

Praktikum Systemprogrammierung

Versuch 0

Grundlagen der Mikrocontrollerprogrammierung

Lehrstuhl Informatik 11 - RWTH Aachen

8. Oktober 2019

Inhaltsverzeichnis

0	Grundlagen der Mikrocontrollerprogrammierung	3
0.1	Versuchsinhalte	3
0.2	Lernziel	4
0.3	Grundlagen in der Verwendung des ATmega 644	4
0.3.1	Softwareentwicklung für Atmel-Mikroprozessoren	4
0.3.2	Informationen zum Evaluationsboard des ATmega 644	4
0.3.3	Konfiguration von Ports	6
0.3.4	Verkabelung des Evaluationsboards	7
0.4	Aufgaben	8
0.4.1	Allgemeines	8
0.4.2	Entwicklungsumgebung	8
0.4.3	Buttons und LEDs	8
0.4.4	Datentypen und LCD	12
0.4.5	Eigene Datentypen	18
0.4.6	Umgang mit Zeigern	21
0.4.7	Speicher des Mikrocontrollers	25
0.5	Pinbelegung	27

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

0 Grundlagen der Mikrocontrollerprogrammierung

Zum Einstieg in das *Praktikum Systemprogrammierung* werden in diesem Versuch die Grundlagen der Mikrocontrollerprogrammierung eingeführt. Mikrocontroller sind Halbleiterchips, die neben dem Prozessor weitere Peripheriefunktionen auf einem Chip vereinen. Dadurch kommen Projekte mit Mikrocontrollern oft mit wenigen Bauteilen aus. Sie sind häufig in alltäglichen Gegenständen zu finden und werden dann eingebettete Systeme genannt. Sie finden insbesondere in der digitalen Kontrolle und Steuerung von Geräten und Prozessen Anwendung. Ihre Leistung und Ausstattung werden für ihren jeweiligen Einsatzzweck angepasst, wodurch die Kosten, das Format und die Leistungsaufnahme optimiert werden können. Das Programmieren von Mikrocontrollern stellt, verglichen mit dem herkömmlicher Computersysteme, zusätzliche Anforderungen an die Entwickler. Dazu zählt, dass hardwarenahe Programmiersprachen wie C oder Assembler benutzt werden, sowie der Umgang mit Ressourcenknappheit.

0.1 Versuchsinhalte

Im Gegensatz zu kommenden Versuchen ist dieser Versuch nicht verpflichtend, um das Praktikum zu bestehen. Er soll den Einstieg in die Grundlagen der Mikrocontrollerprogrammierung erleichtern. Es wird mit der Vorstellung des im Praktikum verwendeten Evaluationsboards, des darauf befindlichen Mikrocontrollers ATmega 644 und dessen Verwendung, begonnen.

Zunächst wird am Beispiel von LEDs, Buttons und einem LCD gezeigt, wie Peripherie angesprochen werden kann. Zusätzlich wird der Gebrauch unterschiedlicher Datentypen bei der Mikrocontrollerprogrammierung vorgestellt. Darauf aufbauend wird die Erstellung von eigenen Datentypen und Datenstrukturen erläutert. Im letzten Teil dieses Dokuments wird der Umgang mit Zeigern, ein Sprachelement der Programmiersprache C, näher erläutert.

Bei Problemen bei der Lösung der Aufgabenstellungen in diesem Dokument kann das begleitende Dokument zum Praktikum Systemprogrammierung konsultiert werden, welches viele nützliche Informationen zur Softwareentwicklung mit dem ATmega 644 enthält.

0.2 Lernziel

Das Lernziel des Versuchs ist das Verständnis der folgenden Zusammenhänge:

- Umgang mit der Entwicklungsumgebung Atmel Studio 7
- Grundlagen der Programmiersprache C
- Programmierung eines Mikrocontrollers
- Eingabe mit Hilfe von Tastern und das Ansteuern von LEDs
- Kennenlernen verschiedener Datentypen und die Verwendung des LCDs
- Erstellung von eigenen Datentypen und Datenstrukturen
- Verwendung von Zeigern

0.3 Grundlagen in der Verwendung des ATmega 644

Im Folgenden werden grundlegende Informationen zur Softwareentwicklung für eingebettete Systeme vorgestellt. Anschließend wird näher auf das Evaluationsboard und den auf diesem platzierten Mikrocontroller ATmega 644 eingegangen.

0.3.1 Softwareentwicklung für Atmel-Mikroprozessoren

Software für Mikroprozessoren der Firma Atmel kann in verschiedenen Programmiersprachen entwickelt werden. Im Rahmen des Praktikums Systemprogrammierung wird ausschließlich die Sprache C mit einzelnen Assembleranweisungen verwendet.

Ein C Programm für einen Mikrocontroller unterscheidet sich nicht grundlegend von einem Programm für ein übliches Computersystem. Es wird sequenziell abgearbeitet und verfügt über eine sogenannte `main`-Methode, welche zu Beginn der Ausführung automatisch aufgerufen wird und den Programmeinstiegspunkt darstellt.

Um ein Programm auf den Mikrocontroller zu übertragen und zu debuggen wird ein sogenannter *JTAG Programmierer* verwendet. Als geeignete Entwicklungsumgebung stellt Atmel das Atmel Studio 7 kostenlos zur Verfügung.

0.3.2 Informationen zum Evaluationsboard des ATmega 644

Im Praktikum Systemprogrammierung wird ein 8-Bit Mikrocontroller der Firma Atmel verwendet – der ATmega 644. Dieser Mikrocontroller ist auf dem Evaluationsboard des Praktikums mit 20MHz getaktet und besitzt verschiedene interne Komponenten, wie zum Beispiel EEPROM-Speicher, AD/DA-Wandler, vier I/O-Ports, eine USART Schnittstelle und drei Timer. Darüber hinaus verfügt er über einen Interrupt-Controller. Das heißt, dass unabhängig vom aktuell ausgeführten Code Ereignisse erkannt und auf diese reagiert werden kann (d.h. ein *Interrupt* eintritt), indem die Ausführung des Hauptprogramms unterbrochen wird. Diese Ereignisse können extern auftreten, beispielsweise

