RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

LEHRSTUHL
INFORMATIK 11
SOFTWARE FÜR
EINGEBETTETE
SYSTEME

Praktikum Systemprogrammierung

Versuch 5

Speicher

Lehrstuhl für Informatik 11 - RWTH Aachen

1. Juni 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

5 Speicher

In heutigen Computersystemen ist der interne Speicher des Hauptprozessors, der Cache, häufig nicht ausreichend groß. Dieser Speicher wird daher durch zusätzlichen Arbeitsspeicher ergänzt, der wesentlich größer und kostengünstiger ist. Dieser Arbeitsspeicher ist durch die Methoden zur Ansteuerung und durch die verwendete Speichertechnologie wesentlich langsamer als der interne Speicher. Ist der interne Speicher zu klein, muss auf den langsameren Erweiterungsspeicher ausgewichen werden.

Beispielsweise verfügt eine moderne x64 CPU über einige Megabyte integrierten Speicher (u.A. L1- und L2-Cache), verwendet aber einen externen Speicher von typischerweise mehreren Gigabyte.

Sollen Daten dauerhaft gespeichert werden, muss nicht-flüchtiger/nicht-volatiler Speicher verwendet werden, da der integrierte sowie der Erweiterungsspeicher nach Neustarten des Systems gelöscht ist. Derzeitige Workstations verfügen beispielsweise über mehrere hundert Gigabyte bis zu einigen Terabyte dauerhaften Speicher.

Diese Konzepte sind ebenso auf einen Mikrocontroller übertragbar, wenn auch in wesentlich kleineren Dimensionen. In diesem Versuch wird daher auf externe Hardware zurückgegriffen.

5.1 Versuchsinhalte

In diesem Versuch wird das in den vorhergehenden Versuchen genutzte Evaluationsboard mit einer Erweiterungsplatine ausgestattet. Diese enthält einen 8 kB großen RAM-Baustein, der über wenige Steuerleitungen kontrolliert werden kann.

Analog zum internen SRAM wird für den externen SRAM ein neuer Speichertreiber benötigt. Darüber hinaus wird ein Treiber für den integrierten EEPROM implementiert, der als permanenter Speicher verwendet wird.

5.2 Lernziel

Das Lernziel dieses Versuchs ist das Verständnis der folgenden Zusammenhänge:

- Kommunikation mit externer Hardware
- Umgang mit nicht-flüchtigem Speicher
- Größenänderung dynamischer Speicherbereiche
- Amortisierte Laufzeitanalyse

5.3 Grundlagen

Im Folgenden werden die wichtigsten Bauteile der Hardware erläutert. Abbildung 5.1 zeigt die verwendete Platine und Abbildung 5.2 die schematische Verschaltung. Beide Abbildungen enthalten die Bauteile RAM und Latch, die in den folgenden Abschnitten erläutert werden.

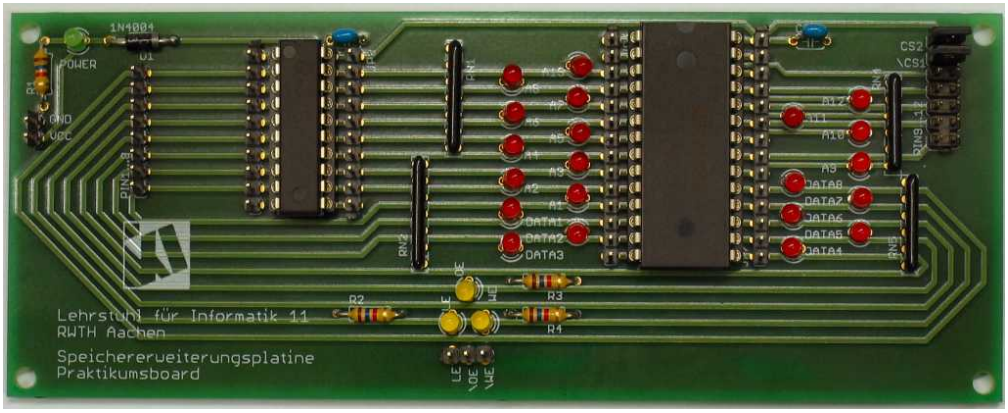

Abbildung 5.1: Die Speichererweiterungsplatine mit Latch und SRAM-Baustein

5.3.1 RAM-Baustein

Aufgaben des RAMs

Der externe RAM hat die Aufgabe Arbeitsdaten des Mikrocontrollers zu speichern, die aufgrund ihrer Größe nicht im internen RAM gespeichert werden können. Die Daten werden byteweise in den RAM geschrieben und entsprechend daraus gelesen.

Aufbau des RAMs

Der in diesem Versuch verwendete Speicherbaustein HY6264A ist ein 8192 Byte großer statischer RAM-Baustein. Die folgenden Pinbeschreibungen sind dem zugehörigen Datenblatt entnommen, welches im L^2P-Lernraum verfügbar ist.
Die wesentlichen Ein- und Ausgangspins dieses ICs sind:

- 13 Adresspins (`Adr1`, ..., `Adr13`)
- 8 Datenpins (`D1`, ..., `D8`)
- 1 Pin zum Schreiben der Daten (\`WE`)
- 1 Pin zum Lesen der Daten (\`OE`)

Hierbei ist zu beachten, dass die Pins *Write Enable* (`WE`) und *Output Enable* (`OE`) invertiert sind. Das bedeutet, eine logische Null (GND) muss angelegt werden, um die

