener Treiber angelegt werden. Diese Treiber werden als Teil des Heaptreibers als Parameter in die Funktionen der oberen Schicht übergeben und ermöglichen den Zugriff auf den Heap des jeweiligen Mediums. Beachten Sie, dass über die Speichertreiber der Zugriff auf das gesamte Speichermedium möglich sein muss, nicht nur auf den Bereich, der für den Heap reserviert wurde.
Abbildung 3.6 verdeutlicht diese Aufteilung. Anwendungsprogramme greifen mit Hilfe der *Memory Funktionen* auf die Treiberstruktur des *Heaps* zu, z. B. um Speicher anzufordern oder freizugeben. Weiterhin können die Anwendungsprogramme auf den Speicher schreiben bzw. vom ihm lesen, indem sie die entsprechenden Zugriffsfunktionen des dem Heap assoziierten *MemDrivers* aufrufen.

Untere Schicht: Treiber

Der Speichertreiber wird als eine Struktur realisiert, die alle charakteristischen Werte des Mediums sowie Funktionszeiger auf die nötigen Zugriffsroutinen enthält. Für jedes

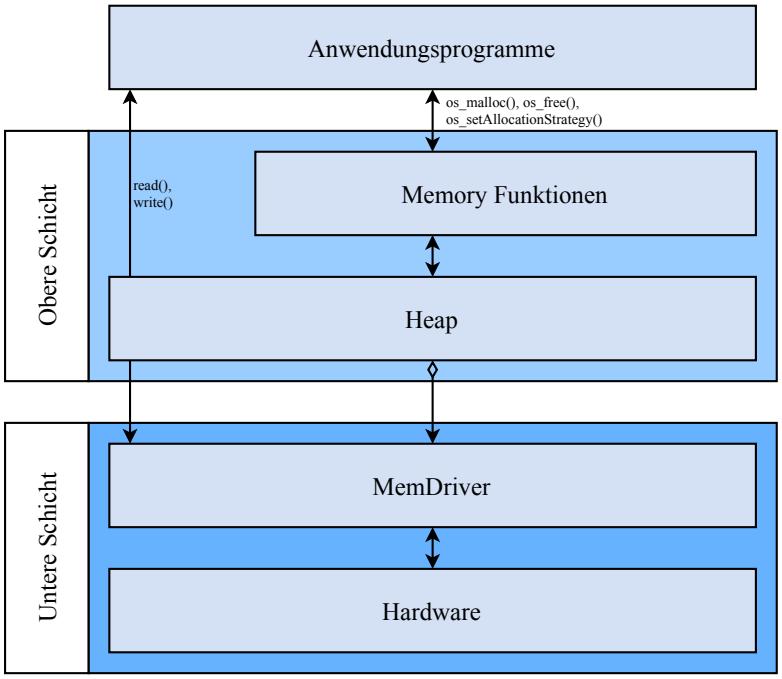

Abbildung 3.6: Zusammenspiel zwischen oberer und unterer Schicht.

konkrete Speichermedium muss eine Instanz dieser Struktur angelegt werden.

Hinweis

Für eine bessere Lesbarkeit des erzeugten Codes ist es hilfreich, die Typen der nötigen Funktionszeiger in der Datei `os_mem_drivers.h` festzulegen.

Üblicherweise erfolgt der Zugriff auf die Daten im SRAM über `uint8_t` Zeiger. Ein solcher Zeiger enthält eine 16-Bit lange Adresse und wird für jede Lese-/Schreiboperation dereferenziert. Im Rahmen des Praktikums müssen jedoch alle Operationen auf dem Speicher indirekt über den jeweiligen Treiber erfolgen. Daher werden statt Zeigern Variablen des Typs `uint16_t` verwendet, wodurch keine direkte Dereferenzierung mehr möglich ist. Für eine bessere Lesbarkeit wird der neue Datentyp `MemAddr` angelegt, der mit Hilfe des Schlüsselworts `typedef` als `uint16_t` definiert ist.
Analog wird der Typ `MemValue` für einzelne durch `MemAddr` referenzierte *Speicheratome* verwendet. Als Atome werden in diesem Kontext einzelne Bytes bezeichnet, da diese nicht weiter auf verschiedene Benutzer (Prozesse) verteilt werden können. Folglich hat `MemValue` die Größe eines Bytes, kann also als `uint8_t` definiert werden.
Legen Sie die Dateien `os_mem_drivers.c` und `.h` an, welche die untere Schicht des Speichertreibers beinhalten. Bedenken Sie, dass die Headerdatei `(.h)` in der Codedatei