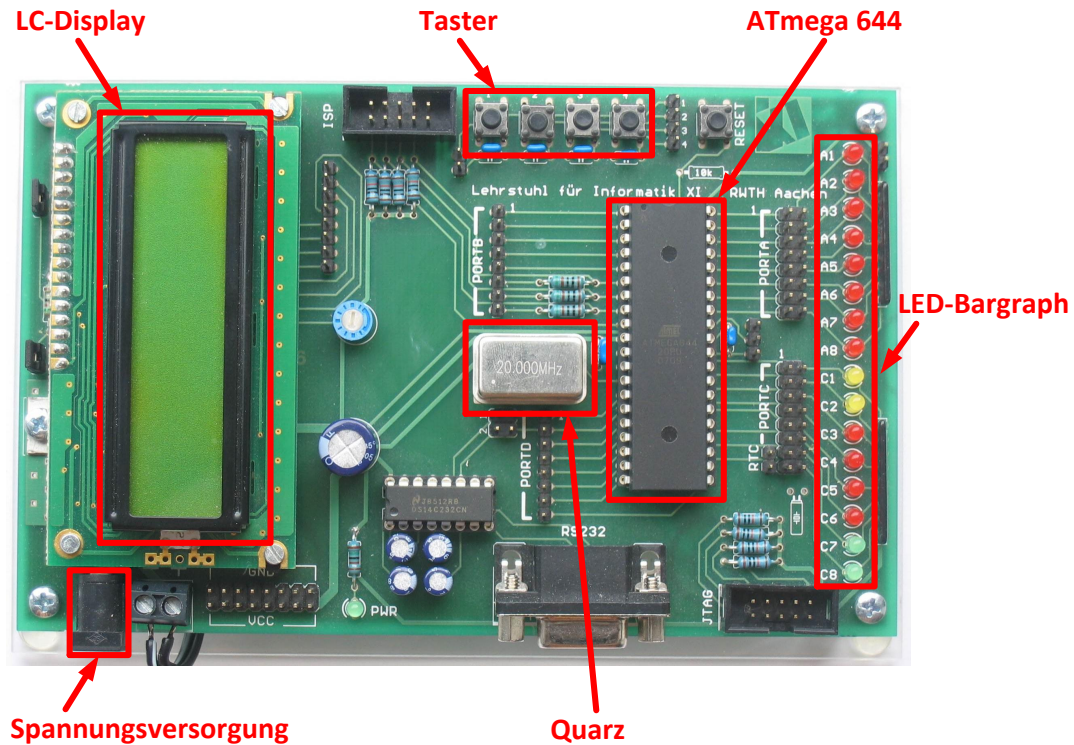


Abbildung 0.1: Das Evaluationsboard des ATmega 644

in Form einer Pegeländerung an einem Pin, oder intern, beispielsweise wenn ein spezielles Register einen bestimmten Wert erreicht hat.

Ein Evaluationsboard ist eine Platine, die es erlaubt, einen Mikrocontroller bequem programmieren und debuggen zu können, ohne zuerst eine Schaltung für diesen zu entwickeln. Das Evaluationsboard des Lehrstuhls für Informatik 11 der RWTH Aachen (Abbildung 0.1) bietet eine komfortable Arbeitsumgebung für den Umgang mit dem Mikrocontroller und stellt Pinleisten bereit, über die dieser leicht mit Eingabe- und Ausgabegeräten (LCD, Buttons, LEDs etc.) verbunden werden kann.

HINWEIS

Eine ausführlichere Beschreibung zu den Komponenten und deren Verwendung kann Kapitel 2.1: *Der Mikrocontroller ATmega 644* im *Begleitenden Dokument zum Praktikum Systemprogrammierung* entnommen werden.

0.3.3 Konfiguration von Ports

Der ATmega 644 verfügt über 40 Pins, von denen 32 als Ein-/Ausgabe Pins verwendet werden können. Ist ein Pin als Ausgang konfiguriert, so kann der Mikrocontroller die Spannung 5V (logisch 1) oder 0V (logisch 0) ausgeben. Mit Hilfe eines als Eingang konfigurierten Pins kann der Mikrocontroller feststellen, ob an diesem Pin ein Spannungspegel von 5V oder 0V angelegt wurde.

Die Pins sind auf vier Organisationseinheiten aufgeteilt: Die Ports A, B, C und D (siehe Abbildung 0.1). In diesen Organisationseinheiten befinden sich jeweils 8 Ein-/Ausgabe Pins, wobei diese von 0 bis 7 durchnummeriert sind. Jeder dieser Pins kann über spezielle Prozessorregister konfiguriert werden. Für jeden Port existieren sogenannte *Data Direction*-, *Port Output*- und *Port Input*-Register (DDR, PORT und PIN), welche der Konfiguration der Pins dienen. Diese können analog zu Variablen ausgelesen und verändert werden, wie im folgenden Abschnitt exemplarisch an Port A gezeigt wird.

Data Direction Register Das *Port A Data Direction Register* (DDRA) steuert, welche Pins eines Ports zur Ein- und welche zur Ausgabe verwendet werden. Eine 0 des least significant Bits, auch Stelle DDRA0 genannt (Analog dazu DDRA1 für das second-least significant Bit usw.), im Register DDRA bedeutet beispielsweise, dass der erste Pin von Port A als Eingang verwendet wird. Eine 1 an dieser Stelle konfiguriert den Pin als Ausgang.

Data Register Das *Port A Data Register* (PORTA) hat verschiedene Funktionen, abhängig davon, ob ein Pin durch das entsprechende Data Direction Register als Ein- oder als Ausgang definiert wurde. Wenn ein Pin als Ausgabepin definiert wurde, wird PORTA dazu verwendet, um zu steuern, welches Signal ausgegeben werden soll. Wenn ein Pin als Eingabepin definiert wurde, wird PORTA dazu verwendet, den Pullup-Widerstand für diesen Pin zu aktivieren oder zu deaktivieren.

Input Pins Das *Port A Input Pins-Register* PINA wird zum Auslesen externer Signale verwendet.

	PORT	PIN	DDR
Eingang	Pullup-Widerstand	anliegender Wert	0
Ausgang	zu schreibender Wert	-	1

Tabelle 0.1: Register

Eine ausführlichere Beschreibung der Register inklusive Beispiele kann im Kapitel 2.1.1: *I/O-Ports* im *Begleitenden Dokument zum Praktikum Systemprogrammierung* nachgeschlagen werden. Zur Konfiguration dieser Register ist die Manipulation einzelner Bits nötig. Dies wird mit Hilfe von Bitmasken erreicht. Bitmasken sind Zahlenwerte, die oft

in binärer oder hexadezimaler Schreibweise angegeben werden. Dabei werden führende Nullen mit angegeben, um die Größe des Ergebnisses zu verdeutlichen. Mit Hilfe binärer Operatoren wie `&` und `|` lassen sich dadurch einzelne Bits eines Registers verändern, ohne dass die restlichen Bits davon betroffen sind. Die Verwendung von Bitmasken ist in den Listings 1 und 2 veranschaulicht. Für weitere Informationen kann das Kapitel 5.1.16: *Registerzugriff, Bitshifting & Bitmasken* im *Begleitenden Dokument* herangezogen werden.

0.3.4 Verkabelung des Evaluationsboards

Auf den letzten Seiten dieses Dokuments ist eine Übersicht über die Pinbelegung des aktuellen Versuchs zu finden. Zudem ist zu beachten, dass die Mikrocontroller im Testpool, falls nicht ausdrücklich anders angegeben, stets gemäß dieser Tabelle verkabelt sind. In der folgenden Abbildung ist die Verkabelung für diesen Versuch schematisch dargestellt. Zu beachten ist, dass das LCD über Kreuz angeschlossen werden muss.