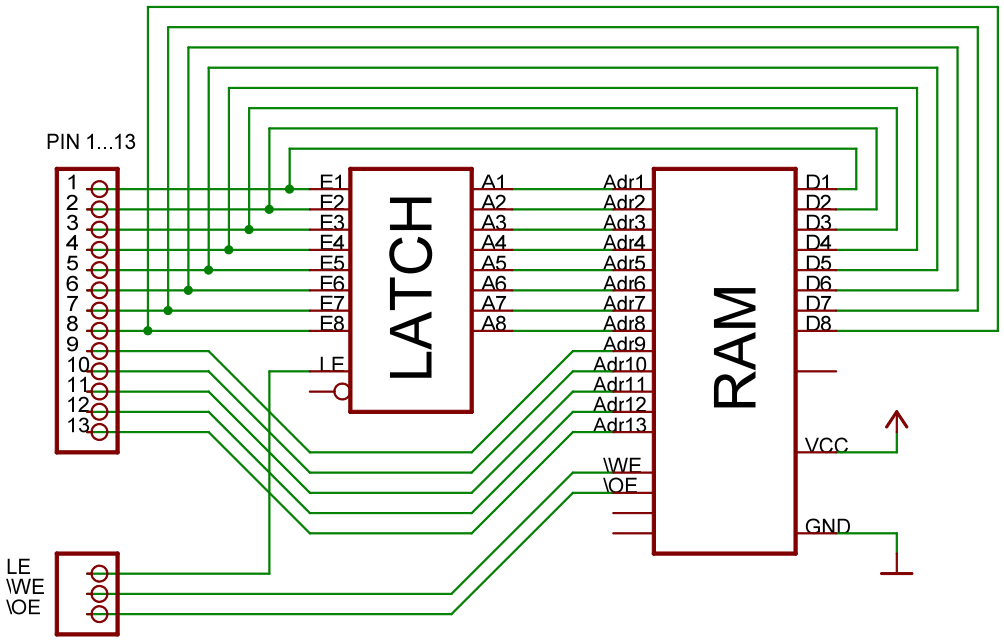

Abbildung 5.2: Schematisch dargestellte Verschaltung der RAM Erweiterungsplatine

jeweilige Operation auszuführen und eine logische Eins (V_{cc}), um die Operation nicht auszuführen. Diese Eigenschaft wird oft durch Voranstellen eines Schrägstriches in der Beschriftung der Eingabepins dargestellt.

Zu beachten ist, dass *Write Enable* Priorität hat. Sollten unbeabsichtigt beide Kontrollpins auf logisch Null gesetzt werden, wird trotzdem der Schreibvorgang ausgeführt. Um einen Lesevorgang zu starten, muss daher strikt darauf geachtet werden, dass *Write Enable* logisch Eins ist.

Funktionsweise des RAMs

Schreiben Zum Beschreiben des RAM-Bausteins wird die gewünschte Speicheradresse an die Adresspins (`Adr1`, ..., `Adr13`) angelegt; an die Datenpins (`D1` ... `D8`) wird das Byte gelegt, das geschrieben werden soll. Anschließend wird der Pin `\WE` kurzzeitig auf logisch Null gesetzt, um den Schreibvorgang auszuführen.

Ein solcher Schreibvorgang ist exemplarisch in Abbildung 5.3 dargestellt. Dort wird an die Adresse `0x6E4` das Datum `0x42` geschrieben, indem dies zunächst an die Datenleitung angelegt und anschließend `\WE` auf `0` gesetzt wird.

Anmerkung: Die Reihenfolge, wie eine 13 Bit Adresse an die Adresspins angelegt wird, ist beliebig wählbar, solange immer die gleiche Permutation benutzt wird.

Lesen Zum Lesen wird die gewünschte Adresse an die Adresspins gelegt. Anschließend wird der Pin `\OE` auf logisch Null gesetzt. An den Datenpins des RAM-Bausteins wird

daraufhin das gewünschte Datum zur Verfügung gestellt. Der Wert liegt so lange an, wie \OE auf logisch Null steht.

Ein Lesevorgang ist exemplarisch in Abbildung 5.3 dargestellt. Dort wird der zuvor geschriebene Wert wieder ausgelesen, indem 0 an \OE angelegt wird.

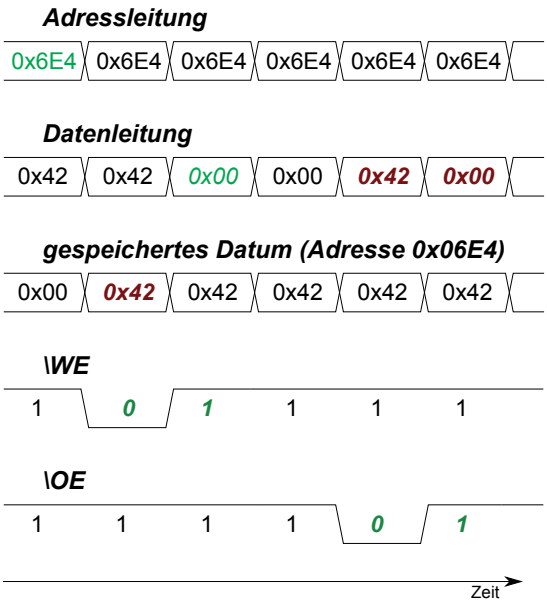

Abbildung 5.3: Signalverlauf eines Schreib-/Lesezyklus. Änderungen sind kursiv markiert. Grün gekennzeichnet sind Änderungen durch den Mikrocontroller. Rot gekennzeichnet sind Änderungen durch den RAM. Diese sind zusätzlich in Fettschrift gesetzt. Aus Platzgründen wird hier davon ausgegangen, dass der RAM-Baustein Änderungen unverzüglich übernimmt.