`(.c)` inkludiert werden muss, um dortige Definitionen nutzen zu können. Um auftretende Mehrfachinkludierung zu vermeiden, empfiehlt es sich die `#ifndef`-Direktive zu verwenden. Weitere Informationen hierzu finden Sie im Kapitel *5.1.2 Der Präprozessor* des *Begleitenden Dokuments zum Praktikum Systemprogrammierung*.
Legen Sie in der Datei `os_mem_drivers.h` die Struktur `MemDriver` des allgemeinen Speichertreibers an. Definieren Sie den Typ `MemDriver` als eine Kurzform für `struct MemDriver`. Die angelegte Struktur kann Konstanten für alle charakteristischen Werte des benutzten Speichermediums enthalten und muss Funktionszeiger für die folgenden Zugriffsroutinen bereitstellen:

- Eine Routine `void init(void)` zur Initialisierung des Speichermediums
- Eine Leseroutine `MemValue read(MemAddr addr)`, die den Wert an der übergebenen Adresse `addr` ausliest und zurückgibt
- Eine Routine `void write(MemAddr addr, MemValue value)` zum Schreiben auf das Medium. Sie speichert den Wert `value` an die Adresse `addr`

Die Interaktion der verschiedenen Komponenten können Sie sich zudem anhand der Doxygen-Dokumentation deutlich machen. Beachten Sie, dass diese Dokumentation zusätzliche Hilfsfunktionen enthalten kann, die nicht zwingend Teil der Hausaufgaben sind.

Treiber für den internen SRAM

Implementieren Sie in der Datei `os_mem_drivers.c` die im letzten Abschnitt angegebenen Funktionen für den direkten Zugriff auf den internen SRAM. Achten Sie dabei auf die korrekte Typisierung der Parameter und Rückgabewerte. Dort legen Sie ebenfalls die eigentliche Treiberinstanz an.
Definieren Sie eine Instanz des Speichertreibers mit dem Bezeichner `intSRAM__` und dem Typ `MemDriver`. Ein `define` mit dem Namen `intSRAM` soll die Adresse und somit einen Zeiger auf die Instanz `intSRAM__` bereitstellen, um den Speichertreiber anderen Funktionen übergeben zu können. Initialisieren Sie die Instanz des Speichertreibers mit den zuvor in `os_mem_drivers.c` implementierten Routinen. Sofern es für die Implementation sinnvoll ist, können weitere charakteristische Konstanten des Speichermediums hinzugefügt werden, wie z.B. die erste verfügbare Adresse sowie die Größe des Speichermediums. Rufen Sie die Funktion `init` der Treiberstruktur an einer geeigneten Stelle in Ihrem Code auf, um das Speichermedium zu initialisieren. Dies ist insbesondere für spätere Versuche von Bedeutung, da der interne SRAM nicht initialisiert werden muss.

Hinweis

- Attribute einer Treiberstruktur, die mit dem Schlüsselwort `const` deklariert wurden, können zur Laufzeit nicht geändert werden. Ein Beispiel der korrekten Vorgehensweise kann in Abschnitt *5.2.3 Umgang mit Daten* des *Begleitenden Dokuments zum Praktikum Systemprogrammierung* gefunden werden.
- Da der Zugriff auf den internen SRAM nur über den Treiber ermöglicht werden soll, werden die Zugriffsfunktionen privat deklariert. Weitere Informationen zur privaten Deklaration von Funktionen finden Sie in Kapitel *5.1.5* im *Begleitenden Dokument zum Praktikum Systemprogrammierung.*
- Beachten Sie bei der Implementierung der Treiberstruktur und der Adresslogik folgende Abschnitte des *Begleitenden Dokuments zum Praktikum Systemprogrammierung*: *Lesbares Bitshifting*, *Typecasts* und *Eigene Datentypen erstellen.*
- Wie Sie die `MemDriver` Struktur genau aufbauen, bleibt Ihnen überlassen. Allerdings müssen Anwendungsprogramme direkt auf die Schreib- und Lesefunktionen zugreifen können, sodass der Zeiger auf die Schreibfunktion zwingend den Bezeichner `write` haben muss. Analog muss der Bezeichner für die Lesefunktion `read` lauten.

Obere Schicht: Heaptreiber

Anders als der Treiber für das Speichermedium selbst, stellt der Heaptreiber die überliegende Speicherverwaltung des Speichermediums dar. Die Speicherverwaltung kann sich hierbei auf Teile des Speichers beschränken, welcher durch den Treiber des Speichermediums bereitgestellt wird. Somit können mehrere unterschiedliche Speicherverwaltungen auf einem gemeinsamen Speichermedium realisiert werden. Das in diesem Versuch vorgestellte Konzept zur Nutzung des internen SRAM des ATmega 644 stellt durch die Unterteilung in Stack und Heap genau so ein Beispiel dar.