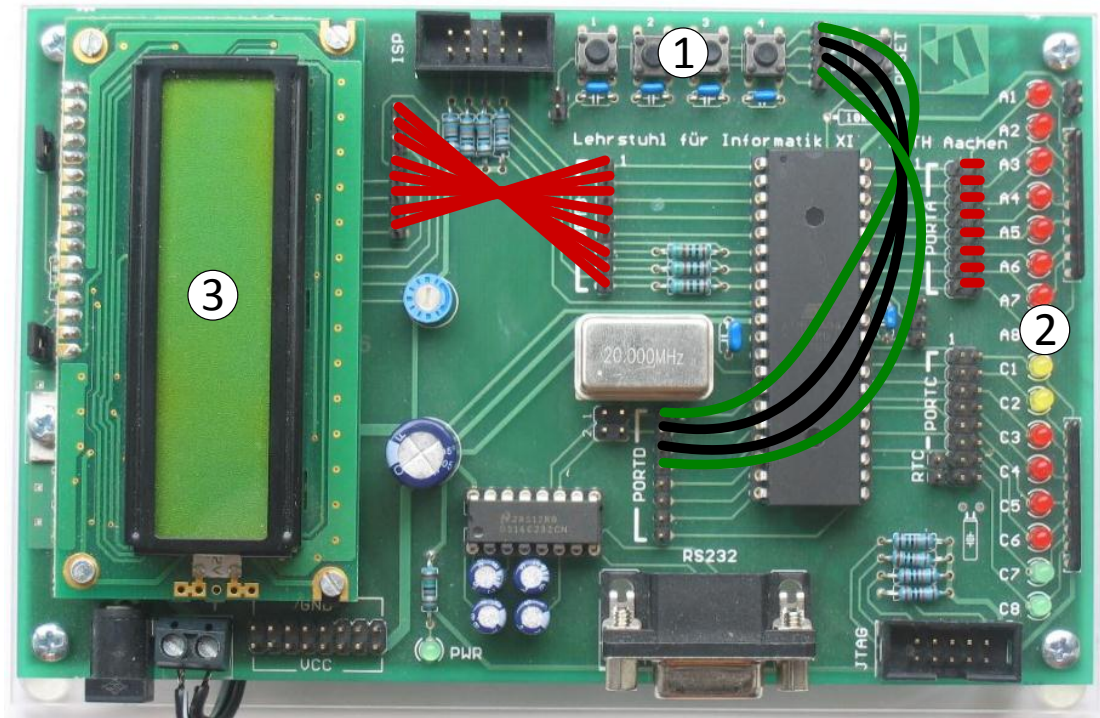


Abbildung 0.2: Die Verkabelung des Evaluationsboards sowie Buttons (1), LEDs (2), LCD (3)

0.4 Aufgaben

Implementieren Sie die in den nächsten Abschnitten beschriebenen Funktionalitäten. Halten Sie sich während der Implementierung an die hier verwendeten Namen und Bezeichnungen für Variablen, Funktionen und Definitionen. Beachten Sie außerdem die angegebenen Hinweise zur Implementierung.

0.4.1 Allgemeines

Für folgende Aufgaben können Sie das *Begleitende Dokument für das Praktikum Systemprogrammierung*, welches im Moodle-Lernraum des Praktikums zur Verfügung steht, als Hilfestellung nutzen. Dieses Dokument enthält alle Informationen, die Sie für den Einstieg in die C-Programmierung eines Mikrocontrollers benötigen. Es dient als grundlegendes Nachschlagewerk und beinhaltet eine Beschreibung aller für dieses Praktikum benötigten Grundlagen.

0.4.2 Entwicklungsumgebung

Im Rahmen dieses Praktikums wird Atmel Studio 7 als Entwicklungsumgebung verwendet. Hierbei handelt es sich um eine Entwicklungsumgebung, die einen Code-Editor, einen Compiler und einen Debugger zur Verfügung stellt. Des Weiteren enthält Atmel Studio 7 die sogenannte Atmel AVR Toolchain, welche die Programmierung des Mikrocontrollers in der Sprache C ermöglicht.

Wie Sie Atmel Studio 7 erhalten und darin ein Projekt erstellen, ist im *Begleitenden Dokument Kapitel 3: Einführung in Atmel Studio 7* nachzulesen. Für diesen Versuch wird Ihnen ein Codecodegrüst zur Verfügung gestellt, welches bereits ein vollständig konfiguriertes Projekt und die für den Versuch vorgegebenen Dateien enthält.

Im Begleitenden Dokument finden Sie darüber hinaus Informationen über den genauen Aufbau der Entwicklungsumgebung. Beispielsweise wird in Kapitel 3.3.5 die Debugging-Umgebung von Atmel Studio 7 vorgestellt, welche Sie für das Lösen der Aufgaben benötigen.

0.4.3 Buttons und LEDs

In diesem Aufgabenabschnitt wird das Ansprechen der Buttons und der LEDs vorgestellt. Ziel ist es, Eingaben mit den Buttons des Evaluationsboards tätigen zu können, diese zu erkennen und somit die LEDs des Evaluationsboards ein- bzw. auszuschalten. Wo die Buttons, LEDs und das LCD auf dem Board zu finden sind, können Sie der Abbildung 0.2 entnehmen.

Allgemein

Buttons Wie zuvor erwähnt, sollen in dieser Aufgabe die Buttons des Evaluationsboards verwendet werden, um Eingaben an ein auf dem Mikrocontroller laufenden Pro-

programm tätigen zu können. Das Evaluationsboard verfügt über vier Buttons, die beliebig mit den vorhandenen Pins verkabelt werden können.

Um den Zustand eines Buttons auslesen zu können, muss zunächst der entsprechende Pin als Eingang konfiguriert werden. Dabei ist es notwendig, sogenannte *Pullup*-Widerstände für die jeweils genutzten Pins zu aktivieren, um zwischen einem gedrückten und nicht-gedrückten Zustand unterscheiden zu können. Im Anschluss daran kann der Zustand des Buttons aus dem zugehörigen PIN-Register ausgelesen werden. Hierbei ist zu beachten, dass das korrespondierende Bit im PIN-Register im gedrückten Zustand den Wert 0 und im nicht-gedrückten Zustand den Wert 1 hat.