Lernerfolgsfragen

- Welche Auswirkungen hat die Tatsache, dass die Pins \WE und \OE des RAM-Bausteins logisch invertiert sind?
- Welche Spannung liegt an den entsprechenden Pins an, wenn dem RAM-Baustein eine Schreib- bzw. Leseanweisung übermittelt wird?
- Wie lange liegt beim Lesen der Wert an den Datenpins an?
- Warum ist die Reihenfolge der Adressbits beliebig?
- Ist die Reihenfolge der Datenbits beliebig?

5.3.2 Latch-Baustein

Aufgaben des Latches

Der in 5.3.1 beschriebene RAM-Baustein benötigt eine Vielzahl von verschiedenen Ein- und Ausgabeleitungen zur Kommunikation. Der in diesem Praktikum genutzte Mikrocontroller besitzt nur 32 Ein- bzw. Ausgabeleitungen, von denen ein Teil bereits für die anderen Funktionalitäten, wie z. B. das LC-Display oder die Buttons, genutzt wird. Der RAM-Baustein benötigt 23 Ein- bzw. Ausgänge (13 Adressleitungen, 8 Datenleitungen und 2 Steuerleitungen); also mehr als verfügbar sind. Aus diesem Grund wurde in die Schaltung ein 8 Bit Latch-Baustein eingefügt, welcher zur Zwischenspeicherung eines Datenbytes genutzt wird.

Aufbau des Latches

Als Latch wurde für diesen Versuch ein integrierter Schaltkreis mit der Bezeichnung 74HC 573 verwendet. Die nachfolgenden sowie weiterführenden Informationen können dem zugehörigen Datenblatt entnommen werden, welches im L^{2}P-Lernraum zur Verfügung steht.
Die relevanten Pins dieses Bausteins sind:

- 8 Eingabepins (`E1` ... `E8`)
- 8 Ausgabepins (`A1` ... `A8`)
- 1 Pin zum Halten des Latches (`LE`)

Funktionsweise des Latches

Wird der Pin *Latch Enable* (`LE`) auf logisch Null gesetzt, bleiben die Werte der Ausgabepins konstant und ändern erst wieder ihren Wert, wenn `LE` auf logisch Eins gesetzt wird. In diesem Zustand (`LE = 1`) entspricht die Ausgabe der Eingabe. Dies ist exemplarisch in Abbildung 5.4 für ein Bit des Bausteins dargestellt, an welches ein Rechtecksignal angelegt ist.

Lernerfolgsfragen

- Wie ist die Arbeitsweise des hier vorgestellten Latches charakterisiert?
- Wie muss das Latch konfiguriert werden, damit `Ei` = `Ai` für `1` $\leq$ `i` $\leq$ `8` gilt?

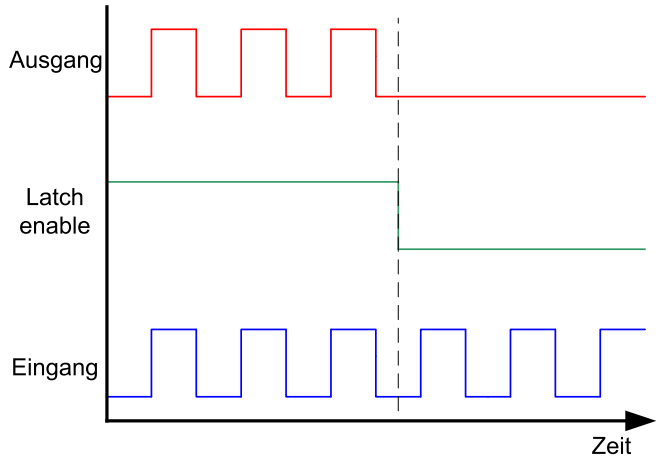

Abbildung 5.4: Funktionsweise eines Latches (hier exemplarisch für 1 Bit)

5.3.3 Die Erweiterungsplatine

Die Erweiterungsplatine verbindet den RAM-Baustein mit dem Latch-Baustein. Auf diese Weise werden, wie oben erläutert, weniger Ein- und Ausgabepins benötigt. Der Ablauf eines Zugriffs wird im Folgenden erläutert:

8 der 13 Adressbits werden an den Latch angelegt und `LE` auf logisch Null gesetzt. Anschließend werden dieselben Eingangspins genutzt, um das Datenwort von `D1` bis `D8` zu lesen bzw. zu schreiben. Dadurch können die Adresse und das Datenwort gleichzeitig an den RAM-Baustein angelegt werden, ohne dass der Mikrocontroller alle Bits auf verschiedenen Pins gleichzeitig anlegen muss. Durch diese Verschaltung werden für die Kommunikation mit dem externen RAM 7 Pins eingespart.

Die Erweiterungsplatine verfügt zusätzlich über LEDs am Daten- und Adressbus, sowie an den Steuerleitungen. Dabei ist zu beachten, dass die LEDs an den Bussen jeweils leuchten, wenn der zugehörige Pin den Wert 1 angenommen hat. Die LEDs der Steuerleitungen leuchten für `LE`, falls diese Leitung auf 1 bzw. bei `\WE` und `\OE`, falls diese auf 0 gesetzt sind (also wenn die jeweils entsprechende Funktion aktiviert ist).