Wie bereits in Abschnitt 3.4.2 erwähnt wurde, teilt sich der Heap in einen Map- (Allokationstabelle) und einen Use-Bereich (Nutzdaten) auf. Während der Treiber des Speichermediums lediglich die Information besitzt, bei welcher Adresse das Speichermedium beginnt und wie groß es ist, besitzt der Heaptreiber Informationen über die Startadresse und Größe des Map- und Use-Bereichs der Heap-Speicherverwaltung. Der eigentliche Speicherzugriff auf das Medium erfolgt mit Hilfe der unteren Treiberschicht, sodass diese in den Heaptreiber eingebunden werden muss.

Legen Sie die Datei `os_memheap_drivers.h` in Ihrem Projekt an und inkludieren Sie

diese in der zugehörigen Datei `os_memheap_drivers.c`. Vermeiden Sie hier ebenfalls Mehrfachinkludierung.
Das tatsächliche Auswählen eines Speicherbereichs soll analog zu den Schedulingstrategien austauschbar sein und von Allokationsstrategien vorgenommen werden. Implementieren Sie für diesen Versuch die *First-Fit* Strategie gemäß Abschnitt 3.3.2. Legen Sie für die Implementierung der Allokationsstrategien die Datei `os_memory_strategies.c` mit entsprechendem Header an, da im Verlauf des Praktikums weitere Allokationsstrategien eingeführt werden. Legen Sie außerdem in der Datei `os_memheap_drivers.h` ein `enum` an, welches Sie per `typedef` als `AllocStrategy` benennen, um die Lesbarkeit bei der Verwendung der Allokationsstrategien zu erhöhen. Benutzen Sie die folgenden Bezeichner: `OS_MEM_FIRST, OS_MEM_NEXT, OS_MEM_BEST` und `OS_MEM_WORST`.
Definieren Sie im Header die Struktur `Heap` des Heaptreibers sowie die Kurzform `Heap` stellvertretend für `struct Heap`. Fügen Sie der Struktur die folgenden Attribute hinzu:

- Einen Zeiger auf den Speichertreiber, welcher dem Heap assoziiert ist
- Die charakteristischen Eigenschaften (Startadresse, Größe) des Map- und Use-Bereichs
- Die aktuell verwendete Allokationsstrategie (siehe Abschnitt 3.4.2) auf dem Heap des Speichermediums
- Einen Zeiger auf den Namen für diesen Heap

Hinweis

Aufgrund der Interaktion mit dem Taskmanager müssen für einige der Attribute dieses Datentyps bestimmte Namen gewählt werden. Benennen Sie das Attribut mit dem Zeiger auf den Speichertreiber des Mediums als `driver` und das Attribut mit dem Namen des Heaps als `name`.

Erstellen des Heaptreibers intHeap

Erstellen Sie die Datei `os_memheap_drivers.c` und legen Sie dort den Heaptreiber `intHeap__` für den Speichertreiber `intSRAM` an. Definieren Sie eine Instanz eines Heaptreibers mit dem Namen `intHeap__` und dem Typ `Heap`, welche den Heap des internen SRAM verwaltet. Legen Sie die im vorherigen Abschnitt erläuterten Attribute des Heaps fest und weisen Sie diese der erstellten Instanz zu. Legen Sie auch für den Heaptreiber ein `define intHeap` an, welches einen Zeiger auf `intHeap__` bereitstellt.
Zur Initialisierung des Heaps soll die Funktion `os_initHeaps()` verwendet werden. Bei Aufruf dieser Funktion müssen alle Nibbles des Map-Bereichs aller vorhandenen Heaptreiber mit dem im Protokoll spezifizierten Zustand für einen freien Speicherblock

überschrieben werden. In dem hier vorgestellten Protokoll entspricht dies dem Wert 0_h. Rufen Sie die Funktion `os_initHeaps` an geeigneter Stelle in Ihrem Betriebssystem auf, um den Heap des jeweiligen Speichermediums zu initialisieren. Halten Sie diese Funktion so allgemein wie möglich, da im weiteren Verlauf des Praktikums zusätzliche Treiber für Speichermedien behandelt werden und für diese ebenfalls Heaptreiber erstellt werden müssen. Die Initialisierung der verwalteten Speichermedien kann ebenfalls in dieser Funktion erfolgen, wobei dies erst in späteren Versuchen relevant sein wird.

Darüber hinaus sollen noch zwei weitere Funktionen implementiert werden. Die Funktion `os_getHeapListLength(void)` soll die Anzahl an existierenden Heaps zurückgeben. Da in diesem Versuch nur der interne Heap zur Verfügung steht, ist der erwartete Rückgabewert `1`. Die Funktion `os_lookupHeap(uint8_t index)` soll zum übergebenen Heapindex den Zeiger auf den entsprechenden Heap zurückgeben. Hierbei wird dem `intHeap` der Index 0 zugeordnet.