Die beiden folgenden Listings 1 und 2 sollen Ihnen bei der Konfiguration der Buttons als Hilfestellung dienen. Weitere Informationen hierzu können dem Kapitel 2.1.1: *Input Pins im Begleitenden Dokument zum Praktikum Systemprogrammierung* entnommen werden. Das folgende Beispiel zeigt, wie der Spannungspegel (5V oder 0V) des Pin D1 ausgelesen werden kann.

```
1 // Schreibweisen: 12 (dezimal) = 0b1100 (binär) = 0xC (
    hexadezimal)
2 // 1. Pin D1 als Eingang konfigurieren (Bit 1 von DDRD auf 0
    setzen)
3 DDRD  &= 0b11111101;
4
5 // 2. Pullup-Widerstand an Pin D1 aktivieren (Bit 1 von
    PORTD auf 1 setzen)
6 PORTD |= 0b00000010;
7
8 // 3. Pin D1 auslesen (Bit extrahieren und nach rechts
    verschieben)
9 uint8_t pinState = (PIND & 0b00000010) >> 1;
```

Listing 1: Zustand von Pin D1 wird abgefragt

Das nächste Beispiel zeigt die Verwendung von Pin D1 als Ausgang. Nach Ausführung des Codes liegt an Pin D1 eine Spannung von 5V an.

```
1 // 1. Pin D1 als Ausgang konfigurieren (Bit 1 von DDRD auf 1
   setzen)
2 DDRD  |= 0b00000010;
3
4 // 2. Logische 1 an Pin D1 ausgeben (Bit 1 von PORTD auf 1
   setzen)
5 PORTD |= 0b00000010;
```

Listing 2: Ausgabe einer logischen 1 an Pin D1

Übung: Implementierung des Buttonmoduls

Implementieren Sie das Modul zur Buttonabfrage in den bereits angelegten vier Funktionsrümpfen in der Datei `button.c`.

Die Funktion `initInput` soll PORTD konfigurieren, mit dem die Taster verbunden sind. Schalten Sie die entsprechenden Pins also auf Eingang. Zudem müssen die Pullup-Widerstände aktiviert sein. Achten Sie darauf, dass Sie zu Beginn ihres Programms die Funktion `initInput` einmal aufrufen, um die Buttons verwenden zu können.

Die Funktion `getInput` fragt die Zustände der Buttons ab. Verwenden Sie dazu `PIND`. Bedenken Sie, dass die Taster im nicht-gedrückten Zustand den Wert 1 haben.

Implementieren Sie anschließend die Funktionen `waitForInput` und `waitForNoInput`. Darin soll auf das Drücken eines Tasters bzw. auf das Loslassen aller Taster gewartet werden. Nutzen Sie die beiden Funktionen an geeigneten Stellen in Ihrem Projekt, um für einen geregelten Ablauf zu sorgen.

HINWEIS

Es empfiehlt sich mit Hilfe der Funktion `waitForNoInput` auf das Loslassen eines Taster zu warten, um irrtümliche Mehrfachbetätigungen zu umgehen.

Testen ihrer Implementierung

Um ihre Implementierung zu testen, rufen Sie zur Initialisierung zuerst die Funktion `initInput` in ihrer `main`-Funktion auf. Anschließend können Sie die zur Verfügung gestellte Funktion `buttonTest` benutzen.

LERNERFOLGSFRAGEN

- Welches Register wird wie verändert, um einen Port auf Eingang oder Ausgang schalten?
- Welche Operatoren werden wie verwendet, um einzelne Pins eines Ports auf V_{CC} oder GND zu ziehen?

LEDs Das Evaluationsboard des ATmega 644 verfügt am unteren Ende über 16 LEDs. eine solche Anordnung von LEDs wird auch als LED-Bargraph bezeichnet. Um die LEDs anzusprechen, muss die untere Reihe der zweireihigen Stiftleiste zur Verkabelung verwendet werden. Beachten Sie, dass, bedingt durch die Verschaltung der LEDs, diese genau dann leuchten, wenn am Ausgang des Mikrocontrollers ein Low-Pegel (0V) anliegt.

Übung: Ansteuerung des LED-Bargraphen

Erzeugen Sie eine Datei `led.c` sowie die dazugehörige Headerdatei `led.h`. Denken Sie daran, die Datei `button.h` mithilfe der `#include`-Direktive (siehe *Begleitendes Dokument Kapitel 5.1.2*) mit einzubeziehen, um ihre Buttonmethoden aus `led.c` aufrufen zu können. Erstellen Sie anschließend in `led.c` die zwei Funktionen `void led_init()` und `void led_fun()`. Mithilfe der Headerdatei sollen die beiden Funktion zudem für andere Dateien sichtbar gemacht werden.

Konfigurieren Sie in der Funktion `led_init` `PORTA`. Das bedeutet, dass der an die LEDs angeschlossene Port auf Ausgang geschaltet werden soll. Sorgen Sie außerdem dafür, dass alle LEDs zu Beginn ausgeschaltet sind. Vergessen Sie nicht, diese Funktion bei Programmbeginn einmal aufzurufen, um die LEDs verwenden zu können.

Die eigentliche Funktionalität wird in der Funktion `led_fun` implementiert. Das Betätigen von drei verschiedenen Buttons soll erkannt und darauf mit der jeweiligen Ansteuerung von LEDs reagiert werden.

Taster 1 Jeder Tastendruck soll den Zustand der ersten LED verändern. Nach dem Loslassen des Tasters soll diese leuchten, falls die LED zuvor ausgeschaltet war. War die LED schon eingeschaltet, so soll diese nach der Betätigung des Tasters aufhören zu leuchten. Wie Sie die Funktion umsetzen, steht Ihnen frei. Sie können mit Hilfe des XOR-Operators `^` versuchen, eine möglichst elegante Lösung zu finden.

Taster 2 Ein Drücken des Buttons soll alle acht LEDs um eine Stelle nach links verschieben (engl. *shift*). Benutzen Sie dazu den Shift-Operator `<<` bzw. `>>` (siehe *Kapitel*

5.1.13 im *Begleitenden Dokument*). Dadurch kann der Inhalt von Variablen bitweise verschoben werden. Leuchtet beispielsweise zu Beginn die erste Leuchtdiode, so soll nach dem Drücken des zweiten Tasters die zweite LED leuchten, während die erste LED ausgeschaltet wird. Zu beachten ist, dass kein Zustand der LEDs verloren gehen soll, sobald über die letzte LED hinausgeschiftet wurde. Das bedeutet, dass nach dem Drücken des Buttons der Zustand der letzten LED an die erste LED weitergegeben werden soll.

Taster 3 Wird der dritte Taster gedrückt, so soll der Zustand aller LEDs invertiert werden. Leuchtende LEDs sollen nicht mehr leuchten während zuvor ausgeschaltete LEDs beginnen zu leuchten. Für die binäre Invertierung existiert der Operator \sim (siehe *Kapitel 5.1.13 im Begleitenden Dokument*), den Sie zur Hilfe nehmen können.

Testen ihrer Implementierung

Rufen Sie zur Initialisierung zuerst die Funktionen `initInput` und `led_init` einmal auf. Anschließend können Sie `led_fun()` in ihrer `main`-Methode aufrufen. Beachten Sie, dass ihr Hauptprogramm nicht verlassen werden darf. Überprüfen Sie ebenfalls, ob alle benötigten Headerdateien eingebunden sind.

LERNERFOLGSFRAGEN

- Leuchten die LEDs, wenn der Ausgang des Mikrocontrollers einen Low-Pegel (0V), oder einen High-Pegel (5V) aufweist?
- Wie kann der XOR-Operator \sim für die Aufgabe des ersten Tasters verwendet werden?

0.4.4 Datentypen und LCD

Im folgendem Kapitel wird näher auf verschiedene Datentypen eingegangen, sowie der Ansteuerung des LCDs auf dem Evaluationsboard.