Lernerfolgsfragen

- Wie viele Pins des Mikrocontrollers wären für die Arbeit mit dem RAM-Baustein ohne Latch nötig?
- Wie viele Pins sind bisher bereits durch andere Peripherie *belegt* beziehungsweise *reserviert*? Hinweis: Siehe Tabelle 5.6.
- Wieso werden durch den Einsatz des Latches 7 Pins weniger benötigt, obwohl das Latch 8 Bit speichert?
- Der ATmega 644 stellt nicht genügend Ausgangspins zur Verfügung. Wie viele Bits muss der Latch-Baustein minimal speichern können, um dieses Problem zu lösen?
- Wieviel Bit können im hier verwendeten RAM-Bausteins maximal ohne Verwendung eines Latches gespeichert werden? Hinweis: Wie können die vorhandenen Pins am besten auf Adress-, Daten- und Steuerleitungen verteilt werden.

5.3.4 Integrierter EEPROM

Der ATmega 644 verfügt über einen integrierten EEPROM („*Electrically Erasable Programmable Read Only Memory*“) der 2048 Byte dauerhaft speichern kann.

Dieser Speicher ist ähnlich zum Flash-Speicher organisiert. Im Gegensatz zum wesentlich größeren Flash-Speicher, der das auszuführende Programm enthält, ist der EEPROM zum Ablegen beliebiger Daten wie Konfigurationseinstellungen vorgesehen.

Die Verwendung des EEPROM als Heap ist aufgrund seiner Geschwindigkeit und Größe nicht zweckmäßig. Hinzukommt, dass der EEPROM nur eine endliche Anzahl von Schreibzugriffen unterstützt, bevor er fehlerhaft wird. Stattdessen kann dieser Speicher mit einem rudimentären Dateisystem ausgestattet werden, das Prozessen erlaubt, dynamisch persistente Daten zu sichern.

Lernerfolgsfragen

- Wieviel Daten können im EEPROM insgesamt gespeichert werden?
- Wieso ist die Verwendung des EEPROM sinnvoll, obwohl SPOS bereits über ca. 10 Kilobyte dynamischen Speicher verfügt?
- Wieviel Schreibzugriffe unterstützt der EEPROM laut Datenblatt?

5.3.5 Dynamische Speicheranpassung

In Versuch 3 (*Heap / Schedulingstrategien*) wurden zwei wichtige Funktionen zur Verwaltung dynamischen Speichers implementiert: `os_malloc` und `os_free`. Häufig sind jedoch die anfallenden Datenmengen im Vorfeld, d.h. beim Anfordern eines Bereiches, nicht bekannt. In solchen Fällen können verkettete Listen o.Ä. Datenstrukturen verwendet werden, um Datenmengen variabler Größe zu verwalten.

Hierbei gibt es zwei wesentliche Probleme: Zunächst geht bei diesen Datenstrukturen die zentrale Eigenschaft des wahlfreien Zugriffs (in konstanter Zeit) des verwendeten Speichers verloren. Zusätzlich ist der Overhead (zusätzlicher Speicherverbrauch für die Vewaltungsdatenstrukturen) bei vielen Heapimplementierungen konstant, anstatt wie bei SPOS linear in der Größe eines Speicherabschnitts, sodass viele kleine Speicherabschnitte ineffizienter sind als ein großer. Hinzu kommt der Overhead der Datenstrukturen selbst, welche pro Element mindestens einen Pointer benötigen.

Dies motiviert, dass Speicherabschnitte dynamisch vergrößert und verkleinert werden. Dies wird als Reallokation bezeichnet. Außerhalb der Speicherverwaltung kann dies nur geschehen, indem ein neuer (passender) Speicherbereich mit `os_malloc` angefordert wird, alle Daten in den neuen Bereich kopiert werden und der alte Bereich mit `os_free` wieder freigegeben wird; siehe Listing 5.1.

```
1  MemAddr myRealloc(MemDriver* driver, MemAddr oldPtr,
2                    uint16_t oldSize, uint16_t newSize) {
3    MemAddr const newPtr = os_malloc(driver,newSize);
4    if (!newPtr)
5      return NULL;
6    uint16_t i = (oldSize < newSize)?oldSize:newSize;
7    while (i--)
8      driver->write(newPtr+i,driver->read(oldPtr+i));
9    os_free(driver,oldPtr);
10   return newPtr;
11 }
```

Listing 5.1: Naive Anpassung der Größe eines Speicherabschnitts (Reallokation) durch einen Anwendungsprozess ohne Zugriff auf interne Datenstrukturen.

Es hat den Anschein als würde diese Funktion korrekt arbeiten. Dennoch ist dieser naive Ansatz problematisch und sollte daher nicht verwendet werden. Hierfür gibt es zwei wesentliche Gründe:

Zunächst kann der restliche Speicher zu klein sein, um den neuen Abschnitt aufzunehmen, obwohl der neue Speicherbereich nach Entfernen des ursprünglichen Bereichs passen würde. Dies ist in Abbildung 5.5 für die Reallokation mit `newSize = 5` dargestellt.

Außerdem ist das Verschieben der Daten (Zeile 5-7 in Listing 5.1) vergleichsweise aufwändig. Falls, wie im Beispiel in Abbildung 5.5, der alte Speicherabschnitt in einem

Abbildung 5.5: Fragmentierung führt bei Reallokation des dunkelblauen Bereichs mit Größe 5 zu falschem Verhalten von Listing 5.1. Jede kleinere Anfrage kann bedient werden, jedoch vergleichsweise nur sehr ineffizient.