Obere Schicht: Speicherverwaltung

Erweitern Sie Ihr Projekt um die Quelldatei `os_memory.c` und einem entsprechenden Header. Inkludieren Sie den Header in der Quelldatei.
Diese Dateien sollen dem Betriebssystem Funktionen zur Allokation und Freigabe des Speichers auf dem Heap zur Verfügung stellen. Implementieren Sie im Anschluss daran die im Folgenden beschriebenen Funktionen in Ihrem Projekt.

> **Achtung**
>
> Folgende Funktionen der Speicherverwaltung stellen kritische Bereiche dar. Sie dürfen somit nicht durch den Scheduler unterbrochen werden.

`MemAddr os_malloc(Heap* heap, uint16_t size):`
Wird diese Allokationsfunktion von einem Prozess aufgerufen, sollen auf dem übergebenen Heap so viele zusammenhängende Bytes alloziert werden, wie der Parameter `size` angibt. Die Allokationstabelle muss entsprechend des vorgestellten Protokolls zur Verwaltung von Speicherbereichen angepasst werden. Der Rückgabewert der Funktion soll die Adresse des ersten Bytes des allozierten Speicherbereichs im Use-Bereich des Heaps sein. Steht nicht genügend zusammenhängender Speicherplatz zur Verfügung, muss die Funktion den Wert 0 zurückgeben und keine Fehlermeldung ausgeben.

`void os_free(Heap* heap, MemAddr addr):`
Diese Funktion soll den Speicherbereich, auf den die übergebene Speicheradresse verweist, freigeben. Alle diesem Speicherbereich entsprechenden Nibbles in der Allokationstabelle müssen folglich auf den im Protokoll spezifizierten Zustand für einen freien

Speicherbereich gesetzt und somit als frei gekennzeichnet werden.
Beachten Sie, dass der von `os_malloc` reservierte Speicher privat ist und dem Prozess *gehört*, der den Speicher angefordert hat. Ein Speicherbereich darf nur von seinem Besitzer wieder freigegeben werden. Die Freigabe eines *fremden* Speicherbereichs darf nicht durchgeführt werden und muss eine Fehlermeldung hervorrufen.
Weiterhin kann die übergebene Speicheradresse innerhalb der Grenzen des Speicherbereichs verschoben sein und somit nicht auf das erste Byte des Blocks zeigen. Auch in diesem Fall muss der gesamte Speicherbereich korrekt freigegeben werden.

`size_t os_getMapSize(Heap const* heap),`
`size_t os_getUseSize(Heap const* heap),`
`MemAddr os_getMapStart(Heap const* heap),`
`MemAddr os_getUseStart(Heap const* heap):`
Diese Funktionen dienen dem Zugriff auf die von Ihnen spezifizierte Struktur `Heap`. Sie werden insbesondere vom Taskmanager verwendet und sollen jeweils die Grenzen der Allokationstabelle bzw. des Nutzdatenbereichs zurückliefern.
Hierbei ist mit `Size` die Größe des jeweiligen Bereichs (Allokationstabelle oder Nutzdaten) in Byte gemeint. Weiterhin steht `Start` für die erste Adresse des jeweiligen Bereichs.
Beachten Sie, dass die obengenannten Funktionen nicht Teil der von Ihnen modellierten Heapstruktur sind, sondern in der Datei `os_memory.c` definiert werden sollen.

`uint16_t os_getChunkSize(Heap const* heap, MemAddr addr):`
Liefert die Größe in Byte eines belegten Speicherabschnitts (*Chunk*) zurück. Beachten Sie auch hier, dass die übergebene Speicheradresse nicht zwangsläufig auf das erste Byte des Chunks zeigt. Falls der Speicherabschnitt frei ist, muss `0` zurückgegeben werden.

`AllocStrategy os_getAllocationStrategy(Heap const* heap),`
`void os_setAllocationStrategy(Heap *heap, AllocStrategy allocStrat):`
Diese Funktionen dienen dem Setzen und Abfragen der auf dem übergebenen Heap benutzten Allokationsstrategie. Sie sollen in der Datei `os_memory.c` definiert werden.

In der Doxygen-Dokumentation finden Sie einige weitere Funktionsprototypen mit detaillierten Erklärungen, die zur Umsetzung dieser Aufgabe hilfreich sind. Funktionen, die in diesen Versuchsunterlagen nicht explizit gefordert werden, müssen von Ihnen nicht implementiert werden. Mehr Informationen dazu werden im Kapitel *3.5 Verwendung der Doxygen-Dokumentation* erläutert.

