



Informatik 11
Embedded Software

RWTH AACHEN
UNIVERSITY

Praktikum Systemprogrammierung

Begleitendes Dokument

Lehrstuhl Informatik 11 - RWTH Aachen

26. Oktober 2020

Inhaltsverzeichnis

1 Allgemeines	5
1.1 Inhalt des begleitenden Dokuments	5
1.2 Programmierrichtlinien	5
2 Hardware	7
2.1 Der Mikrocontroller ATmega 644	7
2.1.1 I/O-Ports	7
2.1.2 Interrupts	9
2.1.3 Timer	10
2.2 Das Evaluationsboard	11
2.2.1 Funktionen des Evaluationsboards	12
2.2.2 Ansteuerung des LCD	12
3 Einführung in Atmel Studio	15
3.1 Verwendete Software	15
3.2 Verwendung von Atmel Studio	16
3.2.1 Organisation der Anwendung	16
3.3 Aufbau der IDE	16
3.3.1 Erstellen einer Solution mit einem Projekt	18
3.3.2 Konfigurieren der Toolchain	18
3.3.3 Ein Programm kompilieren und starten	19
3.3.4 Debugging	20
3.4 Compiler Optimierungen	21
4 Benutzung des Testpools	25
4.1 Remotedesktopverbindung beantragen	25
4.2 Remotedesktopverbindung herstellen	27
4.3 Auf Remotedesktop arbeiten	28
4.4 Teamnetzlaufwerk mounten	29
5 Einführung in die C Programmierung	31
5.1 Syntax und Semantik von C	32
5.1.1 Struktur eines C-Programms	32
5.1.2 Der Präprozessor	33
5.1.3 Kommentare	37
5.1.4 Datentypen	38
5.1.5 Deklaration und Initialisierung von Variablen	39
5.1.6 Typecasts	42

Inhaltsverzeichnis

5.1.7	Zeiger (Pointer)	43
5.1.8	Funktionszeiger	45
5.1.9	Arrays	46
5.1.10	Zeichenketten (Strings)	48
5.1.11	Eigene Datentypen erstellen	49
5.1.12	Type qualifiers	53
5.1.13	Operatoren	56
5.1.14	Funktionen	61
5.1.15	Kontrollstrukturen	63
5.1.16	Registerzugriff, Bitshifting & Bitmasken	67
5.1.17	Inline Assembler	70
5.1.18	Zufallszahlen	71
5.2	Strukturverbesserungen	71
5.2.1	Kommentare	72
5.2.2	Umgang mit Funktionen und Kontrollstrukturen	73
5.2.3	Umgang mit Daten	74
5.3	Quelltextkonventionen	82
5.3.1	Dateien	83
5.3.2	Kommentare	83
5.3.3	Bezeichner	84
5.3.4	Definitionen und Konstanten	84
5.3.5	Klammern	85
5.3.6	Anweisungen	85
5.3.7	Casten von Variablentypen	85
6	Hinweise zum Debuggen	88
6.1	Was ist Debuggen?	88
6.1.1	Allgemeines Vorgehen beim Debuggen	88
6.1.2	Debugger	89
6.2	Debugging-Methoden	89
6.2.1	Überwachen der Programmausführung	89
6.2.2	Disassembler	93
6.2.3	Überwachung des Speichers	94
6.2.4	Nicht überwachbare Funktionalität	99
6.3	Probleme beim Debugging	100
6.3.1	Probleme bei der Programmüberwachung	101
6.3.2	Probleme bei der Speicherüberwachung	103
6.4	Fallbeispiele aus dem Praktikum Systemprogrammierung	105
6.4.1	Fehler durch falsche Datentypen	105
6.4.2	Unerwartetes Verhalten durch Optimierung	107
7	Dokumentation mit Doxygen	111
7.1	Doxygen im Praktikum	111
7.2	Ausgabe von Doxygen	111

Inhaltsverzeichnis

7.3	Verwendung von Doxygen	114
7.3.1	Konfiguration	114
7.3.2	Erzeugen der Dokumentation	114
7.4	Doxygen-Kommentare	116
7.4.1	Grundlagen	116
7.4.2	Kommentieren von Funktionen	118
7.4.3	Kommentieren von Dateien	118
7.4.4	Spezielle Tags	120
8	Weiterführende Literatur	121
8.1	AVR-Mikrocontroller	121
8.2	C-Programmierung	121
8.3	Doxygen	121

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.

1 Allgemeines

Dieses Dokument ist eine Hilfe für das Erarbeiten der Aufgaben des Praktikum Systemprogrammierung. In den folgenden Kapiteln werden nützliche Hinweise gegeben und Richtlinien beschrieben, die im Verlauf des Praktikums befolgt werden müssen.

1.1 Inhalt des begleitenden Dokuments

Kapitel 2, „Hardware-Vorstellung“, stellt die im Praktikum verwendete Hardware vor. Es handelt sich um einen ATmega 644 der Firma Atmel, welcher auf einem Evaluationsboard platziert ist.

Kapitel 3, „Einführung in Atmel Studio“, erläutert den Installationsprozess und die Verwendung der Entwicklungsumgebung des Mikrocontrollers.

Kapitel 4, „Benutzung des Testpools“, stellt den Testpool vor und erläutert wie man diesen benutzt.

Kapitel 5, „Einführung in die C Programmierung“, gibt wichtige Hinweise für den Umgang mit der Programmiersprache C. Je nach Vorwissen ist es nicht nötig dieses Kapitel vollständig zu lesen. Die Lernerfolgsfragen am Ende jedes Abschnitts sind ein guter Indikator dafür, ob das Wissen des Abschnitts bereits verstanden wurde.

Kapitel 6, „Hinweise zum Debuggen“, erklärt den Vorgang des Debuggens und stellt die Tools vor, welche Atmel Studio bereitstellt, um den Programmierer bei der Fehlersuche zu unterstützen.

Kapitel 7, „Dokumentation mit Doxygen“, stellt das Programm Doxygen vor, welches direkt aus dem Quellcode Dokumentationen erstellen kann.

1.2 Programmierrichtlinien

Um die Lesbarkeit des Codes zu erhöhen und den Aufwand zur Fehlersuche während des Versuches zu minimieren, müssen folgende Richtlinien eingehalten werden. Am Ende jedes Punktes findet sich ein Verweis auf Stellen innerhalb dieses Dokuments, welche Tipps für die Umsetzung der entsprechenden Richtlinien enthalten.

- Der Programmcode des Projekts soll sinnvoll modularisiert sein. Insbesondere sollte eine Trennung von Interface und Implementierung stattfinden. Dies geschieht mit .c und .h Dateien unter Verwendung der bereitgestellten Dateien `atmega644constants.h`, `defines.h`, um das Projekt übersichtlich zu halten. Siehe Kapitel 5.1.1.
- Wiederkehrende Funktionen sollen ausgelagert werden, um Codeduplikationen zu vermeiden. Siehe Kapitel 5.2.2.

1 Allgemeines

- Kommentare sollen kurz und aussagekräftig gehalten werden. Siehe Kapitel 5.2.1.
Kommentiert werden müssen mindestens:
 - Funktionen
 - wichtige Variablen, Konstanten, Defines und DatenstrukturenZu Dokumentation siehe auch Kapitel 7: „Dokumentation mit Doxygen“.
- Variablen sollen mit möglichst geringem Gültigkeitsbereich deklariert werden. Globale Variablen sollen vermieden werden. Siehe Kapitel 5.1.5.
- Für eigene Aufzählungstypen sollen `enums` verwendet werden. Siehe Kapitel 5.1.11.
- Als Kontrollstruktur bei aufzählbaren Datentypen (`enums`) sollen Switch-cases verwendet werden. Siehe Kapitel 5.31.
- Bitmasken sollen unter Zuhilfenahme der durch den avr-gcc bereitgestellten *defines* und gezieltes Bitshifting erzeugt werden. Siehe Kapitel 5.2.3.
- Mit Zeigern sollte vorsichtig und überlegt umgegangen werden. Bei der Definition eines Zeigers soll das Sternchen an den Variablennamen und nicht an den Datentypen geschrieben werden. Siehe Kapitel 5.1.7.
- Aussagekräftige *defines* für feste Werte erhöhen die Übersicht des Programmcodes. Dabei muss auf ausreichende Klammerung geachtet werden, um unerwünschtes Verhalten zu vermeiden. Siehe Kapitel 5.1.2
- Variablen müssen aussagekräftig benannt werden. Siehe Kapitel 5.2.3

2 Hardware

In diesem Kapitel wird die im Praktikum Systemprogrammierung verwendete Hardware vorgestellt. Dabei handelt es sich um einen ATmega644 der Firma Atmel, welcher in ein am Lehrstuhl entwickeltes Evaluationsboard eingebunden ist.

2.1 Der Mikrocontroller ATmega 644

Im Praktikum Systemprogrammierung wird ein 8 Bit Mikrocontroller der Firma Atmel verwendet – der ATmega 644. Dieser Mikrocontroller ist mit 20 MHz getaktet und besitzt verschiedene interne Komponenten, wie z. B. EEPROM-Speicher, einen AD/DA-Wandler, vier I/O-Ports, Unterstützung für interne und externe Interrupts, eine USART Schnittstelle und drei Timer.