Bereich liegt, der auch den neuen Speicherabschnitt beinhalten könnte, so können die Daten an der bisherigen Adresse stehen bleiben, anstatt verschoben zu werden. Für genaue Effizienzbetrachtungen – also Untersuchung der Gesamtlaufzeit – wäre selbstverständlich noch die Laufzeit der `os_malloc` und `os_free` Funktionen zur Gesamtlaufzeit hinzuzuzählen.

Falls die Reallokation von der Speicherverwaltung des Betriebssystems durchgeführt wird, ist diese von den hier beschriebenen Effekten, je nach Implementierung, nicht betroffen, da diese Zugriff auf die gesamte Struktur des Heaps (bei SPOS auch der Heapmap) hat. Hierbei kann effizient geprüft werden, ob der Speicherbereich vergrößerbar ist, oder ob ein neuer Bereich angefordert werden muss.

Lernerfolgsfragen

- Wie groß ist der Overhead der Speicherverwaltung von SPOS? Welcher Overhead wird benötigt, um die Zeichenkette `"Informatik"` ohne Nullterminierung zu speichern?
- Welche Probleme treten auf, wenn die Reallokation durch Anwendungsprozesse implementiert wird, anstatt von der betriebssysteminternen Speicherverwaltung? Betreffen diese Probleme die Korrektheit, die Effizienz oder beides?
- Unter welcher Bedingung ist eine effiziente Vergrößerung eines Speicherbereichs nicht möglich?
- Gibt es ein Beispiel, bei dem das Verschieben der Daten notwendig ist, obwohl insgesamt nur ein einziger Speicherabschnitt existiert? Wie sieht dies aus?

5.3.6 Effiziente Verwendung einer Reallokationsroutine

Eine realistische Annahme für einen mittel bis stark fragmentieren Speicher ist, dassdie Reallokationsroutine im Erwartungswert nicht besser als linear ($\Omega(n)$) in der Größe der neuen Abschnittsgröße n ist. Für die Realisierung eines dynamischen Puffers[1] ist Reallokation bei jeder Änderung der Größe des Speichers daher nicht sinnvoll, falls sich die Größe häufig ändert.

[1] Dies ist ein dynamisches Array, das immer um einen Eintrag verkleinert oder vergrößert wird.

Wenn die Reallokation im Erwarungswert nicht nur nicht besser, sondern auch nicht schlechter als linear ($\mathcal{O}(n)$) ist, lässt sich durch einen „seltenen“ Aufruf der Reallokationsfunktion die Gesamtlaufzeit deutlich reduzieren, falls man in Kauf nimmt, dass mehr Speicher belegt wird, als mindestens notwendig. Dies bezeichnet man als amortisierte Analyse, siehe dafür die Vorlesung „*Datenstrukturen und Algorithmen*“.

Lernerfolgsfragen

- **Erwartete** Laufzeit der Reallokation für einen Speicher der Größe n.
  - Können Sie die Abschätzung mit $\Omega(n)$ motivieren? Gilt dies bei starker oder schwacher Fragmentierung?
  - Können Sie die Abschätzung mit $\mathcal{O}(n)$ motivieren? Gilt dies bei starker oder schwacher Fragmentierung?
- Wie kann ein Algorithmus konstruiert werden, der in einen dynamischen Puffer in amortisiert konstanter Zeit einfügen (*enqueue*) sowie herausnehmen (*dequeue*) kann und dabei einen Speicheroverhead von maximal Faktor 4 hat (falls die erwartete Laufzeit der Reallokation eines Speicherbereichs in $\Theta(n)$ ist)?

5.4 Hausaufgaben

In den Versuchen 3 und 4 wurde bereits die Speicherverwaltung für den internen Heap implementiert. Diese wurde so konzipiert, dass das Austauschen des Datenträgers die Implementierung eines neuen Treibers erfordert, jedoch keine Änderungen an der Implementierung der Speicherverwaltung notwendig sind. Dieser Treiber besitzt die Funktionen *Initialisieren*, *Lesen* und *Schreiben*. Darüber hinaus enthält der Treiber spezifische Eigenschaften des jeweiligen Speichermediums wie dessen Größe.

Inhalt dieses Aufgabenblocks ist die Berechnung dieser Eigenschaften und Implementierung der Funktionen für den externen Speicher. Analog zu Versuch 3 (*Heap / Schedulingstrategien*) werden diese Fragmente in einer Treiber-Datenstruktur zusammengefasst.

5.4.1 Spezifische Größen des RAM-Bausteins

Wie die Definitionen aus den vorherigen Versuchen, welche die Größen des internen Heaps charakterisieren, sollen die Größen des externen RAM-Bausteins in Form von Definitionen abgelegt werden.

Achtung

Bedenken Sie bei der Implementierung der nachfolgenden Funktionen, dass der externe RAM im Gegensatz zum internen Datenspeicher von Adresse 0 an adressiert werden kann. Der RAM hat eine Größe von 2^{13} Byte, jedoch nutzt Ihre Speicherfunktionen Blöcke, deren Größe ein Vielfaches von 3 Byte (= 1 Byte Allokationstabelle + 2 Byte Nutzdaten) ist.