Hinweis

- Alle Fehler und potentielle Gefahren sollen explizit ausgegeben werden.
- Bauen Sie an sinnvollen Stellen Parameterüberprüfungen ein. Dabei soll insbesondere auf die Einhaltung der Speichergrenzen geachtet werden. Die charakteristischen Werte können der übergebenen Treiberstruktur entnommen werden.
- In der Präsenzaufgabe zu diesem Versuch werden Sie mindestens eine weitere Allokationsstrategie implementieren und in Versuch 4 kommen weitere dazu. Beachten Sie bereits jetzt die Austauschbarkeit der Allokationsstrategien und treffen Sie die nötigen Vorbereitungen.
- Überprüfen Sie, ob Ihre Implementierung kritische Sektionen verwendet, wo dies benötigt wird.

Lernerfolgsfragen

- Auf Anwendungsebene werden Speicherbereiche anhand von Adressen innerhalb des Use-Bereichs des Heaps verwaltet. Ist es sinnvoll, in der Speicherverwaltung des Betriebssystems die Adressierung anhand von Map-Adressen vorzunehmen oder sollten diese erst für den direkten Zugriff auf die Heapmap berechnet werden?
- In welcher Form muss die Allokationstabelle initialisiert werden, damit die Speicherverwaltung korrekt funktioniert?
- Müssen für die korrekte Funktionsweise der Speicherverwaltung auch die Nutzdaten initialisiert werden? Warum (nicht)?
- Wie viel Speicher kann mit Ihrer Implementierung von `os_malloc` maximal angefordert werden?
- Welcher Datentyp wird in SPOS für Speicheradressen verwendet?
- Was ist die Aufgabe der einzelnen Schichten der Speicherverwaltung? Warum ist diese Aufteilung sinnvoll?
- Welche Prozesse dürfen einen Speicherbereich an der Adresse `addr` mit `os_free(..., addr)` freigeben?

3.4.3 Terminierung von Prozessen

Um den Verwaltungsaufwand möglichst gering zu halten, wurde bis jetzt angenommen, dass laufende Prozesse nicht terminieren können. Diese Annahme trifft im Allgemeinen nicht zu. Um Anwendungsprozessen die Terminierung zu ermöglichen, sollen diese in einer Funktion gekapselt werden.

Legen Sie in der Datei `os_scheduler.c` die Funktion `void os_dispatcher(void)` an, die die Kapselung der Anwendungsfunktion übernehmen soll. Der Ablauf der Programmausführung ändert sich nun wie folgt:

1. In der Funktion `os_exec` wird dem Funktionszeiger nicht mehr das eigentliche Programm direkt zugewiesen, stattdessen soll die Adresse der Funktion `os_dispatcher` übergeben werden. Somit dient `os_dispatcher` als Programmeinstiegspunkt.

2. In der Funktion `os_dispatcher` wird die Prozess-ID des aktiven Prozesses, sowie der zugehörige Funktionszeiger des Programmes bestimmt.

3. Aus `os_dispatcher` wird das Programm mittels des Funktionszeigers direkt aufgerufen. Der Aufruf des Programmes stellt sich somit wie ein gewöhnlicher Funktionsaufruf dar.

4. Sobald die Programmfunktion abgearbeitet und wieder verlassen wurde, kehrt die Ausführung automatisch in die Funktion `os_dispatcher` zurück.

5. Da die Funktion verlassen wurde, muss der aktive Prozess terminiert werden. Dazu wird der Prozesszustand des aktiven Prozesses auf `OS_PS_UNUSED` gesetzt und somit dessen Slot im Array `os_processes` freigegeben. Achten Sie außerdem darauf, dass der Scheduler terminierte Prozesse nicht wieder auswählbar macht, indem er sie auf `OS_PS_READY` setzt.

6. Die Funktion `os_dispatcher` fällt in eine Endlosschleife und wartet, bis der Scheduler den nächsten Prozess auswählt.

Aus Sicht der Funktion `os_exec` besteht jeder Prozess aus einer Instanz der Funktion `os_dispatcher`. Die Funktion `os_dispatcher` wird somit für jeden Prozess als Programm hinterlegt und realisiert intern den tatsächlichen Aufruf der Programmfunktion. Sie startet also keinen neuen Prozess, sondern führt innerhalb des aktiven Prozesses lediglich die entsprechende Programmfunktion aus. Der Prozess wird nach wie vor in der Funktion `os_exec` erzeugt.