Allgemein

LCD Ansteuerung Zur Ansteuerung des LCDs steht Ihnen in den Dateien `lcd.c/.h` eine Treiber-Implementierung zur Verfügung. Diese bietet unter anderem folgende Funktionen:

Befehl	Bedeutung	Beispiel
<code>lcd_init(void)</code>	Initialisierung	<code>lcd_init();</code>
<code>lcd_clear(void)</code>	Löscht das Display. Das nächste Zeichen wird in der ersten Spalte und Zeile ausgegeben.	<code>lcd_clear();</code>
<code>lcd_writeChar(char)</code>	Gibt ein einzelnes Zeichen aus.	<code>lcd_writeChar('!');</code>
<code>lcd_writeDec(uint16_t)</code>	Gibt eine Ganzzahl ohne führende Nullen aus.	<code>lcd_writeDec(471);</code>
<code>lcd_writeString(const char*)</code>	Gibt eine Zeichenkette aus.	<code>lcd_writeString("xy");</code>

Tabelle 0.2: Ausgewählte Funktionen des LCD-Treibers

Weitere Funktionen können Sie im Header (`lcd.h`) des Treibers nachschlagen. Im *Praktikum Systemprogrammierung* sollen Strings mit Hilfe des Makros `PSTR()` im *Flash*-Speicher abgelegt werden. Das Makro kann wie folgt verwendet werden:

```

1 // Saving a String to Flash
2 const char* myString = PSTR("Dies ist mein String");
3
4 // Outputting a String to LCD
5 // With variable
6 lcd_writeProgString(myString);
7
8 // Directly without a variable
9 lcd_writeProgString(PSTR("Direkt ausgegebener String"))

```

Listing 3: Verwendung des Makros `PSTR()`

Datentypen Die Größe von Datentypen ist vom verwendeten Compiler und der Zielplattform abhängig. Im *Praktikum Systemprogrammierung* wird die Atmel AVR Toolchain benutzt. Tabelle 0.3 listet die verfügbaren Datentypen in dieser Umgebung mit ihrer Größe und ihrem Wertebereich auf. Da der verwendete Mikrocontroller nur über 4kB Arbeitsspeicher verfügt, empfiehlt es sich, auf große Datentypen wie `float` und `long` nach Möglichkeit zu verzichten.

Zudem haben Gleitkommatypen wie `float` den Nachteil, dass Variablen dieses Typs starken Rundungen unterliegen können und nicht auf allen Architekturen durch den Prozessor unterstützt werden. Gleitkommaoperationen sind im Vergleich zu ganzzahligen sehr aufwändig und erlauben nur Vergleiche auf Fast-Gleichheit innerhalb gewisser Schranken.

Datentyp	Spezifizierer	Größe in Bytes	Min. Wert	Max. Wert
void	-	-	-	-
char	int8_t	1	-128	127
unsigned char	uint8_t	1	0	255
short	int16_t	2	-32768	32767
unsigned short	uint16_t	2	0	65535
int	int16_t	2	-32768	32767
unsigned int	uint16_t	2	0	65535
long	int32_t	4	-2147483648	2147483647
unsigned long	uint32_t	4	0	4294967295

Tabelle 0.3: Datentypen in 16-Bit Architekturen

Die Eigenschaften des Datentyps `int` (Integer) hängen vom verwendeten System ab. Auf dem im Praktikum verwendeten Mikrocontroller ATmega 644 hat der Datentyp `int` die Größe 2 Byte, auf einer 32 Bit Architektur, wie etwa x86, aber die Größe 4 Byte. Um dies auszugleichen wird mit der `stdint.h` aus der *Standard C Library* eine Headerdatei mitgeliefert, welche es ermöglicht, Datentypen fester Größe zu benutzen.

Der Typ `void` kann nicht instanziiert werden, sondern wird für Zeiger unbekannten oder beliebigen Typs verwendet. Der Umgang mit Zeigern wird Ihnen im letzten Teil dieses Dokuments vorgestellt.

Vorbereitung

Um das LCD Display verwenden zu können, muss einmalig die Funktion `lcd_init()` aufgerufen werden, um das LCD nutzen zu können.

Übung: Wertebereich von Datentypen

In dieser Aufgabe werden die Wertebereiche und der Unterschied zwischen *signed* und *unsigned* Datentypen genauer betrachtet. Erstellen Sie dazu die Funktion `void loop()` in der Datei `datatypes.c` und eine entsprechende Deklaration in `datatypes.h`. Legen Sie in der Funktion folgende for-Schleife an:

```

1  uint8_t result = 0;
2  for (uint8_t i = 5; i >= 0; i--){
3      result += i * 2 + 2;
4  }
5  lcd_writeProgString(PSTR("Loop finished:"));

```

Listing 4: Inhalt der Funktion `loop()` in `datatypes.c`

Geben Sie anschließend mithilfe von `lcd_writeDec` den Wert der Variable `result` auf dem LCD aus. Erwarten Sie eine Ausgabe, nachdem Sie die Funktion `loop()` in ihrem Hauptprogramm aufgerufen haben?

Sie können zur Ermittlung des Problems *Breakpoints* sowie die Variablenüberwachung durch die *Watch List* nutzen. Dabei fällt auf, dass die Variable `i` von Atmel Studio 7 als *Invalid Location* angegeben wird. Dies liegt daran, dass der Compiler diesen Codeabschnitt optimiert hat und die Variable `i` nicht mehr im Arbeitsspeicher angelegt wird, sondern nur in einem Register liegt. Um die Variable dennoch während des Debuggens einsehen zu können, bietet sich eine der folgenden Vorgehensweisen an:

- Die Variable temporär als `volatile` deklarieren (`volatile uint8_t i`)
- Die Variable temporär in einer globalen Debugvariable ablegen
- Die Compileroptimierung niedriger stellen

Denken Sie daran, die Änderungen nach dem Debuggen wieder rückgängig zu machen, da die Effizienz des Programms abnimmt und es nicht garantiert ist, dass die Vorgehensweisen Nebeneffektfrei sind. Wie Sie genau Variablen mithilfe der Watch List überwachen, ist im Kapitel 6.2.3: *Überwachung des Speichers unter Variablenüberwachung im Begleitenden Dokument* beschrieben. Eine Anleitung zum Setzen von Breakpoints finden Sie im Kapitel 6.2.1 *Überwachen der Programmausführung im Begleitenden Dokument*.

LERNERFOLGSFRAGEN

- Wie viele Iterationen benötigt die for-Schleife, um verlassen zu werden?
- Wie muss der Datentyp von `i` verändert werden, damit das gewünschte Verhalten erreicht wird?

Übung: Zuweisung von Datentypen unterschiedlicher Größe

In dieser Aufgabe wird die Zuweisung zwischen verschiedenen Datentypen und daraus eventuell resultierenden Problemen näher betrachtet. Erstellen Sie die Funktion `void convert()` in der Datei `datatypes.c` und eine entsprechende Deklaration in `datatypes.h`. Legen Sie außerdem in der Funktion folgende zwei Variablen an:

```
uint8_t target = 200;  
uint16_t source = target + 98;
```

Schauen Sie sich mithilfe der Watch List den Inhalt beider Variablen an. Welche Werte haben sie jeweils?