Die Steuersignale (`LE`, `\OE` und `\WE`) werden bei der Ansteuerung des RAM Bausteins häufig geändert, es bietet sich daher an, hierfür zusätzliche Definitionen anzulegen. Die Verwendung von Definitionen hat den Vorteil gegenüber Unterfunktionen, weniger Taktzyklen zur Abarbeitung erfordern, und damit den Zugriff auf den RAM nicht verlangsamen. Ein Beispiel für die Verwendung von Definitionen ist:

```
1 #define LATCH_ENABLE (LATCH_PORT |= 1 << LATCH_PIN)
```

Das in diesem Beispiel definierte `LATCH_ENABLE` setzt den `LATCH_PIN`ten Pin von `LATCH_PORT` auf logisch 1. Somit gibt das Latch alle an den Eingangspins angelegten Daten direkt an die Ausgangspins weiter.

Dazu müssen Sie sich überlegen, welche Datenleitung des Latches bzw. des RAM-Bausteins Sie an welchen Pin des ATmega 644 anschließen wollen. Eine Auflistung aller Pins findet sich in Tabelle 5.6 am Ende dieses Dokuments.

Lernerfolgsfragen

- Woraus ergibt sich die Anforderung, dass die nutzbare externe Speicherkapazität ein Vielfaches von drei Byte sein muss?
- Hat die Verwendung von Definitionen anstatt Unterfunktionen Nachteile?
- Welche Größen definieren in SPOS einen Datenträger?
  - Gibt es Redundanzen? Begründen Sie Ihre Antwort.
  - Was sind die Vor- und Nachteile (keine) Redundanzen in den charakteristischen Attributen des Datenträgertreibers zu speichern?

5.4.2 Funktionen zum externen Speicherzugriff

Analog zur unteren Schicht des internen Speichers, werden auch Funktionen zur Erstellung eines Treibers für den externen Speicher benötigt. Erstellen Sie die in den folgenden Abschnitten beschriebenen Funktionen und Datenstrukturen.

Lesen des RAMs

Legen Sie eine private Funktion an, welche einen Parameter vom Typ `MemAddr` erhält – dies ist die zu lesende Adresse. Die Rückgabe dieser Funktion soll vom Typ `MemValue` sein. Der Rückgabewert ist das gelesene Datenwort. Beachten Sie bei Ihrer Implementierung die Verschaltung des Latch- und RAM-Bausteins, wie in Kapitel 5.3 beschrieben.

Achtung

In seltenen Schreib-/Lesekonstellationen (Schreiben eines Datenwortes, das den ersten 8 Bit der Adresse entspricht, und anschließendes Lesen aus dieser Adresse) kann es zu Fehlfunktionen im RAM-Baustein kommen. Daher sollte vor dem Anlegen der eigentlichen Adresse ein anderes Datenwort an die Pins 1 bis 8 angelegt werden. Dazu bietet sich die bitweise invertierte Adresse an, da hier alle Bits mindestens einmal geändert werden. Diese Funktionalität kann in eine eigene Funktion ausgelagert werden, die das Anlegen der Adresse übernimmt. Dies wird sowohl beim Lesen, als auch beim Schreiben benötigt.

Beim Lesen aus dem externen Speicherbaustein müssen Sie den Latch-Baustein in Ihre Überlegungen mit einbeziehen. Darüber hinaus müssen Sie beim Lesen berücksichtigen, dass, nachdem Sie `\OE` auf logisch Null gesetzt haben, eine Pause von mindestens 200 ns notwendig ist, bevor das Datenwort zuverlässig ausgelesen werden kann. Dazu können

Sie z. B. die Assembleranweisung `nop` nutzen, die keinen Effekt hat, außer dass ein Taktzyklus verstreicht. Rechnen Sie dazu aus, wieviele Instruktionen Sie benötigen, um 200 ns verstreichen zu lassen[2]. Beachten Sie, dass der Mikrocontroller 2 Taktzyklen nach dem Umschalten eines oder mehrerer Pins von Ausgangs- zu Eingangspins benötigt, bis korrekte Werte eingelesen werden können. Dieses Problem können Sie umgehen, indem Sie zuerst den entsprechenden Port auf Eingang umschalten, und erst dann `\OE` setzen. Anschließend kann, nach der oben erwähnten Pause, das 8 Bit breite Datenwort eingelesen werden.

Lernerfolgsfragen

- An welcher Stelle in Ihrem Code müssen Sie die zuvor beschriebene Lesefunktion ablegen bzw. aufrufen?
- Können der Lesefunktion weitere als die oben beschriebenen Parameter übergeben werden?

Beschreiben des RAMs

Legen Sie eine weitere private Funktion an, welche zwei Argumente erhält: Die Adresse (vom Typ `MemAddr`) und das zu schreibende Byte (vom Typ `MemValue`). Für diese Funktion gelten die gleichen Hinweise wie für die Funktion, die ein Byte liest.

Damit ein an den RAM-Baustein angelegtes Datenwort gespeichert wird, müssen Sie den Pin `\WE` für mindestens 200 ns auf logisch Null setzen. Beachten Sie die Hinweise zum Lesen des RAMs.

Initialisierung des RAMs

Sie benötigen eine Funktion, die den RAM initialisiert, also in einen definierten Zustand überführt. Somit sind die Konfiguration der Ports, die Sie für die Kommunikation benötigen und das Initialisieren des Speichers mit Nullen nötig.

Lernerfolgsfragen

- Erwarten Sie Probleme, wenn Sie die Map zu Beginn nicht vollständig mit Nullen überschreiben?
- Erwarten Sie Probleme, wenn Sie den Heap zu Beginn nicht vollständig mit Nullen überschreiben?