Legen Sie die Funktion `bool os_kill(ProcessID pid)` in der Datei `os_scheduler.c` an. Diese Funktion dient dazu, aus dem Taskmanager heraus oder durch andere Prozesse das vorzeitige Terminieren eines Prozesses mit Prozess-ID `pid` zu ermöglichen.
Hierzu muss das Array `os_processes` aufgeräumt und der Slot des übergebenen Prozesses freigegeben werden. Beachten Sie hierbei, dass der Idle-Prozess nicht terminieren

darf. Machen Sie sich ebenfalls den Unterschied zwischen der eigenen Terminierung und der Terminierung eines anderen Prozesses deutlich. Terminiert ein Prozess sich selbst, so ist er nach dem Aufruf von `os_kill` nicht mehr verfügbar. Dies hat zur Folge, dass die Funktion nicht verlassen werden darf, sofern vorher kein anderer Prozess durch den Scheduler ausgewählt wurde. Achten Sie zusätzlich bei der Terminierung auf noch geöffnete kritische Bereiche, welche Sie behandeln müssen.
Die Funktion `os_kill` soll genau dann `true` zurückgeben, falls die übergebene Prozess-ID gültig ist und der entsprechende Prozess korrekt terminiert wurde.
Da sich der Inhalt dieser Funktion stark mit dem der Funktion `os_dispatcher` überschneidet, ist es sinnvoll, dass `os_dispatcher` in geeigneter Form `os_kill` aufruft.

Lernerfolgsfragen

- Welches Fehlverhalten kann auftreten, wenn der Zustand eines Prozesses auf `OS_PS_UNUSED` gesetzt wird und anschließend ein Aufruf der Scheduler-ISR erfolgt?
- Aus welchem Grund darf der Leerlaufprozess niemals beendet werden?
- Kann es zu Problemen kommen, wenn der Slot des aktiven Prozesses außerhalb bzw. innerhalb des Schedulers freigegeben wird?
- Müssen die Interrupts ein- oder ausgeschaltet sein, wenn die Endlosschleife des Dispatchers betreten wird? Warum (nicht)?
- Was geschieht, wenn ein Prozess `os_kill` mit seiner eigenen Prozess-ID aufruft?
- Was geschieht, wenn ein Prozess terminiert, der sich in einer kritischen Sektion befindet?

3.4.4 Zusammenfassung

Folgende Übersicht listet alle Typen, Funktionen und Aufgaben auf. Alle aufgelisteten Punkte müssen zur Teilnahme am Versuch bis zur Abgabefrist bearbeitet und hochgeladen werden. Diese Übersicht kann als Checkliste verwendet werden und ist daher mit Checkboxen versehen.

- ☐ `os_scheduler:`
  - ☐ Funktionen
    - ☐ `void os_dispatcher(void)`
    - ☐ `bool os_kill(ProcessID pid)`
- ☑ `os_scheduling_strategies:`
  - ☑ Datentypen
    - ☑ `struct SchedulingInformation` mit passendem `typedef`
  - ☑ Variablen
    - ☑ `SchedulingInformation schedulingInfo`
  - ☑ Funktionen
    - ☑ `void os_resetProcessSchedulingInformation(ProcessID id)`
    - ☑ `void os_resetSchedulingInformation(SchedulingStrategy strategy)`
    - ☑ `ProcessID os_Scheduler_RoundRobin(Process const processes[], ProcessID current)`
    - ☑ `ProcessID os_Scheduler_InactiveAging(Process const processes[], ProcessID current)`
    - ☑ `ProcessID os_Scheduler_RunToCompletion(Process const processes[], ProcessID current)`
- ☑ `os_mem_drivers:`
  - ☑ Datentypen
    - ☑ `MemAddr` (16 Bit Ganzzahl, vorzeichenlos)
    - ☑ `MemValue` (8 Bit Ganzzahl, vorzeichenlos)
    - ☑ `struct MemDriver` mit passendem `typedef`
  - ☑ Globale Variablen
    - ☑ `MemDriver intSRAM__` (sowie das `define intSRAM`)
  - ☑ Funktionen
    - ☑ Die Funktionen des `MemDriver`-Typs (init, read, write)
- ☑ `os_memheap_drivers:`

- ☑ Datentypen
  - ☑ `struct Heap` mit passendem `typedef`
  - ☑ `enum AllocStrategy` mit passendem `typedef`
- ☑ Globale Variablen
  - ☑ `Heap intHeap__` (sowie das `define intHeap`)
- ☑ Funktionen
  - ☑ `void  os_initHeaps(void)`
  - ☑ `uint8_t os_getHeapListLength(void)`
  - ☑ `Heap* os_lookupHeap(uint8_t index)`

☐ `os_memory:`

- ☐ Funktionen
  - ☐ `MemAddr os_malloc(Heap* heap, uint16_t size)`
  - ☐ `void os_free(Heap* heap, MemAddr addr)`
  - ☐ `size_t os_get{Map,Use}Size(Heap const* heap)`
  - ☐ `MemAddr os_get{Map,Use}Start(Heap const* heap)`
  - ☐ `uint16_t os_getChunkSize(Heap const* heap, MemAddr addr)`
  - ☐ `void os_setAllocationStrategy(Heap* heap, AllocStrategy allocStrat)`
  - ☐ `AllocStrategy os_getAllocationStrategy(Heap const* heap)`