Dieses Kapitel stellt die wichtigsten Komponenten vor und erklärt ihre Verwendung. Für tiefergehende Informationen über den Mikrocontroller kann das zugehörige Datenblatt eingesehen werden, auf welches im 12p verwiesen wird.

2.1.1 I/O-Ports

Der ATmega 644 besitzt vier I/O-Ports zur Ein- und Ausgabe von Signalen (Bezeichner 4 bis 7 in Abbildung 2.1). Diese sind mit *Port A* bis *Port D* bezeichnet. Die Ports sind aus jeweils 8 Pins zusammengesetzt. Um einen I/O-Port zur Ein- oder Ausgabe zu verwenden, muss er zunächst konfiguriert werden. Für die Portkonfiguration existieren zu jedem Port drei Register, welche im Folgenden bezüglich des *Port A* erläutert werden. Jedes Register kann als Ganzes beschrieben oder gelesen werden, um einen I/O-Port in einem Schritt zu setzen oder auszulesen. Die Pins eines Ports können ebenfalls einzeln konfiguriert werden, indem nur die entsprechenden Bits für diesen Pin in den Steuerregistern gesetzt werden. Die Prozessorregister können wie Variablen über ihren Namen angesprochen und manipuliert werden. Eine Deklaration entsprechender Variablen ist nicht notwendig, die Entwicklungsumgebung ersetzt vor dem Kompilieren jedes Vorkommnis eines Port Bezeichners durch dessen exakte Adresse. Für die Manipulation einzelner Bits in den Registern müssen Bitoperationen und Bitmasken verwendet werden. Siehe Kapitel 5.1.16 für eingehende Erklärungen bezüglich dieser, wie auch für Beispiele zum Registerzugriff. Es existieren *defines* (wie z. B. PORTA0 bis PORTA7 für das Register PORTA), die die Position der Bits, die für die Konfiguration eines einzelnen Pins zuständig sind, enthalten. Diese können in den Bitmasken verwendet werden.

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 an der Stelle DDRA0 im Register DDRA bedeutet, 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.

Verwendung des Data Registers bei der Eingabe Um bei der Eingabe definierte Werte auslesen zu können, selbst wenn von außen kein definiertes Signal am Pin anliegt, werden Pullup-Widerstände verwendet. Pullup bezeichnet einen (relativ hochohmigen) Widerstand, der eine Signalleitung mit dem höheren Spannungs-Potential verbindet. Durch ihn wird die Leitung auf das höhere Potential gebracht, für den Fall, dass kein Ausgang die Leitung aktiv auf ein niedrigeres Potential bringt.

Wenn ein Pin als Eingabepin definiert wurde, wird PORTA dazu verwendet, um den Pullup-Widerstand für diesen Pin zu aktivieren oder zu deaktivieren. Der Pullup-Widerstand für den zweiten Pin von Port A wird aktiviert, indem man eine 1 an die Stelle PORTA1 schreibt. Eine 0 bewirkt die Deaktivierung des Pullup-Widerstands.

Die Pullup-Widerstände müssen für jeden Pin aktiviert sein, an dem eine Eingabe von außen erfolgen soll, damit immer ein definiertes Signal anliegt. Eine Eingabe ist beispielsweise mit den Tastern des Evaluationsboards möglich (Bezeichner 11 in Abbildung 2.1).

Verwendung des Data Registers bei der Ausgabe Wenn ein Pin (etwa der zweite Pin von Port A) als Ausgabepin definiert wurde, wird PORTA dazu verwendet, um zu steuern, welches Signal ausgegeben werden soll. Wird dazu an die Stelle PORTA1 eine 0 geschrieben, so liegt auf dem zweiten Pin von Port A eine Spannung von 0 V (GND) als Ausgangssignal an. Eine 1 an dieser Stelle sorgt dafür, dass die Spannung VCC als Signal ausgegeben wird.

Wird eine Stelle des Registers PORTA ausgelesen, so erhält man den Wert, der dort gerade ausgegeben wird.

Input Pins

Das *Port A Input Pins-Register* PINA wird zum Auslesen externer Signale verwendet. Da Register nur als Ganzes gelesen werden können, muss auf den gelesenen Wert eine Bitmaske angewendet werden, um den Wert zu erhalten, der an einem bestimmten Pin anliegt. Nachdem für einen Pin (etwa den dritten Pin von Port A) der Pullup-Widerstand aktiviert wurde, um definierte Werte zu erhalten, enthält das Register an der Stelle PINA2 eine 1, falls ein Signal am Pin anliegt, und eine 0, wenn kein Signal anliegt.

In diesem Praktikum wird ein Board mit invertierter Buttonbelegung verwendet. Hier ist es so, dass ohne Buttondruck VCC anliegt und durch Druck eines Buttons der PIN mit Ground (GND) verbunden wird. Dadurch ergibt sich die Situation, dass beim Drücken eines Buttons eine 0 in PIN steht und nach Lösen des Buttons eine 1. Weitere Informationen über die Verwendung der I/O-Ports bietet das Datenblatt des Mikrocontrollers ATmega 644 im Abschnitt *I/O Ports*. Dort sind auch die Konfigurationsmöglichkeiten für die Ports in einer Tabelle zusammengestellt.

Beispiele

Im Folgenden finden Sie ein Beispiel für das Auslesen der an Pin C1 anliegenden Spannung (5V oder 0V).

```

1 // 1. Pin C1 als Eingang konfigurieren
2 DDRC  &= 0b11111101;
3
4 // 2. Pullup-Widerstand an Pin C1 aktivieren
5 PORTC |= 0b00000010;
6
7 // 3. Pin C1 auslesen (Bit extrahieren und nach rechts
8     verschieben)
9 uint8_t pinState = (PINC & 0b00000010) >> 1;
```

Listing 2.1: Button an Pin C1 wird abgefragt

Das nächste Beispiel zeigt die Verwendung von Pin C1 als Ausgang. Nach Ausführung des Codes liegt an Pin C1 eine 5V Spannung an.

```

1 // 1. Pin C1 als Ausgang konfigurieren
2 DDRC |= 0b00000010;
3
4 // 2. Logische 1 an Pin C1 ausgeben
5 PORTC |= 0b00000010;
```

Listing 2.2: Ausgabe einer logischen 1 an Pin C1

2.1.2 Interrupts

Es gibt externe und interne Ereignisse, die einen Interrupt auslösen können. Ein Interrupt, der durch ein externes (internes) Ereignis ausgelöst wird, heißt *externer (interner) Interrupt*. Ein externer Interrupt wird z.B. durch eine Signaländerung an einem Eingabepin ausgelöst; ein interner Interrupt wird z.B. von einem integrierten Timer (siehe nächster Abschnitt) ausgelöst. Externe und interne Interrupts werden vom Mikrocontroller gleich behandelt.

Als Interrupt bezeichnet man eine vorübergehende Unterbrechung des normalen Programmablaufes als Reaktion auf ein bestimmtes Ereignis. Nach dem Auslösen eines In-

terrupts wird die zu diesem Interrupt gehörende *Interrupt Service Routine (ISR)* ausgeführt. Danach wird der Programmablauf dort fortgesetzt, wo er durch den Interrupt unterbrochen wurde.

Ein interner Interrupt kann beispielsweise verwendet werden, wenn eine Funktion in bestimmten Zeitabständen immer wieder ausgeführt werden soll. In diesem Fall würde man einen Timer-Interrupt nutzen, der immer dann ausgelöst wird, wenn ein interner Timer des Mikrocontrollers einen vom Benutzer festgelegten Wert erreicht hat.

Implementierung Um Interrupts zu nutzen, muss zunächst eine Interrupt Service Routine angelegt werden, welche den Quelltext enthält, der als Reaktion auf den Interrupt ausgeführt werden soll.

Weiterhin muss der Interrupt aktiviert werden. Dazu muss sowohl der spezielle Interrupt einzeln eingeschaltet, als auch ein globales Interrupt-Enable-Flag gesetzt (I-Flag) werden. Das Einschalten einzelner Interrupts erfolgt über dafür vorgesehene Steuerregister. Weiterführende Informationen, insbesondere die benötigten Registerdaten, finden sich im Datenblatt des Mikrocontrollers in den Abschnitten *Interrupts*, *External Interrupts* und in den Abschnitten für die jeweiligen Komponenten, die einen Interrupt auslösen können.

2.1.3 Timer

Der in diesem Praktikum verwendete Mikrocontroller verfügt über drei Timer mit verschiedenen großen Zähler-Registern. Das Zähler-Register wird bei jedem Timer-Tick um eins erhöht. Die Timer des ATmega 644 sind im Einzelnen:

- Timer 0 (8 Bit Zähler-Register)
- Timer 1 (16 Bit Zähler-Register)
- Timer 2 (8 Bit Zähler-Register)

Die Timer können sehr flexibel konfiguriert werden und somit unterschiedliche Funktionen realisieren. Nähere Informationen sind im Datenblatt des ATmega 644 in den Abschnitten *8-bit Timer/Counter0 with PWM*, *16-bit Timer/Counter1 with PWM* und *8-bit Timer/Counter2 with PWM and Asynchronous Operation* zu finden.