[2] Der ATmega 644 ist mit einem externen Quarz mit 20 MHz getaktet.

5.4.3 Treiber für den externen RAM erstellen

Erstellen Sie analog zum Treiber aus Versuch 3 eine Treiberstruktur vom Typ `MemDriver` in `os_mem_drivers.c/.h`. Wiederum analog zu `intSRAM` muss ein Symbol `extSRAM` erstellt werden, welches den Zeiger auf die Treiberstruktur enthält. Initialisieren Sie die einzelnen Elemente der Struktur mit den passenden Werten bzw. Funktionszeigern. Dazu ist es hilfreich, analog zu den vorherigen Versuchen Definitionen anzulegen. Beachten Sie, dass für den externen Speicher eine Initialisierungsfunktion erstellt werden muss und stellen Sie sicher, dass diese an geeigneter Stelle (z. B. in der Funktion `os_initDrivers`) aufgerufen wird.

5.4.4 Hinweise zur Simulation

Im L^{2}P-Lernraum steht eine XML-Konfigurationsdatei für HapSim bereit, mit der die Pinbelegung, die während des Versuchs für den Anschluss der RAM-Platine verwendet wird, simulierbar ist. Die LEDs auf der simulierten RAM-Platine verhalten sich analog zu denen auf der realen Platine (siehe Kap. 5.3.3). Für die Simulation ist es wichtig, zunächst die Datenrichtung eines Ports zu ändern und erst dann die gewünschten Werte anzulegen. Sollte Ihr Code nicht korrekt mit der RAM-Platine kommunizieren, erhalten sie in HapSim (im Feld *Output*) eine entsprechende Fehlermeldung; achten Sie während des Testens darauf.

Achtung

Der simulierte RAM-Baustein arbeitet wesentlich langsamer als der reale Baustein, wodurch z. B. die Initialisierung des gesamten Speichers sehr viel Zeit in Anspruch nimmt. Es empfiehlt sich daher, in HapSim nur die Grundfunktionen zu testen und mit Hilfe der in 5.4 vorgestellten Definitionen den RAM-Baustein zu verkleinern.

Es besteht natürlich weiterhin die Möglichkeit, Ihren Code vorab an den im RBI zur Verfügung gestellten Boards zu testen. Auf diesen Boards befindet sich die in diesem Versuch genutzte RAM-Erweiterungsplatine.

Lernerfolgsfragen

- Warum wird vor dem Lesen eines Datenwortes auf den externen RAM-Baustein die invertierte Adresse angelegt? Was ist der Vorteil gegenüber einem kostanten Wert?
- Warum kann der externe RAM im Gegensatz zum internen RAM ab Adresse 0 verwendet werden?
- Wie viel Byte des externen RAM können insgesamt genutzt werden? Wie groß ist der RAM insgesamt?

5.4.5 EEPROM

Wie in 5.3.4 beschrieben, ist die Verwendung des EEPROM als Heap zum zwischenspeichern von Prozessdaten nicht sinnvoll. Zur Vereinfachung soll jedoch trotzdem der EEPROM-Zugriff mit einer `MemDriver` Treiberstruktur `eepROM` implementiert werden, um mit der Speicherverwaltung ein rudimentäres Dateisystem zu bilden.

Anstatt des im Datenblatt des ATmega 644 vorgestellten Zugriffs auf den EEPROM über `EEAR`, `EEDR` und `EECR`, kann der EEPROM einfacher mit der `avr/eeprom.h` Bibliothek angesprochen werden. Für diesen Versuch sind jedoch lediglich die folgenden Funktionen relevant:

- `uint8_t eeprom_read_byte(uint8_t const* address)`
  Diese Funktion liest den Wert, der an der Adresse `address` im EEPROM gespeichert ist und liefert diesen zurück.
- `void eeprom_write_byte(uint8_t* address, uint8_t value)`
  Diese Funktion schreibt den Wert `value` an die Adresse `address` im EEPROM.

Verwenden Sie für die Spezifikation der Größe des EEPROMs die in der vorgegebenen Datei atmega644constants.h oder legen Sie die Größe selbst fest.

Der EEPROM soll mithilfe von der Funktionen `os_malloc` und `os_free` verwaltet werden können. Die Speicherung des Ergebnisses von `os_malloc` in Variablen welche im SRAM liegen ist für dauerhaften Speicher ungeeignet, da ihr Wert nach Neustarten des Systems nicht mehr vorhanden ist. Daher ist es notwendig, Variablen im EEPROM abzulegen, die dauerhaft die durch `os_malloc` vergebenen Adressen speichern. Diese können mit dem Attribut `EEMEM` angelegt werden, wie in Listing 5.2 gezeigt. Daher ist es notwendig, den Anfang der Heapmap des EEPROM nicht an Adresse 0 zu legen, da der Heap ansonsten mit den statischen EEPROM-Variablen kollidiert.

```
// ... mehr Header ...
#include <avr/eeprom.h>

// Feste Variable im EEPROM.
MemAddr e_ptr EEMEM = 0;

PROGRAM(1,AUTOSTART) {
  // Wert der Vairablen auslesen (logisch: *&e_ptr)
  MemAddr ptr = eepROM->read((MemAddr)&e_ptr);
  // Wurde bereits ein Bereich angefordert?
  // Und ist dieser noch gültig, oder wurde der EEPROM
  // zwischenzeitlich gelöscht?
  if (!ptr || eepROM->read(ptr) != 0xAC) {
    // 50 Byte permanenten Speicher anfordern.
    ptr = os_malloc(eepROM,51);
    // Erhaltenen Speicher dauerhaft merken.
    eepROM->write((MemAddr)&e_ptr,ptr);
    // Canary setzen (AC).
    eepROM->write(ptr,0xAC);
  }
  // ptr auf die erste Adresse setzten.
  ptr++;
  // ... weitere Daten im EEPROM ablegen ...
}
```

Listing 5.2: Verwaltung von dauerhaften EEPROM Adressen.