☐ Weitere Funktionalitäten

- ☐ Allokationsstrategie: First-Fit
- ☐ Überprüfung des Sicherheitsabstandes durch das Symbol `__heap_start` mit Hilfe des Defines `HEAPOFFSET`

3.5 Verwendung der Doxygen-Dokumentation

Im Gegensatz zum vorherigen Versuch enthält dieser Versuch weniger Vorgaben. Dadurch werden Ihnen mehr Freiheiten bei der Implementierung der Hausaufgaben gelassen. Insbesondere die Entwicklung des dynamischen Speichers erfordert mehr Eigeninitiative, da neben den beiden Funktionen `os_malloc` und `os_free` keine genauen Vorgaben für die obere Schicht der Speicherverwaltung existieren. Um Ihnen dennoch den Übergang der Versuche zu erleichtern, empfiehlt es sich besonders in diesem Versuch, die Doxygen-Dokumentation bei der Implementierung als Hilfestellung zu nutzen.
Zusätzlich ist als Beispiel die Lösung der oberen Speicherverwaltung, wie sie in der Doxygen-Dokumentation vorhanden ist, in der folgenden Abbildung 3.7 verdeutlicht. Diese soll Ihnen zur Orientierung und zur Findung einer eigenen Lösung dienen. Sie können aber auch die Doxygen-Implementierung auf ihr eigenes Projekt übertragen.

Zusätzlich sollte in regelmäßigen Abständen mit Hilfe eigener Testtasks Ihre Implementierung überprüft werden. Dabei ist die Verwendung zur Anzeige des Speichers (siehe Kapitel *6.2.3 Überwachung des Speichers* im *Begleitenden Dokument*) sehr zu empfehlen.

Beispielimplementierung der oberen Schicht der Speicherverwaltung

1. Um Speicher zu allozieren, wird in der Funktion `os_malloc` die Allokationsstrategie bestimmt, um damit anschließend den Map-Bereich gemäß des Protokolls zu füllen. Dabei werden die beiden Hilfsfunktionen `os_getMapEntry` und `os_setMapEntry` verwendet.

2. In der Implementierung der Doxygen-Dokumentation wurde entschieden, über den Use-Bereich zu iterieren. Um dennoch auf den Map-Bereich zuzugreifen und ihn gegebenenfalls zu verändern, wird in diesen beiden Funktionen anhand einer Use-Adresse die dazu passende Map-Adresse berechnet. Aufgrund des Verhältnisses von 1:2 von Map- zu Use-Bereich, verwaltet ein Halbbyte des Map-Bereichs ein Byte des Use-Bereichs. Dies ist bei der Berechnung zu berücksichtigen. Anschließend kann mit Hilfe der vier Funktionen `get/setLow/HighNibble` das passende Halbbyte (eng. *Nibble*) verändert bzw. gelesen werden.

3./4./5. Erst mit Hilfe des Heaptreibers kann auf die unterste Treiberschicht zugegriffen werden. Dort wird die Hardware byteweise durch die Verwendung der beiden Funktionen `write` und `read` gelesen und beschrieben. Bei der Verwendung der beiden unteren Treiberfunktionen ist jedoch darauf zu achten, dass nicht das ganze Byte gelesen bzw. beschrieben werden soll, sondern immer nur ein Halbbyte. Dementsprechend muss der Übergabewert bzw. der Rückgabewert von `write` und `read` in den vier Funktionen angepasst werden.