Prescaler Für gewöhnlich wird für jeden Timer ein Prescaler festgelegt. Dieser gibt an, welcher Abstand zwischen zwei Timer-Ticks liegt. Der Prescaler beeinflusst somit wie schnell der Timer zählt. Es gilt folgende Formel:

$$\text{Timergeschwindigkeit} = \frac{\text{Frequenz des Controllers}}{\text{Prescaler}}$$

Je größer der Prescaler ist, desto langsamer zählt der jeweilige Timer. Der Wert des Prescalers kann jedoch nicht frei programmiert werden. Hardwarebedingt sind nur einige bestimmte Zweierpotenzen als Werte möglich. Alle Timer des ATmega 644 unterstützen die Werte 1, 8, 64, 256 und 1024 als Prescaler. Timer 2 ermöglicht zusätzlich die Prescaler-Werte 32 und 128.

Auslesen des Timers Innerhalb eines Programms kann jederzeit auf die Zähler-Register der Timer zugegriffen werden. Diese sind mit TCNT0 bis TCNT2 durchnummert, und können beliebig gelesen und beschrieben werden. Bei der Verwendung von TCNT1 ist zu beachten, dass sich TCNT1 intern aus zwei 8 Bit-Registern (TCNT1H und TCNT1L) zusammensetzt.

Timer Interrupts Timer werden verwendet, um Code in bestimmten Zeitabständen auszuführen. Dazu muss zunächst ein Vergleichswert festgelegt werden, so dass der Timer einen Interrupt auslöst, sobald das zugehörige Zähler-Register diesen Wert erreicht oder überschritten hat. Der für ein bestimmtes Zeitintervall nötige Vergleichswert kann aus der Frequenz des Quarzes, dem Prescaler des Timers und der Länge des gewünschten Zeitintervalls berechnet werden.

2.2 Das Evaluationsboard

Das Evaluationsboard, auf dem der Mikrocontroller eingebettet ist stellt, nützliche Peripherie zur Verfügung. Es sind z. B. ein LC-Display, ein RS232-Pegelkonverter, Pins für die vier I/O-Ports, vier Taster und verschiedenfarbige LEDs vorhanden, die sich leicht durch den Mikrocontroller ansteuern lassen.

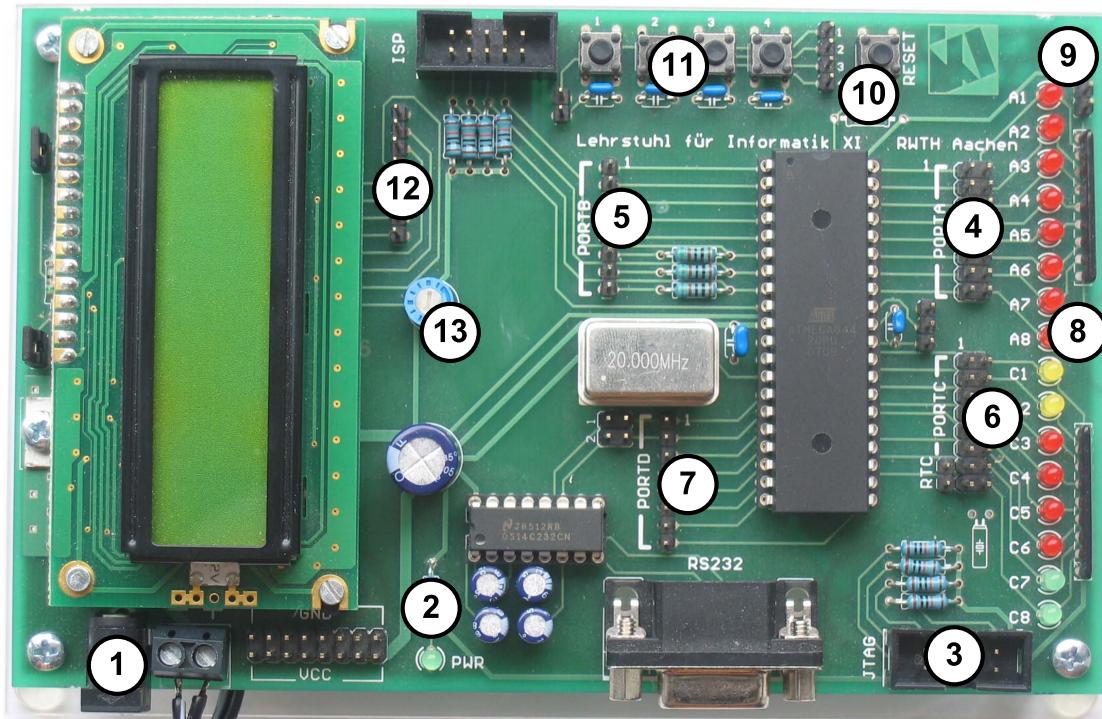


Abbildung 2.1: Das im Praktikum verwendete Evaluationsboard

2.2.1 Funktionen des Evaluationsboards

Abbildung 2.1 stellt die verwendete Platine mit ihren Anschlüssen, Tastern und Anzeigen dar. Die jeweils ausgewiesenen Bereiche sind im Einzelnen:

1. Anschluss zur Spannungsversorgung
2. Leuchtdiode „PWR“: Leuchtet, wenn eine Spannungsversorgung angeschlossen ist
3. Anschluss „JTAG“: Schnittstelle für den In Circuit Emulator (ICE) zum Programmieren und Debuggen
4. Pins „PORTA“: Anschlusspins für Zugriff auf Port A des Mikrocontrollers
5. Pins „PORTB“: Anschlusspins für Zugriff auf Port B des Mikrocontrollers
6. Pins „PORTC“: Anschlusspins für Zugriff auf Port C des Mikrocontrollers
7. Pins „PORTD“: Anschlusspins für Zugriff auf Port D des Mikrocontrollers
8. Dioden „A1“...„A8“ und „C1“...„C8“: Leuchtdioden z. B. für die Ausgabe von Signalen an einem Port
9. LED Enable: Dieser Jumper muss gesetzt werden, damit die Leuchtdioden als Ausgang genutzt werden können (Die jeweiligen Anschlusspins liegen direkt neben den Pins für Port A und C und müssen zusätzlich für jede LED gesetzt werden, die genutzt werden soll.)
10. Taster „RESET“: Setzt den Mikrocontroller zurück
11. Taster „1“...„4“: Taster mit Anschlusspins z. B. für die Eingabe von Signalen an einem Port
12. Port für den Anschluss des LC-Displays. Dieses muss mit Port A (siehe Kapitel 2.2.2) verbunden werden.
13. Potentiometer zum Einstellen des Displaykontrastes

Eine Besonderheit im Zusammenhang mit diesem Evaluationsboard ergibt sich durch die Verschaltung der Taster und LEDs. Eine logische 1 wird hier durch das Potential GND („active low“) und eine logische 0 durch das Potential VCC dargestellt. Daher erzeugt ein gedrückter Taster das Potential GND und die LEDs leuchten nur, wenn der entsprechende Ausgang auf dem Potential GND liegt.

2.2.2 Ansteuerung des LCD

Um das Display auf der Platine verwenden zu können, muss dieses mit Port A des Mikrocontrollers verbunden werden. Zur softwareseitigen Ansteuerung des Displays ist im Moodle ein fertiger Treiber in Form einer Header-Datei samt zugehöriger Implementierung (`1dc.c` und `1cd.h`) vorhanden. Die wichtigsten Funktionen werden hier kurz vorgestellt:

Initialisierung

- `lcd_init(void)`
Um das Display zu initialisieren muss diese Funktion zu Beginn der Laufzeit einmal aufgerufen werden.

Navigation des (nicht sichtbaren) Cursors Der Cursor bestimmt die Position auf dem LCD, an welche das nächste Zeichen geschrieben wird.

- `lcd_line1(void)`
Der Cursor springt zur ersten Position in der ersten Zeile.
- `lcd_line2(void)`
Der Cursor springt zur ersten Position in der zweiten Zeile.
- `lcd_goto(unsigned char row, unsigned char column)`
Bewegt den Cursor auf die übergebene Position. Gültige Werte für `row` sind {1,2}.
Gültige Werte für `column` sind {1 ... 16}.

Löschen des Displays

- `lcd_clear(void)`
Löscht den Displayinhalt und setzt den Cursor wieder auf die Anfangsposition zurück.
- `lcd_erase(uint8_t line)`
Löscht nur eine Zeile des LCD. Gültige Werte für `line` sind {1,2}.

Ausgabe (Zeilenumbrüche werden automatisch am Zeilenende vorgenommen)

- `lcd_writeChar(char character)`
Schreibt ein Zeichen auf das Display. Der Parameter `character` muss ein Zeichen aus dem ASCII Code sein. Gültige Zeichen sind alle Zeichen des englischen Alphabets, die Zeichen 'ä', 'ö', 'ü', 'ß' und Ziffern. Das Steuerzeichen '\n' führt zum sofortigen Zeilenumbruch.
- `lcd_writeHexNibble(uint8_t number)`
Schreibt ein Halbbyte, genannt „Nibble“. Gültige Eingaben sind die Zahlen 0-15, die als "0"-”F“ ausgegeben werden.
- `lcd_writeHexByte(uint8_t number)`
Schreibt eine 8 bit Zahl im hexadezimalen Format auf das Display.
- `lcd_writeHexWord(uint16_t number)`
Schreibt eine 16 bit Zahl mit führenden Nullen im hexadezimalen Format auf das Display.