5.4.6 Reallokation

Implementieren Sie die Funktion `os_realloc`, welche die Größe eines Speicherabschnittes dynamisch ändern kann, und dabei die in Abschnitt 5.3.5 aufgezeigten Probleme löst. Überlegen Sie sich welche Randfälle eintreten können und welche Möglichkeiten existieren, um einen Speicherbereich zu vergrößern. Die Funktion `os_realloc` hat folgende Signatur:

```
MemAddr os_realloc(MemDriver* driver, MemAddr ptr, uint16_t size)
```

Falls es nicht möglich ist, den durch `ptr` angegebenen Speicherbereich zu vergrößern bzw. zu verschieben, muss der übergebene Speicherbereich unverändert bleiben und `NULL` zurückgeliefert werden (analog zur Funktion `myRealloc` in Listing 5.1).

5.4.7 Effizienter dynamischer Puffer (optional)

Implementieren Sie eine Datenstruktur, die einen effizienten dynamischen Puffer (siehe Listing 5.3) implementiert, wie in Abschnitt 5.3.6 motiviert. Achten Sie darauf, dass die Laufzeit beider Operationen (`enqueue`, `dequeue`) amortisiert konstant ist, unter der Annahme, dass `os_realloc` eine Laufzeit in $\Theta($`size`$)$ besitzt.

```
typedef struct Buffer {
  /* ... beliebige Variablen ... */
  // Initialisiere Objekt. Verwende angegebenen Treiber.
  void (*init)(struct Buffer*, MemDriver*);
  // Lese die aktuell genutzte Größe des Objektes aus.
  size_t (*getSize)(struct Buffer*);
  // Füge ein neues Element am Ende des Puffers ein.
  void (*enqueue)(struct Buffer*, uint8_t);
  // Lösche das älteste Element und gib es zurück.
  uint8_t (*dequeue)(struct Buffer*);
} Buffer;
```

Listing 5.3: Schnittstelle des effizienten dynamischen Puffers.

5.4.8 Übersicht der zu bearbeitenden Hausaufgaben

Die folgende Aufzählung stellt eine Zusammenfassung der bereits erläuterten Aufgaben dar. Sie dient zur Übersicht der verpflichtend zu bearbeitenden Hausaufgaben.

Im Vorfeld dieses Versuchs muss vorbereitet und fristgerecht hochgeladen werden:

- Spezifische Größen des RAM-Bausteins; festgelegt als Definitionen
- Spezifische Größen des EEPROM; festgelegt als Definitionen
- Untere Schicht für jeweils den externen RAM und den EEPROM:
  - Initialisierungsfunktion
  - Funktion zum Lesen
  - Funktion zum Schreiben
  - Speichertreiber, der die ermittelten Größen und Zeiger auf die Zugriffsfunktionen enthält
- Funktion `os_realloc`
- Effiziente `Buffer` Datenstruktur

5.5 Präsenzaufgaben

Die folgenden Aufgaben sind während des Praktikums zu bearbeiten, dürfen jedoch soweit sinnvoll bereits vorbereitet werden.

5.5.1 Durchlauf der Testsuite

Sie erhalten während des Versuchs von den Betreuern eine Sammlung von Testprogrammen, mit denen die Funktionalität Ihres Betriebssystems zum aktuellen Entwicklungsstand getestet wird. Wenn es mit diesen Programmen Probleme gibt, haben Sie wahrscheinlich bestimmte Anforderungen nicht erfüllt oder gewisse Sonderfälle nicht abgefangen. Ergänzen Sie Ihr Projekt entsprechend.

Achtung

Stellen Sie sicher, dass die Jumper 1 und 2 neben Port D **nicht** gesetzt sind. Sonst werden die Pins 1 und 2 von Port D zusätzlich mit der RS232-Schnittstelle auf dem Board verbunden, was zu Fehlfunktionen während der Zugriffe auf den externen RAM führen kann. Zusätzlich müssen auf dem Erweiterungsboard die Jumper bei `CS1` und `CS2` gesetzt sein, da der RAM-Baustein ansonsten nicht korrekt arbeitet.

Hinweis

Achten Sie darauf, dass Ihr Code wieder die volle Größe des externen RAM-Bausteins nutzt, falls Sie diese zur Simulation verkleinert hatten.

5.5.2 Kenndaten aller SPOS-Speicher

Entwerfen Sie ein Testprogramm um alle implementierten Speicherarten zu vergleichen. Zu der Analyse der Kenndaten gehört die Geschwindigkeit beim Schreiben und Lesen. Wie stark kann man die Delays im externen Speicher verkleinern bis die Funktionalität fehlerhaft wird? Messen Sie ebenfalls die Performanz der Speicherverwaltung die auf den jeweiligen Speichertreiber aufsetzt.

5.6 Pinbelegungen

Port	Pin	Belegung
PORTA	1	frei (schwer zugänglich)
	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 5.