Weisen Sie nun `target` den Wert von `source` zu:

```
target = source;
```

Schauen Sie sich erneut den Wert beider Variablen an. Haben beide Variablen den Wert, den Sie erwarten? Warum haben Sie diesen Wert?

LERNERFOLGSFRAGEN

- Warum gibt es verschiedene Datentypen für ganzzahlige Zahlen? (`uint8_t`, `uint16_t`, etc.)
- Was passiert, wenn eine Variable vom Typ `uint8_t` ein Wert vom Typ `uint32_t` zugewiesen wird?

Übung: Explizite Typumwandlung (Cast)

In diesem Abschnitt wird vorgestellt, wie Variablen eines bestimmten Datentyps in einen anderen Typ umgewandelt werden. Eine schlichte Zuweisung von Variablen unterschiedlichen Typs führt häufig zu ungewolltem Verhalten, wie in der vorherigen Aufgabe bereits demonstriert wurde.

Um eine explizite Typumwandlung durchzuführen, muss während der Zuweisung der gewünschte Datentyp in Klammern vor die Variable geschrieben werden. Weitere Informationen erhalten Sie im *Kapitel 5.1.6 Typecasts* im *Begleitenden Dokument*.

```

1 // Variablen unterschiedlichen Typs
2 int8_t var1 = 0;
3 uint8_t var2 = 130;
4
5 // Zuweisung von Variablen unterschiedlichen Typs
6 var1 = var2;
7
8 // Typumwandlung
9 var1 = (int8_t)(var2);

```

Listing 5: Beispiel für explizite Typumwandlung

Erstellen Sie anschließend die Funktion `void shift()` in der Datei `datatypes.c` und eine entsprechende Deklaration in `datatypes.h`. Wir möchten in dieser Funktion eine 32-Bit Variable namens `result` definieren, deren höchstwertiges Bit gesetzt ist (d.h. den Wert 1 hat). Danach soll der Wert von `result` mithilfe der vorgegebenen Funktion `lcd_write32bitHex` auf dem LCD ausgegeben werden. Übertragen Sie dazu folgenden Code in ihr Projekt:

```

1 uint32_t result = 1 << 31;
2 lcd_write32bitHex(result);

```

Listing 6: Inhalt der Funktion `shift()` in `datatypes.c`

Lassen Sie sich die Variable `result` ausgeben. Die Ausgabe sollte `0x00000000`, also 0, sein. Dies liegt daran, dass der Compiler alle Zahlen im Quellcode implizit als `uint16_t` interpretiert.

```

1 uint32_t result = (uint16_t)(1 << 31);
2 lcd_write32bitHex(result);

```

Listing 7: Interpretation des Compilers

Wird aber eine `uint16_t` Variable um 31 Stellen geshiftet, liegen alle Daten außerhalb der 16 Bit und werden somit verworfen. Der Rest der Variable wird mit Nullen aufgefüllt. Um den Fehler zu vermeiden, muss dem Compiler also mitgeteilt werden, dass die 1 mehr Bits hat. Dies ist durch das bereits eingeführte Casten möglich:

```
1 uint32_t result = (uint32_t)1 << 31;  
2 lcd_write32bitHex(result);
```

Listing 8: Korrekte Implementation

LERNERFOLGSFRAGEN

- Können Sie mit Hilfe der expliziten Typumwandlung das Problem der vorherigen Aufgabe lösen (Zuweisung von `source` zu `target`)?

0.4.5 Eigene Datentypen

Die Programmiersprache C bietet unterschiedliche Möglichkeiten an, eigene Datentypen und Datenstrukturen zu definieren. Im folgenden Abschnitt werden einige der Möglichkeiten vorgestellt.

Vorbereitung

Legen Sie die Datei `structures.c` sowie eine dazu passende Headerdatei an. Inkludieren Sie in der Datei `structures.c` folgende Headerdateien: `structures.h`, `lcd.h`, `button.h` und `stdint.h` und in der `main.c` die Headerdatei `structures.h`.

Enum

Mit der Anweisung `enum` wird ein Aufzählungstyp definiert (siehe Listing 9). Aufzählungstypen haben einen definierten Wertebereich und führen durch ihre eindeutige Benennung der Elemente zu deutlich besser lesbarem Quelltext. Da `enums` intern in primitive Datentypen übersetzt werden, ist es möglich, eine manuelle Zuordnung vorzunehmen (siehe Listing 10).

```
1 enum Condition {  
2     EXCELLENT,  
3     GOOD,  
4     POOR,  
5     BAD,  
6 };  
7  
8 enum Condition conditionOfItem = GOOD;
```

Listing 9: Definition und Nutzung eines `enum` Datentyps

```
1 enum Condition {
2     EXCELLENT,           // == 0
3     GOOD,               // == 1
4     //Explizite Wertzuweisung an POOR
5     POOR = 7,           // == 7
6     BAD,                // == 8
7 };
```

Listing 10: Manuelle Zuweisung von Werten in einem `enum`

Legen Sie in der von Ihnen erstellen C-Datei ein `enum` mit dem Namen `DistributionStatus` und den Werten `AVAILABLE`, `BACKORDER` und `SOLD_OUT` an. Setzen Sie nun den Wert von `SOLD_OUT` auf 99.

Strukturen

Die Anweisung `struct` erzeugt einen aus anderen Datentypen zusammengesetzten Datentyp (siehe Listing 11). Durch die Bündelung zusammengehöriger Variablen erlaubt dies eine bessere Lesbarkeit des Quellcodes.

Implementieren Sie mit dem Schlüsselwort `struct` eine Struktur mit dem Namen `ArticleNumber`. Dieses `struct` soll folgende zwei Attribute vom Datentyp `uint8_t` besitzen: `manufactureId` und `productID`.

```
1 struct Employee {
2     uint8_t id;
3     uint16_t salary;
4 };
5
6 //Deklaration einer Variable maxMustermann mit id = 12 und
7 //salary = 2000
8 struct Employee maxMustermann = {12, 2000};
9 maxMustermann.salary = 3000; //setze salary auf 3000
```

Listing 11: Definition und Nutzung eines `struct`

Typnamen

Mithilfe der Anweisung `typedef` können neue Datentypen auf Basis anderer Datentypen definiert werden (siehe Listing 12). Dies lässt den Quellcode deutlich lesbarer erscheinen. Typnamen (eng. **Typedefs**) können aber auch bei `enums`, `structs` und `unions` hilfreich sein, um die entsprechenden Schlüsselwörter zu sparen.