2 Hardware

- **lcd_writeHex(uint16_t number)**
Schreibt eine 16 bit Zahl ohne führende Nullen im hexadezimalen Format auf das Display
- **lcd_writeDec(uint16_t number)**
Schreibt ein `unsigned int` als dezimale Zahl auf das Display. Führende Nullen werden unterdrückt.
- **lcd_writeString(const char* text)**
Schreibt einen nullterminierten String aus dem Arbeitsspeicher auf das Display. Bei Erreichen des Zeilenendes wird der Inhalt der nächsten Zeile gelöscht und neu beschrieben. Wenn das Zeilenende der zweiten Zeile erreicht ist, wird in der ersten Zeile weitergeschrieben. Ein enthaltenes Steuerzeichen '\n' führt zum Zeilenumbruch.
- **lcd_writeProgString(const char* string)**
Verhält sich analog zu `lcd_writeString`, der Parameter wird allerdings aus dem Flashspeicher gelesen.
- **lcd_drawBar(unsigned char percent)**
Zeigt einen Fortschrittsbalken aus bis zu 16 Elementen (eine Zeile) an. Der Parameter ist ein Prozentwert (Wertebereich {0...100}). Vor Aufruf der Funktion muss der Cursor an den Anfang einer Zeile gesetzt werden.

3 Einführung in Atmel Studio

In diesem Kapitel wird die Installation und Verwendung der im Praktikum Systemprogrammierung verwendeten Entwicklungsumgebung erläutert.

3.1 Verwendete Software

Im Rahmen dieses Praktikums wird Atmel Studio 7 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 die sogenannte Atmel AVR Toolchain, welche die Programmierung des Mikrocontrollers in der Sprache C ermöglicht. Atmel Studio kann unter <https://www.microchip.com>¹ kostenlos heruntergeladen werden.

Toolchain zur Entwicklung von ATmega-Programmen

Die Entwicklung von Programmen für einen ATmega-Mikrocontroller in der Programmiersprache C läuft folgendermaßen ab:

1. Erstellen des Quellcodes (.c- und .h-Dateien) mit Atmel Studio
2. Kompilieren der Dateien zu Objektdateien in Maschinencode (.o-Dateien) mit Hilfe der Atmel AVR Toolchain
3. Linken der Maschinencodedateien zu Programmdateien (.hex- oder .elf-Dateien) mit dem Linker
4. Laden der Programmdateien auf den Mikrocontroller mit Atmel Studio über die ISP- oder JTAG-Schnittstelle
- (5.) Debuggen des laufenden Programms über die JTAG-Schnittstelle mit Atmel Studio

Atmel AVR Toolchain Die Atmel AVR Toolchain ist eine Sammlung von Tools und Bibliotheken, welche die Erstellung von Applikationen für Mikrocontroller der Firma Atmel ermöglicht. Je nach gewählter Einstellung wird die Struktur des Programmes vor dem Erstellen des Maschinencodes aus Optimierungszwecken automatisch verändert. In Kapitel 3.4 wird das Vorgehen des Compilers erklärt. Dies kann zu unerwartetem Verhalten führen, wie in Kapitel 6.4.2, „Hinweise zum Debuggen“ genauer ausgeführt wird.

¹ Atmel Corporation wurde 2016 von Microchip Technology Inc. übernommen. Daher sind heute Datenblätter zu Atmel Produkten auf der Microchip Website zu finden.

Linker Der Linker hat die Aufgabe die einzelnen generierten Objektdateien zu einer einzelnen Programmdatei zu verbinden und den Programmeinstiegspunkt zu definieren. Es kann nötig sein, die Linker-Optionen umzustellen, um zum Beispiel ein Programm nicht am Anfang des Speichers des Mikrocontrollers abzulegen.

ISP- und JTAG-Schnittstelle Der Mikrocontroller kann über verschiedene Schnittstellen programmiert werden, von denen zwei besonders verbreitet sind: Die *In System Programming*-Schnittstelle (ISP) und die *Joint Test Action Group*-Schnittstelle (JTAG). Mit beiden Schnittstellen ist es möglich den Mikrocontroller zu programmieren und zu konfigurieren. Die JTAG-Schnittstelle ist zusätzlich in der Lage die Inhalte aller flüchtigen sowie nichtflüchtigen Speicher des Mikrocontrollers während des Betriebs auszulesen und sowohl die Speicherbereiche als auch den Programmablauf zu manipulieren. Dies erleichtert das Debuggen des Mikrocontrollers. Zur Verwendung der *JTAG-Schnittstelle* wird ein zusätzliches Gerät, der sogenannte *JTAG Programmer*, benötigt. Dieser ermöglicht die Kommunikation zwischen dem Computer und dem Mikrocontroller.

3.2 Verwendung von Atmel Studio

In diesem Kapitel wird Atmel Studio 7 vorgestellt. Zunächst werden die Elemente der Entwicklungsumgebung erläutert. Im Anschluss wird die Erstellung und Ausführung eines Projektes beschrieben. Abschließend werden die Debugfunktionen von Atmel Studio vorgestellt.

3.2.1 Organisation der Anwendung

Der Programmquellcode, der über mehrere Dateien modularisiert sein kann, wird in Projekten organisiert. In Atmel Studio werden mehrere Projekte innerhalb einer sogenannten *Solution* zusammengefasst. Eine Solution besteht aus mindestens einem Projekt, umgekehrt ist jedes Projekt einer eindeutigen Solution zugeordnet. Welche Projekte Teil einer Solution sind, speichert Atmel Studio in der Solution-Datei mit der Endung „atsln“. Eine Datei mit der Endung „cproj“ enthält die Namen der jeweils zu einem Projekt gehörenden Dateien.

3.3 Aufbau der IDE

Der Aufbau der Entwicklungsumgebung ist in Abbildung 3.1 dargestellt. Neben den üblichen Menü- und Symbolleisten existieren folgende Arbeitsbereiche:

1. *Solution Explorer*: In diesem Navigator sind alle Projekte inklusive ihrer Dateien zu finden, die zu der momentan geöffneten Solution gehören.
2. *Editor*: In diesem Bereich wird der Quellcode editiert. Am oberen Rand finden sich Reiter, mit deren Hilfe zwischen mehreren geöffneten Dateien gewechselt wird.

3 Einführung in Atmel Studio

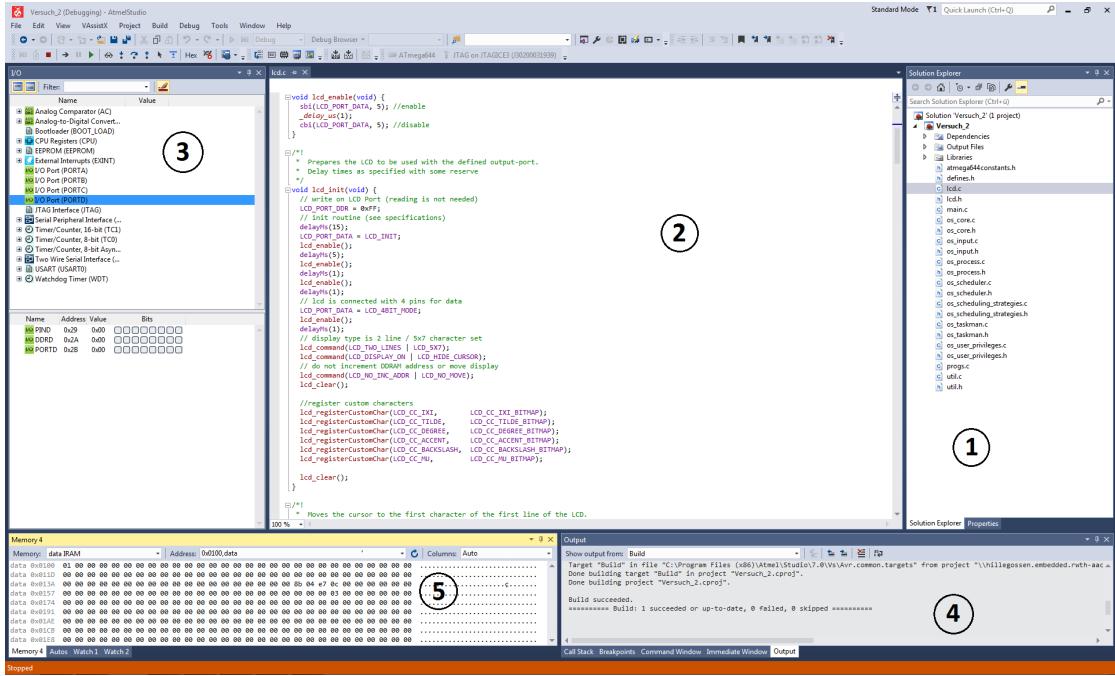


Abbildung 3.1: Atmel Studio

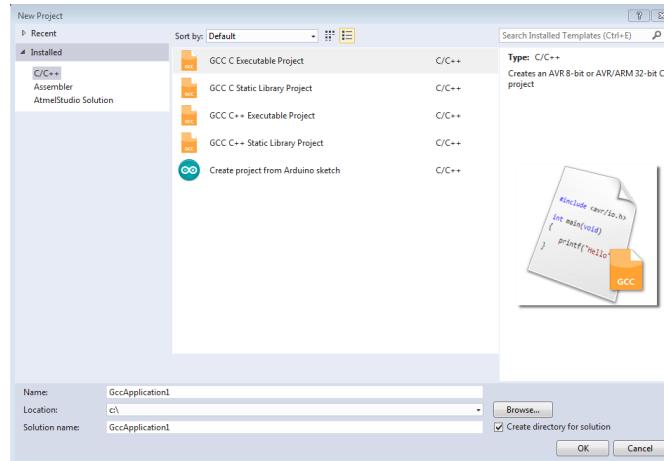
den kann. Mit einem Klick auf den linken Rand neben einer Codezeile kann ein Haltepunkt gesetzt werden.