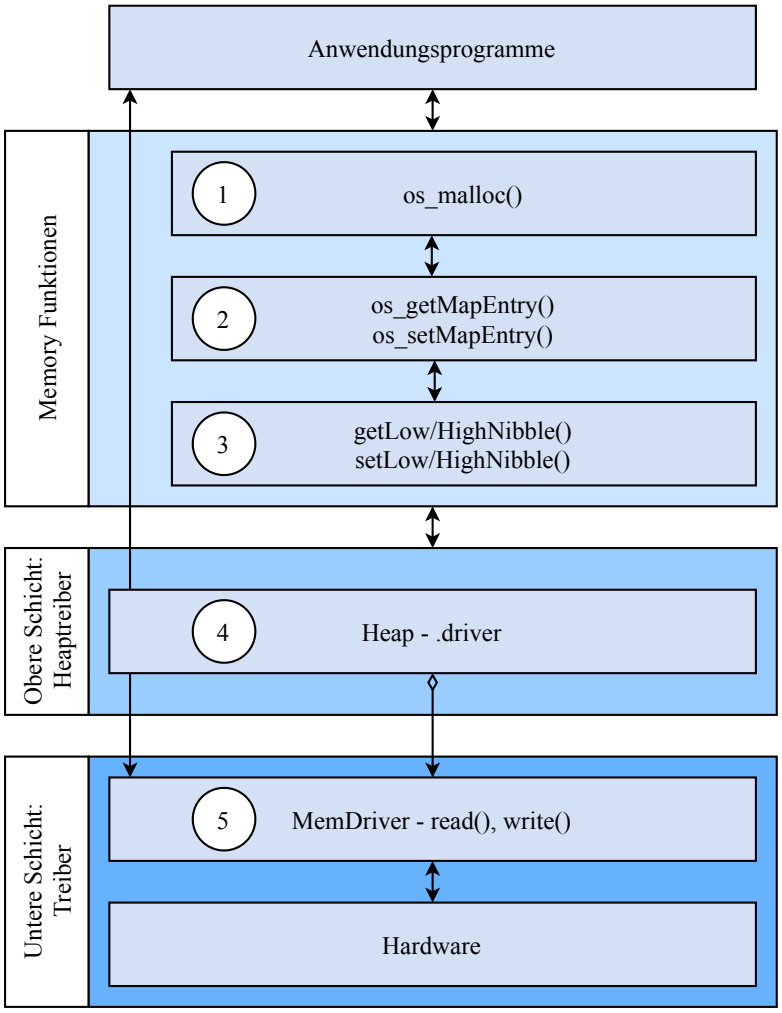

Abbildung 3.7: Beispiel der Speicherverwaltung anhand der Doxygen-Dokumentation.

3.6 Präsenzaufgaben

Die als „in V. 3. optional“ gekennzeichneten Aufgaben sind für das Bestehen von Versuch 3 optional, werden jedoch für Versuch 4 als Hausaufgabe zu bearbeiten sein. Es ist erlaubt, den Praktikumstermin nach Abschluss aller Präsenzaufgaben früher zu verlassen.

3.6.1 Testtasks

Sie finden im Moodle eine Sammlung von Testtasks, mit denen die Funktionalität Ihres Betriebssystems zum aktuellen Entwicklungsstand getestet werden kann. Am Ende des Versuchs müssen alle nicht optionalen Testtasks fehlerfrei laufen. Der korrekte Ablauf der Testtasks gemäß der ebenfalls im Moodle verfügbaren Beschreibung wird während des Versuchs geprüft. Die Implementierungshinweise, Achtung-Boxen und Lernerfolgs-

fragen in diesen Unterlagen weisen meist auf notwendige Kriterien für den erfolgreichen Durchlauf der Testtasks hin.
Wenn Ihre selbst entwickelten Anwendungsprogramme fehlerfrei unterstützt werden, starten Sie die Testtasks. 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 Testtasks auftreten.

Achtung

Sollten Sie alle zur Verfügung gestellten Testtasks erfolgreich ausführen können, ist dies keine Sicherheit dafür, dass Ihr Code fehlerfrei ist. Diese Testtasks decken lediglich einen Teil der möglichen Fehler ab.

3.6.2 Heap Cleanup (in Versuch 3 optional)

Implementieren Sie die Funktion `void os_freeProcessMemory(Heap*, ProcessID)`, die den gesamten Speicher freigibt, der von dem Prozess mit der angegebenen Prozess-ID auf dem angegebenen Heap alloziert wurde.
Fügen Sie anschließend in der Funktion `os_dispatcher` bzw. `os_kill` einen Aufruf dieser Funktion ein, um den Speicher des terminierten Prozesses sowohl im internen als auch im externen Heap automatisch freizugeben.

3.6.3 Allokationsstrategie (in Versuch 3 optional)

Implementieren Sie eine zusätzliche Allokationsstrategie, anhand welcher der Speicherbereich in `os_malloc` ausgewählt wird.

3.7 Pinbelegung

Port	Pin	Belegung
Port A	A0	LCD Pin 1 (D4)
	A1	LCD Pin 2 (D5)
	A2	LCD Pin 3 (D6)
	A3	LCD Pin 4 (D7)
	A4	LCD Pin 5 (RS)
	A5	LCD Pin 6 (EN)
	A6	LCD Pin 7 (RW)
	A7	frei
Port B	B0	frei
	B1	frei
	B2	frei
	B3	frei
	B4	frei
	B5	frei
	B6	frei
	B7	frei
Port C	C0	Button 1: Enter
	C1	Button 2: Down
	C2	Reserviert für JTAG
	C3	Reserviert für JTAG
	C4	Reserviert für JTAG
	C5	Reserviert für JTAG
	C6	Button 3: Up
	C7	Button 4: ESC
Port D	D0	frei
	D1	frei
	D2	frei
	D3	frei
	D4	frei
	D5	frei
	D6	frei
	D7	frei

Die vorgegebene Pinbelegung des ATmega 644 in Versuch 3 (*Heap / Schedulingstrategien*).