Legen Sie ein `typedef` für das `enum DistributionStatus` an. Ändern Sie anschließend die Deklaration von `struct ArticleNumber` wie in Listing 12 ab.

```
1  typedef uint8_t Alter;
2
3  Alter max = 12;
4
5  typedef struct {
6      uint8_t id;
7      uint16_t salary;
8  } Employee;
9
10 Employee maxMustermann = {12, 2000};
```

Listing 12: Definition eines `typedef` und Anwendung

Vereinigung

Vereinigungen (eng. **Unions**) ermöglichen das Zusammenfassen von verschiedenen Datentypen. Dabei befinden sich alle Attribute des Unions an derselben Stelle im Speicher, sodass ein Union immer so groß ist wie sein größtes Element. Dies führt dazu, dass die Elemente in einem Union nicht unabhängig voneinander geändert werden können. Unions werden vor allem genutzt um Typen umzuinterpretieren oder um eingelesene Daten unterschiedlich interpretieren zu können. Sie können explizite Typecasts im Code ersparen.

Erstellen Sie ein Union mit dem Namen `FullArticleNumber` und nutzen Sie dabei ein `typedef`. Fügen Sie das `uint16_t` Attribut `combinedNumber` und das `ArticleNumber` Attribut `singleNumbers` ein.

```

1 union IPAddress {
2     uint8_t block[4];
3     uint32_t ip;
4     uint8_t firstBlock;
5 }
6
7 union IPAddress localhost = {{127, 0, 0, 1}};
8 union IPAddress copyIp.ip = localhost.ip;
9 //copyIp.firstBlock == 127
10 //copyIp.block == {127, 0, 0, 1}

```

Listing 13: Definition und Implementierung eines union

Übung: struct, typedef und union

Legen Sie ein weiteres `struct Article` mit folgenden Attributen an:

```

1 FullArticleNumber articleNumber;
2 DistributionStatus status;

```

Listing 14: Attribute der Struktur Article

Implementieren Sie nun eine Funktion `displayArticles` und rufen diese in der `main()` auf. Legen Sie dazu zunächst einen Array des Typs `Article` an und füllen dies mit folgenden Werten:

```

{{0x1101}, AVAILABLE}
{{0x1110}, BACKORDER}
{{0x0101}, SOLD_OUT}

```

Beachten Sie dabei die Byte-Reihenfolge. Aufgrund der Little-Endian Architektur wird das Byte mit den niederwertigen Bits an der kleinsten Speicheradresse gespeichert. Dadurch aufgrund der Attributreihenfolge im `struct ArticleNumber` bei einer `ArticleNumber` von `0x1101` die `11` die `productID`, und die `01` die `manufacturerID`.

Geben Sie alle Artikel nacheinander auf dem LCD aus, wobei immer zum nächsten gewechselt werden soll, wenn ein Button gedrückt wurde. Geben Sie `manufacturerID` und `productID` getrennt voneinander aus, sowie den `DistributionStatus` in lesbarer Form.

0.4.6 Umgang mit Zeigern**Allgemein**

Um Ihnen den Umgang mit *Zeigern* (engl. *Pointer*) in dieser Aufgabe zu erleichtern, empfiehlt es sich die Kapitel 5.1.7: *Zeiger* und 5.1.8: *Funktionszeiger* im *Begleitenden*

H	e	r	s	t	e	l	l	e	r	:		0	1		
A	r	t	#		0	1		L	a	g	e	r			

Abbildung 0.3: Beispielausgabe von `manufacturerID`

Dokument als Hilfestellung zu benutzen. Aufbauend auf den dort beschriebenen Grundlagen dient diese Aufgabe dazu, den Umgang mit Zeigern zu erlernen.

Zeiger haben in der C-Programmierung eine besondere Bedeutung. Sie sind Variablen, die eine Adresse einer anderen Variablen beinhalten. Diese Adresse wird je nach genutzten Datentyp entsprechend interpretiert. Auch gibt es die Möglichkeit, vorhandenen Programmcode mittels Funktionszeiger auszuführen, indem der Programmzähler auf die Adresse des Zeigers gesetzt wird.

In dieser Aufgabe ist es erforderlich mit Hilfe der Funktion `malloc` Speicher zu allozieren. Die Verwendung der Funktion wird in Listing 15 an einem Beispiel verdeutlicht und soll Ihnen bei der kommenden Aufgabe helfen.

```

1 // 1. Zeiger auf einen Speicherbereich an Adresse 270 (gemäß
  // Rückgabewert von malloc):
2 uint8_t *position = malloc(2 * sizeof(uint8_t));
3 uint8_t offset = 1;
4
5 // 2. Beschreiben der Bytes an der Speicherstellen 270 mit
  // dem Wert 100:
6 *position = 100;
7
8 // 3. Beschreiben der Bytes an der Speicherstellen 271 mit
  // dem Wert 200.
9 *(position + offset) = 200;
10
11 // 4. Auslesen des Wertes der Speicherstellen 270:
12 uint16_t value = *(position); // value = 100

```

Listing 15: Verwendung von `malloc`

Aufgaben

In der Datei `pointer.c` befindet sich die Funktion `learningPointers`, die Sie vervollständigen sollen. Rufen Sie die Funktion `learningPointers` an einer geeigneten Stelle auf, um ihre Implementierung zu testen. Denken Sie ebenfalls daran, die entsprechende

Headerdatei `pointers.h` zu inkludieren. Machen Sie sich anschließend mit der Funktion `learningPointers` und deren Hilfsfunktionen vertraut und lösen Sie die folgenden sieben Teilaufgaben.

1. Allozieren von Speicher Allozieren Sie mithilfe der Funktion `malloc` aus der Standardbibliothek `stdlib.h` einen Speicherbereich von 6 Byte. Überlegen Sie sich dazu, wie viel Speicherplatz Sie dafür benötigen. Ist die Allokation erfolgreich, so ist der Rückgabewert von `malloc` die erste Adresse des allozierten Speicherbereiches. Weisen Sie der schon vorhandenen Zeigervariablen `memory` diese Startadresse zu. Sollte die Allokation fehlschlagen, so gibt `malloc` die Adresse 0 zurück, was entsprechend abgefangen werden sollte.

Da Sie hier mit Adressen des SRAMS arbeiten, können Sie das Memoryfenster (siehe *Kapitel 6.2.3: Überwachung des Speichers unter Anzeige des Speichers im Begleitenden Dokument*) zur Hilfe nehmen. Dadurch erhalten Sie einen direkten Einblick in den Speicher des Mikrocontrollers. Dies kann in dieser Aufgabe und in späteren Versuchen bei der Lösung von Problemen oder dem Nachvollziehen von Programmabläufen helfen.

2. Dereferenzierung und Arithmetik mit Zeigervariablen Der angelegte Speicherbereich soll nun mithilfe des vorhandenen Arrays `alphabet` mit sinnvollen Werten gefüllt werden. Schreiben Sie dazu den Wert des letzten Elements des Arrays an das Ende des allozierten Speicherbereiches `memory`. Dazu müssen Sie, basierend auf der Startadresse `memory`, die erforderliche Speicheradresse ermitteln und mittels des Dereferenzierungsoperators `*` auf den Inhalt zugreifen. Die benötigte Zeigerarithmetik, um an bestimmte Adressen zu gelangen, ist in Listing 15 veranschaulicht.