- 3. IO-View:** Sobald der Debug-Modus betreten wird, überträgt Atmel Studio die entwickelte Anwendung mit Hilfe der JTAG- oder ISP-Schnittstelle auf den Mikrocontroller. Der I/O-View kann anschließend dazu verwendet werden, um den aktuellen Zustand des Mikrocontrollers einzusehen. Es können zum Beispiel die Werte der Timer, die Zustände der Ein- und Ausgangspins, sowie die Inhalte wichtiger Register angezeigt werden.
- 4. Output Tabs:** In diesem Fenster können verschiedene Systemmeldungen eingesehen werden. Dies betrifft vor allem Meldungen des Compilers, der hier über Warnungen und Fehler im Quellcode informiert. Des Weiteren stehen eine Suchfunktion und eine Auflistung aller momentan vorhandenen Haltepunkte zur Verfügung.
- 5. Memory:** Hier können die Inhalte aller Speicher (Flash, SRAM und EEPROM) des Mikrocontrollers ausgelesen werden.

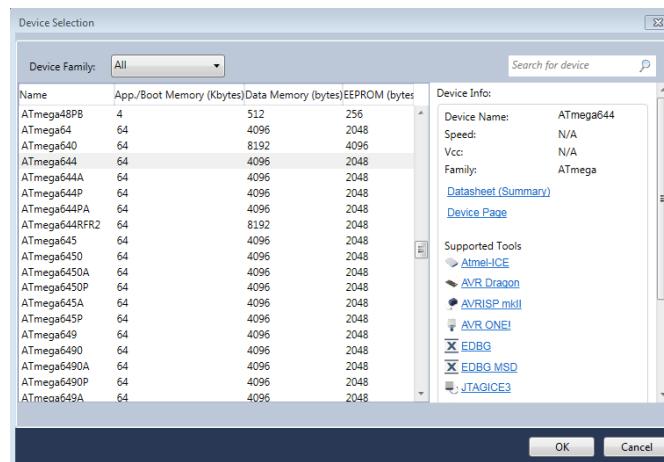
Die einzelnen Komponenten der Entwicklungsumgebung können flexibel angeordnet werden. Dazu besitzt der Menüpunkt „Debug“ das Untermenü „Windows“, in dem weitere Fenster ein- und ausgeblendet werden können.

3.3.1 Erstellen einer Solution mit einem Projekt

Nach dem Programmstart kann eine neue Solution, die initial über genau ein Projekt verfügt, über das Betätigen der Menüpunkte „File“ → „New“ → „Project“ angelegt werden. In dem anschließend erscheinenden Menü muss der Speicherort sowie der Name der Solution und des Projektes festgelegt werden. Als Projekttyp ist „GCC C Executable Project“ auszuwählen.



Auf der nächsten Seite wird der verwendete Mikrocontroller spezifiziert. Für dieses Praktikum muss an dieser Stelle immer „ATmega644“ ausgewählt werden.



Anschließend enthält der Solution Explorer eine neue Solution, die aus genau einem Projekt besteht. Das Hinzufügen weiterer Projekte zu der momentan geöffneten Solution erfolgt über die Menüpunkte „File“ → „Add“ → „New Project“ .

3.3.2 Konfigurieren der Toolchain

Atmel Studio verfügt über die Möglichkeit das Projekt eigenständig zu Kompilieren und zu Linken. Dieses Verfahren ist allerdings recht undurchsichtig für den Entwickler

und in der Praxis nicht wünschenswert. Im Praktikum wird der Kompilierprozess daher von einem speziellen *Makefile* gesteuert. Das Makefile wird stets gemeinsam mit den im Praktikum verwendeten Codegerüsten bereitgestellt. Die Datei befindet sich im obersten Verzeichnis neben der `.atsln`-Dabei und ist bereits im Projekt korrekt eingebunden. Um ein Makefile manuell einzubinden, muss in der Solution-Einstellung (Erreichbar durch TODO) im Untermenü *Build* *TODO confirm* die Option „Use external Makefile“ aktiviert werden. Um Systemabhängig zu arbeiten sollte dies nicht über den Dateidialog vorgenommen werden, sonder durch manuelle Eingabe des relativen Pfades „`../Makefile`“.

Das Makefile ist so konfiguriert, dass alle Source-Dateien im Projektordner (typischerweise *SPOS*) und allen Unterverzeichnissen kompiliert werden und die dabei erzeugten Artefakte im *bin*-Ordner abgelegt werden. Zusätzlich wird die *elf*-Datei, welche auf den Microcontroller geladen wird, neben dem Makefile abgelegt.

3.3.3 Ein Programm kompilieren und starten

Nach Erstellen des Quellcodes kann das Programm kompiliert und auf den Mikrocontroller geladen werden. Das Kompilieren erfolgt bei Aufruf des Befehls „Build Solution“ (F7), welcher über den Menüpunkt „Build“ zu erreichen ist. Falls es bei der Überprüfung des Quellcodes zu Fehlern gekommen ist, so wird der Kompiliervorgang abgebrochen und eine entsprechende Fehlermeldung ausgegeben. Ein Doppelklick auf eine Fehlermeldung führt zu der entsprechenden Stelle im Quellcode. Um das Programm auf den Mikrocontroller zu laden, steht die nachfolgend dargestellte Menüleiste zur Verfügung.



Das Programm wird durch den Befehl „Start Debugging and Break“ (1, Alt + F5) auf den Mikrocontroller geladen. Dabei wird das Programm im Debug-Modus gestartet und unmittelbar vor Ausführung der ersten Anweisung angehalten. Der Benutzer kann anschließend mit Hilfe des Befehls „Continue“ (4, F5) die Ausführung des Programms starten. Zur Laufzeit kann die Programmausführung über den Befehl „Break All“ (3, Strg + F5) pausiert werden. Die Programmausführung kann mittels der „Reset“-Funktion (5, Shift + F5) an den Anfang zurückgesetzt werden. Der Debugging-Modus kann über den Befehl „Stop Debugging“ (2, Strg + F5) verlassen werden. Die Menüleiste stellt weitere Debugging Funktionen zur Verfügung, welche im Kapitel 6 ausführlich vorgestellt werden.

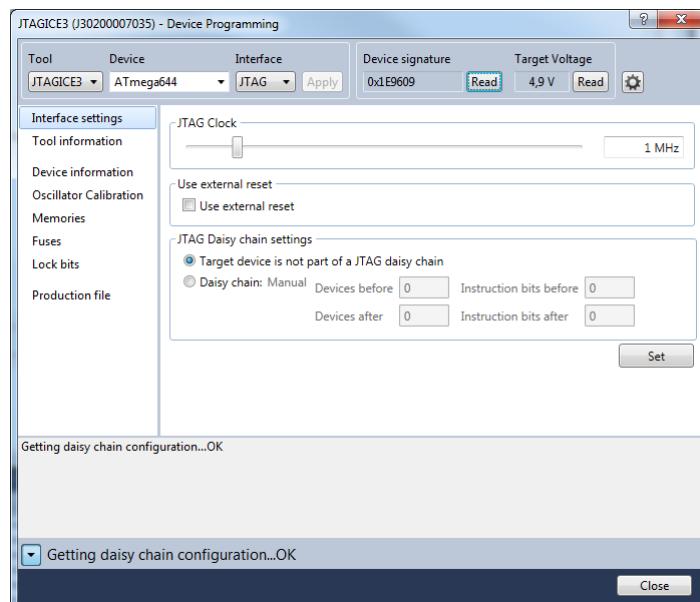
Alternativ zur im vorherigen Abschnitt beschriebenen Programmierung des Mikrocontrollers im Debug-Modus, kann der fertige Maschinencode direkt auf den Mikrocontroller geladen werden (Programmier-Modus). In diesem Modus gibt es keine Möglichkeit den Programmfluss oder den Speicherinhalt des Mikrocontrollers direkt zu beeinflussen.

Um den Mikrocontroller im Programmier-Modus zu beschreiben, muss zunächst auf den Button „Connect to the Selected AVR Programmer“ geklickt werden (siehe Markierung):

3 Einführung in Atmel Studio



Im sich anschließend öffnenden Fenster muss zunächst als Tool *JTAGICE3* und als Device *ATmega644* ausgewählt werden. Nach einem Klick auf Apply können im Reiter „Memories“ verschiedene Maschinen-Code Formate direkt auf den Mikrocontroller geladen werden. In diesem Fall der Programmierung beginnt der Mikrocontroller sofort nach dem Hochladen des Codes mit dessen Abarbeitung. Dieses Menü stellt die Möglichkeit bereit, Daten im Intel-HEX-Format direkt in einen Speicherbereich zu laden oder ein vollständiges Projekt als *.elf*-Datei auf den Mikrocontroller zu flashen.



3.3.4 Debugging

Atmel Studio stellt viele Hilfsmittel bereit, um Code zu testen und zu debuggen. Die wichtigsten sind:

- Das Setzen von Breakpoints - siehe Kapitel 6.2.1.
- Das Überwachen von Variablen - siehe Kapitel 6.2.3.
- Das Anzeigen von Registereinstellungen - siehe Kapitel 6.2.3.

In Kapitel 6, „Hinweise zum Debuggen“, wird ausführlicher auf diese und andere Methoden eingegangen. Um das Debugging schneller und effizienter durchführen zu können, empfiehlt sich das Lesen dieses Kapitels.

ACHTUNG

Durch Codeoptimierungen seitens des Compilers kann das Debuggen erschwert werden! Beachten Sie dazu Kapitel 3.4. Beachten Sie außerdem, dass die Aktivierung des Debug-Modus einige Einstellungen des Mikrocontrollers (z. B. die LockBits) zurücksetzen.

3.4 Compiler Optimierungen

Der Compiler nutzt eine Vielzahl von Optionen, um den C-Code in effizienteren Maschinencode zu übersetzen. Ab einer gewissen Stufe der Optimierung schließen sich die Optimierungen zur Verkleinerung des Programms und jene zur Steigerung der Geschwindigkeit teilweise aus. Die Optimierungen führen dazu, dass das Debuggen erschwert wird, da der vorhandene C-Code nicht mehr direkt auf Maschinencode abgebildet wird. Im Folgenden wird ein Teil der vom Compiler angewandten Optimierungen genauer dargestellt. Zum besseren Verständnis sind die hier dargestellten Techniken teilweise vereinfacht.

Sprungvorhersage Existieren im Programm bedingte Sprünge, die beispielsweise in *if*-Anweisung oder Schleifen auftreten, so versucht der Compiler abzuschätzen, welcher Sprung am wahrscheinlichsten ausgeführt wird. Falls möglich wird der Code mit der höheren Wahrscheinlichkeit direkt nach dem Sprung platziert.

Entfernung von unerreichbarem Code Ist ein Abschnitt des Codes nicht durch Sprünge erreichbar, wird er entfernt, um das resultierende Programm zu verkleinern.

Beispiel

Vor der Optimierung	Nach der Optimierung
<pre>// Anweisungen vorher if (1==0) { // Nie erreichbarer Code } // Anweisungen nachher</pre>	<pre>// Anweisungen vorher // Anweisungen nachher</pre>

Die Bedingung in der *if*-Abfrage ist nie erfüllt und ein Teil des Codes daher nicht erreichbar. Dieser wird zusammen mit der *if*-Abfrage entfernt.

Schleifenoptimierung Zur Optimierung von Schleifen stehen dem Compiler verschiedene Mittel zur Verfügung. Diese Maßnahmen verringern hauptsächlich die Anzahl benötigter Sprungbefehle im Maschinencode. So gibt es unter anderem für den Compiler folgende Möglichkeiten:

- Loop unrolling - Teilweises Auflösen von Schleifen, wenn kein inhaltlicher Zusammenhang zwischen dem Schleifenkörper und der Abbruchbedingung vorliegt. Dies verringert u.a. die Anzahl nötiger Sprünge.
- Zusammenfassen mehrerer voneinander unabhängiger Schleifen mit identischen Abbruchbedingungen, wodurch der Verwaltungsaufwand für Schleifen reduziert wird.
- Umformen von *while*-Schleifen in *do-while*-Schleifen. Diese haben nicht zwei bedingte Sprünge zu Beginn der Schleife, sondern nur einen einzigen am Ende.
- Umformen der Abbruchbedingungen von *for*-Schleifen in die Form Wert > 0. Dies ermöglicht die Verwendung des effizienten „Jump-Not-Zero“-Befehls.

Beispiel: Loop unrolling

Vor der Optimierung	Nach der Optimierung
<pre>int16_t i; for (i=0;i<5;i++) { array[i]=array[i]*2; }</pre>	<pre>array[0]=array[0]*2; array[1]=array[1]*2; array[2]=array[2]*2; array[3]=array[3]*2; array[4]=array[4]*2;</pre>

Die Anzahl der benötigten Sprunganweisungen im resultierenden Maschinencode und die Anweisungen zur Schleifenverwaltung wurden eingespart.

Beispiel: Zusammenfassen von Schleifen

Vor der Optimierung	Nach der Optimierung
<pre>int16_t i; for (i=0;i<5;i++) { array1[i]=0; } for(i=0;i<5;i++) { array2[i]=1; }</pre>	<pre>int16_t i; for (i=0;i<5;i++) { array1[i]=0; array2[i]=1; }</pre>

Hierbei wurde der Verwaltungsaufwand für die zweite Schleife eingespart.

Einsparung von Unterfunktionsaufrufen Bei Unterfunktionen mit simplen Berechnungen ist der Rechenaufwand für den Sprung in die Unterfunktion, die Übergabe der Parameter und den Rücksprung in die Hauptfunktion größer als der Aufwand zur Abarbeitung des Inhalts der Unterfunktion. Der Compiler kann Unterfunktionsaufrufe einsparen, indem der Aufruf durch den Code der aufzurufenden Unterfunktion ersetzt wird.

Beispiel

Vor der Optimierung	Nach der Optimierung
<pre>int16_t doubleValue(int x){ return x*2; } ... a=doubleValue(b); ...</pre>	<pre>... a=b*2; ...</pre>

Hier wurde ein Sprung in eine schnell abzuarbeitende Unterfunktion eingespart; der Inhalt der Unterfunktion wurde adaptiert und an die Stelle des Aufrufes verschoben.

Einsparen von Speicher und Maschinenbefehlen Der Compiler kann zusätzlich zu den erwähnten Methoden auf verschiedene weitere Arten versuchen, den Speicherverbrauch und die Rechenzeit zu optimieren:

- Berechnungen mit Konstanten können meist während des Kompilierens durchgeführt werden, um Rechenzeit während der Ausführung einzusparen.
- Berechnungen mit Hilfsvariablen können derart zusammengefasst werden, dass ein Teil der Variablen nicht mehr benötigt wird.
- Häufig verwendete Variablen können in den Prozessorregistern gespeichert werden, um den Zugriff zu beschleunigen. Dies findet beispielsweise Verwendung bei Laufvariablen von Schleifen.

ACHTUNG

Diese Optimierungen können beim Debuggen dazu führen, dass einige Variablen bzw. Codepassagen nicht mehr überwacht werden können. Lösungen für solche Probleme finden sich im Kapitel 6 „Hinweise zum Debuggen“.

Beispiel: Vorabberechnung von Konstanten und Einsparung von Hilfsvariablen

Vor der Optimierung	Nach der Optimierung
<pre>... float pi = 3.14159; float radius = 2*pi*r; ...</pre>	<pre>... float radius = 6.28318*r; ...</pre>

Hier wurde die Berechnung von 2π bereits vom Compiler vorgenommen und muss nicht mehr während der Ausführung des Programmes geschehen. Die Variable `pi` und damit der von ihr verbrauchte Speicher konnten eingespart werden.

4 Benutzung des Testpools

Der Testpool des Praktikum Systemprogrammierung besteht aus Virtuellen Maschinen (VMs), welche per Windows-Remotedesktopverbindung erreichbar sind. Diese sind mit der gleichen Peripherie ausgestattet, welche auch während der Präsenzzeit des Praktikums Verwendung findet. Das Verhalten der Mikrocontroller kann durch eine Kamera beobachtet und durch Bedienelemente beeinflusst werden. Im Folgenden wird erläutert, wie eine Verbindung zu diesen Virtuellen Maschinen hergestellt werden kann und was bei der Benutzung zu beachten ist.

4.1 Remotedesktopverbindung beantragen

Um den Testpool zu verwenden muss über die Praktikumswebsite <https://psp.embedded.rwth-aachen.de> zunächst eine Sitzung beantragt werden. Dazu werden die TIM-Kennung und der dazugehörige TIM-Service „PC-Pool“ benötigt. Der PC-Pool Service ist standardmäßig nicht freigeschaltet und muss, falls noch nicht geschehen, im TIM-Accountmanagement angelegt werden.

Nach erfolgreichem Login auf der Website befinden Sie sich auf der Startseite. Wenn Sie nun links unten auf die Schaltfläche mit der Bezeichnung *Session* klicken, so erscheint ein Fenster in welchem Sie eine Sitzung beantragen können. Nun erscheint eine Auswahl der verfügbaren Sitzungstypen (Abbildung 4.1). Jeder vorhandene Sitzungstyp ist mit einer bestimmten Hardwarekonfiguration assoziiert, welche für die jeweiligen Versuche geeignet ist.