LERNERFOLGSFRAGEN

- Welche Funktion hat der Operator `*` bei Zeigern zusätzlich zur Beschreibung von Speicheradressen?

3. Deklaration und die Operatoren `*` und `&` In diesem Schritt nutzen Sie die bereits vorimplementierte Variable `charVar` mit Inhalt `'A'`. Erstellen Sie nun einen Zeiger, dem Sie die Adresse der oben genannten Variablen übergeben. Dazu benötigen Sie den Adressoperator `&`, welcher die Adresse zurückliefert, an der eine Variable im Speicher abgelegt ist. Verändern Sie nun wieder mithilfe des Dereferenzierungsoperators `*` den Inhalt an der Adresse des von Ihnen erstellten Zeigers so, dass dieser den Buchstaben I aus dem Array `alphabet` erhält.

LERNERFOLGSFRAGEN

- Welchen Wert besitzt die Variable `charVar`? Warum wurde dieser ohne neue Definition von `charVar` verändert?
- Gibt es eine Möglichkeit den Wert von `charVar` nachträglich zu verändern, wenn `charVar` als Konstante deklariert wurde?

4. Call-By-Value und Call-By-Reference Machen Sie sich mit den implementierten Funktionen `callByValue` und `callByReference` vertraut und rufen Sie diese an den vorgesehenen Stellen im Code auf. Als Übergabeparameter dient das erste Element des Arrays `alphabet`.

LERNERFOLGSFRAGEN

- Welches Verhalten können Sie beobachten? Können Sie das aufgetretene Verhalten erklären?
- Ist es möglich, der Funktion `callByReference` das erste Element von `alphabet` zu übergeben, ohne den Adressoperator `&` zu benutzen?

5. Zeiger auf Zeiger Es ist ebenfalls möglich Zeiger zu definieren, welche auf Zeiger zeigen. Erstellen Sie einen Zeiger `ppChar`, der die Adresse des von Ihnen erstellten Zeigers aus Schritt 3 erhält. Weisen Sie daraufhin dem Zeiger aus Schritt 3 die Array-Variable `alphabet` zu. Füllen Sie anschließend mithilfe der Zeigervariablen `ppChar` die vorletzte Speicheradresse des Speicherbereiches `memory`. Beachten Sie dabei, dass der Inhalt von `ppChar` einen Zeiger beinhaltet, weswegen ein einzelner `*`-Operator nicht ausreicht.

LERNERFOLGSFRAGEN

- Wie sieht die Deklaration von `ppChar` aus?
- Warum kann `alphabet` einer Zeigervariable zugewiesen werden?

6. Zeiger auf Strukturen Während es in den vorherigen Schritten um Zeiger auf einfache Datentypen ging, wird hier gezeigt, wie Zeiger auf Strukturen verwendet werden. Eine simple Struktur `sFirst` wurde bereits implementiert. Außerdem wurde bereits eine Instanz `sfoo` dieser Struktur erzeugt. Erstellen Sie zusätzlich eine weitere Struktur `sSecond`, welche als Attribut einen Zeiger auf die erste Struktur enthält. Legen Sie nachträglich eine Instanz der Struktur `sSecond` namens `pStruct` an und initialisieren Sie das Attribut mit der Instanz `sfoo` der ersten Struktur. Greifen Sie anschließend mit Hilfe von `pStruct` auf das Attribut `bar` der ersten Struktur zu und schreiben Sie den Inhalt an den Anfang ihres Speicherbereiches `memory`.

Anders als im vorherigen Kapitel existiert in dieser Aufgabe ein Zeiger auf eine Struktur. Um Zugriff auf Strukturelemente zu erhalten, benötigt man den Pfeiloperator `->` anstatt des Punktoperators.

```

1 //Pfeilnotation
2 ZeigerAufStruktur->Attribut
3 //Punktnotation
4 Struktur.Attribut

```

Listing 16: Zugriff auf Strukturen


LERNERFOLGSFRAGEN

- Um über einen Zeiger Zugriff auf Strukturelemente zu erhalten, benötigt man den Pfeiloperator anstatt des Punktoperators. Ist es dennoch möglich, bei Zeigern lediglich den Punktoperator zu verwenden?

7. Funktionszeiger Machen Sie sich mit der bereits implementierten Funktion `charFunction` vertraut und erstellen Sie einen Funktionszeiger, der auf eben genannte Funktion zeigt. Tätigen Sie anschließend mit Hilfe des Funktionszeigers einen Funktionsaufruf. Der Rückgabewert der Funktion soll an die drittletzte Position des zu Anfang allozierten Speicherbereiches `memory` geschrieben werden.

Während der einzelnen Schritte werden verschiedene Meldungen auf dem LCD ausgegeben. Dies soll Ihnen beim Verständnis und der Überprüfung auf Korrektheit helfen. Die Meldungen werden solange angezeigt, bis ein beliebiger Taster gedrückt wird. Anschließend wird ein Lösungswort angezeigt.

0.4.7 Speicher des Mikrocontrollers

Atmel Studio 7 bietet neben der *Watchlist* auch die Möglichkeit, den Speicher des Mikrocontrollers direkt zu inspizieren. Über das Memory Fenster () können der *Flash*, in dem das Programm gespeichert ist und der Arbeitsspeicher des Mikrocontrollers (*SDRAM*)

ausgelesen werden. Besonders die Inspektion des SDRAMs kann bei der Suche nach Fehlern im Programm hilfreich sein, da in diesem Daten des Programms abgelegt werden. Im Memory Fenster kann dieser in der *Memory* Dropdown-Liste als *data IRAM* ausgewählt werden.

Aufgabe

Die Datei `pointer.c` enthält eine Funktion `inspectMe()`. Diese schreibt ein Lösungswort an die Speicherstelle `0x200` des SDRAMs. Setzen sie einen Breakpoint an der markierten Stelle der Funktion und nutzen sie das Memory Fenster, um den Speicher an Adresse `0x200` einzusehen und das Lösungswort zu finden.

0.5 Pinbelegung

Port	Pin	Belegung
Port A	A0	LED 1
	A1	LED 2
	A2	LED 3
	A3	LED 4
	A4	LED 5
	A5	LED 6
	A6	LED 7
	A7	LED 8
Port B	B0	LCD Pin 1 (D4)
	B1	LCD Pin 2 (D5)
	B2	LCD Pin 3 (D6)
	B3	LCD Pin 4 (D7)
	B4	LCD Pin 5 (RS)
	B5	LCD Pin 6 (EN)
	B6	LCD Pin 7 (RW)
	B7	frei
Port C	C0	frei
	C1	frei
	C2	Reserviert für JTAG
	C3	Reserviert für JTAG
	C4	Reserviert für JTAG
	C5	Reserviert für JTAG
	C6	frei
	C7	frei
Port D	D0	Button 1: Enter
	D1	Button 2: Down
	D2	Button 3: Up
	D3	Button 4: ESC
	D4	frei
	D5	frei
	D6	frei
	D7	frei

Pinbelegung für die Aufgaben zum ATmega 644.