Abbildung 4.1: Typenauswahl

Sobald eine Sitzung beantragt wurde, werden Sie in die Warteschlange eingereiht (Abbildung 4.2). Position 1 bedeutet dabei, dass die nächste verfügbare Instanz reserviert wird. Sobald eine Instanz verfügbar ist, werden Sie dieser zugewiesen und haben fünf Minuten Zeit die Sitzung zu aktivieren. Wenn die Sitzung nicht rechtzeitig aktiviert wird, verfällt

4 Benutzung des Testpools

diese und die Instanz wird für andere Studierende freigegeben. Dieser Mechanismus dient der Fairness und verhindert, dass Computer im Testpool ungenutzt blockiert werden.



Abbildung 4.2: Testpool Warteschlange

Nach Klicken auf den Button „Session aktivieren“ wird der zugewiesene Host für Ihren Nutzer freigegeben. Der Zugriff auf den Host kann nur von der IP-Adresse erfolgen, mit der die Session aktiviert wurde. (Abbildung 4.3).

Sofern keine anderen Studenten auf Testpool Sitzungen warten, besteht 15 Minuten vor Ende Ihrer Sitzung die Möglichkeit diese zu verlängern.

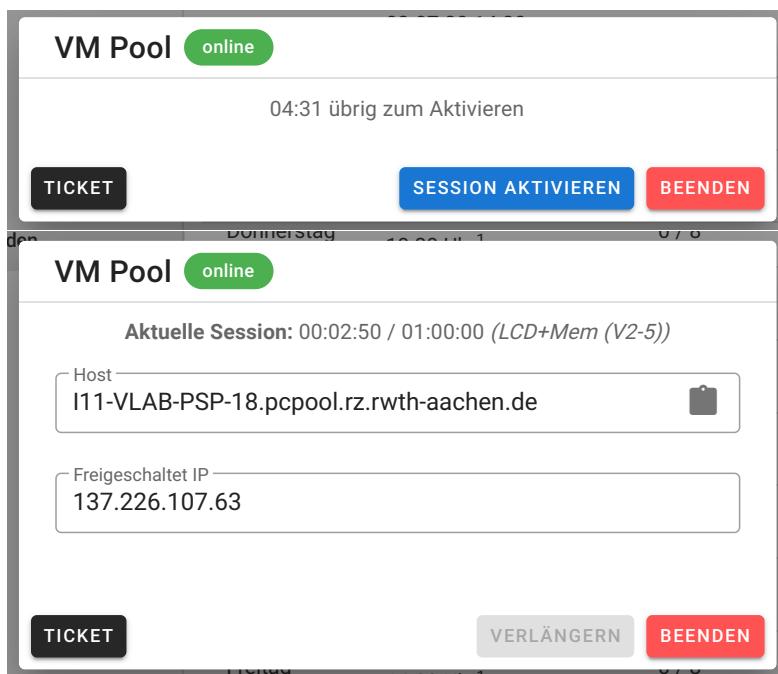


Abbildung 4.3: Aktivierte Testpool Sitzung

4.2 Remotedesktopverbindung herstellen

Zum Herstellen einer Remotedesktopverbindung kann unter Windows das Programm „Remotedesktopverbindung“, wie in Abbildung 4.4 dargestellt, verwendet werden. Zur Fernsteuerung unter Linux kann beispielsweise „Terminal-Server-Client“ sowie unter Mac „Microsoft Remotedesktopverbindungs-Client“ genutzt werden.

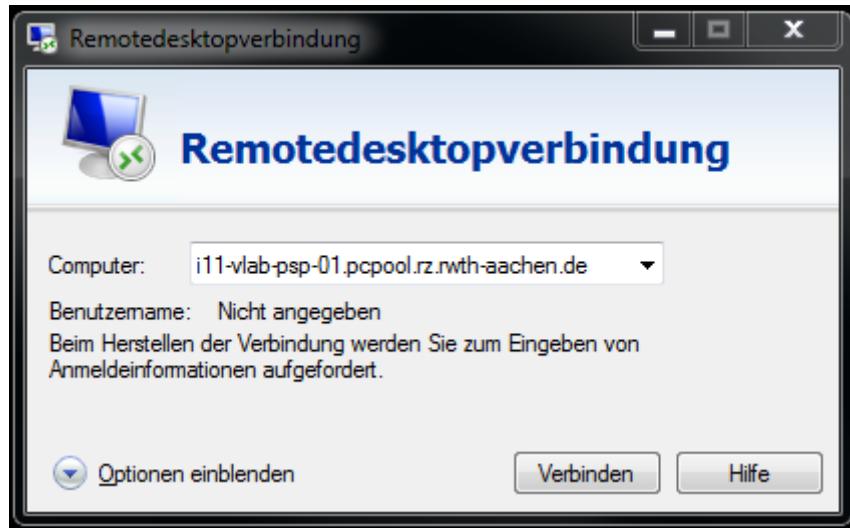


Abbildung 4.4: Remotedesktopverbindung zum Testpool unter Windows

Eine wichtige Anforderung an die Remotesitzung besteht darin, eine ausreichende Farbtiefe auszuwählen. Diese muss mindestens 16 Bit betragen, um das Videobild der Kamera darzustellen (siehe Abbildung 4.6). Bei einer zu geringen Farbtiefe wird das Videobild schwarz dargestellt.

Bei der Nutzung des Windows-Tools „Remotedesktopverbindung“ muss der zugewiesene Host in das entsprechende Eingabefeld eingetragen und auf „Verbinden“ bzw. „Connect“ geklickt werden.

Anschließend wird ein Login in der Domäne **PC-POOL** mit einer TIM-Kennung erwartet. Wichtig ist hierbei, dass **PC-POOL** vor der TIM-Kennung als explizite Domäne angegeben wird, wie in Abbildung 4.5 dargestellt ist. Die beim Verbinden eventuell erscheinende Warnung sollte gelesen und bestätigt werden.

Da die Vergabe der Sitzungen in Form einer priorisierten Warteschlange erfolgt und die Nutzungsdauer in die Priorität für zukünftige Sitzungsanträge eingeht, sollten Sitzungen nach Beendigung der Arbeit manuell beendet werden.

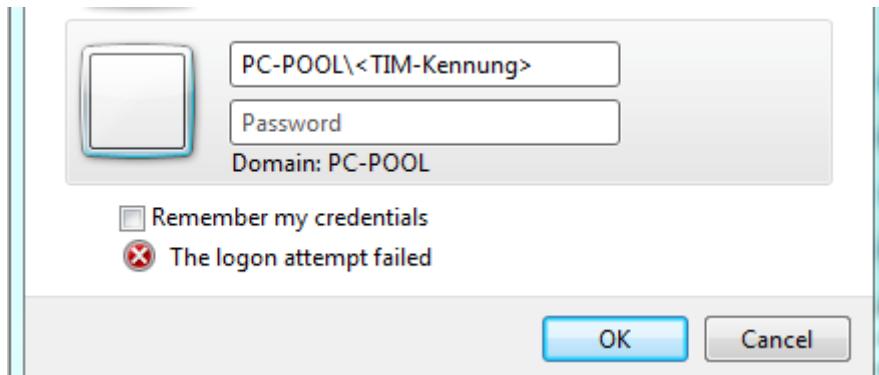


Abbildung 4.5: Benutzerlogin auf einem Testpoolrechner unter Windows

4.3 Auf Remotedesktop arbeiten

Nach erfolgter Anmeldung wird der Remotedesktop angezeigt. Um den Mikrocontroller programmieren zu können, muss die Software ATMega Remote gestartet werden. Die Hardware wird nur dann mit Strom versorgt, wenn diese Software aktiv ist. In Abbildung 4.6 ist das Programmfenster beispielhaft für einen Versuch dargestellt. Da die Arbeitsgeschwindigkeit eingeschränkt wird, wenn das Video Bild über die Remotedesktopverbindung übertragen wird, bietet es sich an, das Hauptfenster der Software zu minimieren, wenn das Kamerabild nicht benötigt wird. Somit wird eine ausreichende Arbeitsgeschwindigkeit gleichermaßen wie die Möglichkeit, den Mikrocontroller zu programmieren, gewährleistet.

Die Interaktion mit dem Mikrocontroller gestaltet sich durch die ATMega Remote Software wie folgt:

Die Taster, die das Video Bild überlagern, können per Maus bedient werden. Hierbei löst ein **Linksklick** ein kurzes Drücken des Tasters aus und ein **Rechtsklick** dient dazu, den Taster gedrückt zu halten. Ein erneuter Links- oder Rechtsklick setzt den Taster in seinen Ursprungszustand zurück. In der oben angeordneten Toolbar des Programmfensters befindet sich ein Button „Hardware-Reset“, mit welchem sich die Mikrocontrollerhardware vollständig neustartet lässt. Ein Neustart ist notwendig, falls die Geräte von Atmel Studio nicht korrekt erkannt werden. In einigen Fällen ist zudem ein Neustart des Atmel Studio notwendig, um die korrekte Funktionsweise des Programmiergerätes wiederherzustellen.

Zudem kann das Programmfenster mit den Lupen-Tasten auf drei mögliche Größen eingestellt werden. Aufgrund der beschränkten Datenübertragungsrate durch die Remotedesktopverbindung ist eine kleinere Bildgröße zu bevorzugen, um eine höhere Bildwiederholrate zu erreichen.

Um Fehlermeldungen von Atmel Studio zu vermeiden, sollte dieses immer vor ATMega Remote beendet werden.

Der Austausch von Dateien zwischen dem lokalen Computer und dem Testpool Rechner wird seitens der Windows Remotedesktopverbindung durch einfaches Kopieren und

4 Benutzung des Testpools

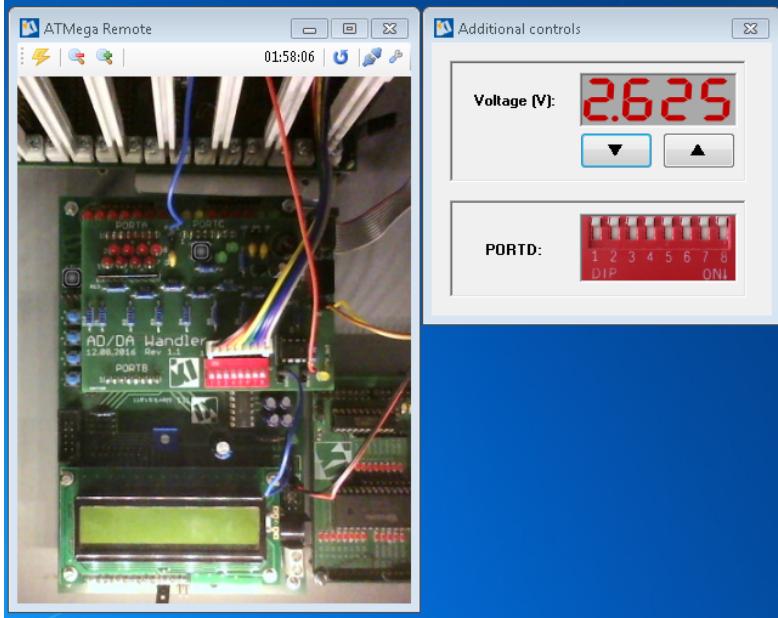


Abbildung 4.6: Das Programm ATMega Remote

Einfügen bereitgestellt. Hierbei ist zu beachten, dass Dateien aufgrund von Rechteeinschränkungen im Allgemeinen nur auf den Desktop bzw. das Benutzernetzlaufwerk des Testpool Computers kopierbar sind. Das Benutzernetzlaufwerk ist auf dem Arbeitsplatz zu finden und trägt als Volumenbezeichnung die TIM-Kennung des angemeldeten Nutzers.

4.4 Teamnetzlaufwerk mounten

Im Rahmen des Praktikums steht ein Netzlaufwerk zur Verfügung, welches als Arbeitsumgebung für die Versuchstermine dient. Über die Praktikumswebseite hochgeladene Hausaufgaben werden zu Beginn der Versuchstermine dort bereitgestellt. Der Zugriff auf dieses Laufwerk ist innerhalb des RWTH-Netzwerks oder per VPN über folgenden Pfad möglich:

`\pcpool.rz.rwth-aachen.de\files\Group\embedded\psp\GruppeX`

Hierbei bezeichnet X die Gruppennummer, welche bei der Einführungsveranstaltung bekannt gegeben wurde. Diese muss ohne führende Nullen in den Pfad eingefügt werden. Abbildung 4.7 veranschaulicht das Verbinden des Netzlaufwerks am Beispiel von Gruppe 1.

Informationen zum Aufbau einer Verbindung in das RWTH-Netzwerk über VPN stellt das Rechenzentrum bereit: VPN Installationshilfe. Falls sie den Cisco Client verwenden sollte als Konfiguration für die VPN Verbindung „Full Tunnel“ innerhalb des Clients

4 Benutzung des Testpools

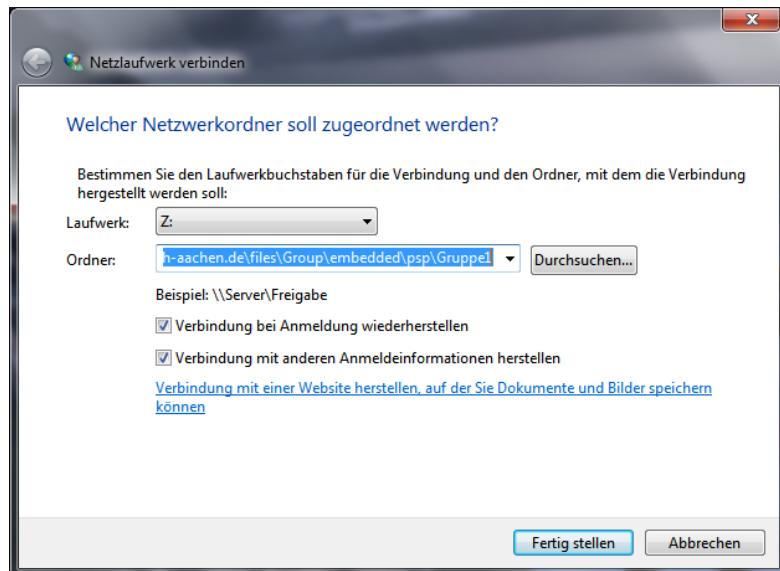


Abbildung 4.7: Den Gruppenordner als Netzlaufwerk mounten.

ausgewählt werden.

5 Einführung in die C Programmierung

In der hardwarenahen Systemprogrammierung wird häufig die Programmiersprache C eingesetzt. *Hardwarenah* bedeutet, dass dem Programmierer normalerweise kein Betriebssystem zur Verfügung steht, welches den Umgang mit der Maschine vereinfacht. Bei hardwarenaher Programmierung muss beispielsweise direkt auf Register zugegriffen oder Signale erzeugt werden, um mit Peripheriegeräten zu kommunizieren. Solche Aufgaben werden bei der Entwicklung von x86-Programmen mit Hochsprachen (z. B. Java und auch C++) vom Betriebssystem übernommen, ein direkter Zugriff ist häufig gar nicht möglich. Einerseits vereinfacht eine Hochsprache die Entwicklung erheblich und verringert die Abhängigkeit des Codes von einer bestimmten Plattform. Andererseits entzieht diese Abstraktion dem Entwickler teilweise die Kontrolle über das, was die Maschine ausführt. In vielen Fällen, gerade im Bereich eingebetteter Systeme, sind die Aufgaben so simpel, dass es nicht nötig ist ein Betriebssystem zu implementieren. Die erzeugten Programme können auch direkt auf der Hardware laufen, müssen allerdings selbst für Funktionalitäten, wie z. B. Input/Output, sorgen.

Wird ein Programm nicht in C, sondern direkt in einer Assemblersprache geschrieben, wird es unmittelbar in die zugehörige Maschinensprache übersetzt. Das bedeutet, dass jedem Assemblerbefehl ein eindeutiger Maschinenbefehl zugeordnet wird. Aus diesem Grund ist jede Assemblersprache stark maschinenabhängig. Sollte beispielsweise das gleiche Programm auf einem anderen Prozessortyp ausgeführt werden sollen, wird womöglich eine vollständige Neuentwicklung notwendig. Obwohl sämtliche Algorithmen logisch gleichwertig bleiben, muss das Programm neu geschrieben und getestet werden. Weiterhin sind Assemblersprachen relativ schwer für Menschen lesbar und daher fehleranfällig. Die Sprache C stellt einen Kompromiss zwischen der Hardwarenähe und Vorhersagbarkeit von Assemblersprachen und der Unterstützung des Entwicklers durch Hochsprachen dar.

Abhängig vom vorhandenen Vorwissen ist es nicht nötig dieses Kapitel komplett zu lesen. Dennoch wird in vielen Abschnitten auf Details eingegangen, welche selbst bei grundlegendem Wissen über die Sprache C eventuell unbekannt sind. Insbesondere im Vergleich zu Hochsprachen, wie etwa Java oder Pascal, weist C viele Eigenschaften auf, die unter Umständen zu unerwarteten Problemen führen können. Die meisten und vor allem die wichtigsten Informationen und Besonderheiten werden in Lernerfolgsfragen abgedeckt: Wenn Sie der Meinung sind, das im jeweiligen Abschnitt vorgestellte Thema bereits zu kennen, können Sie dies anhand der Fragen am Ende eines jeden Abschnitts auch vor dem Lesen des Abschnitts überprüfen.

5.1 Syntax und Semantik von C

5.1.1 Struktur eines C-Programms

Die Architektur eines Softwaresystems wird typischerweise in Form von *Modulen* beschrieben, welche einzelne Funktionalitäten zusammenfassen. So können Module mathematische Funktionen zusammenfassen und bereitstellen, andere Module bieten stattdessen Grafikfunktionen oder Treiber für externe Hardware. Module können beliebig zusammengefasst und wiederverwendet werden, wodurch sich ein komplexeres System strukturiert zusammenbauen lässt. In Java entspricht der allgemeine Begriff des Moduls den Klassen oder auch, falls mehrere Klassen zusammengehören, den Paketen. In Pascal und Delphi entsprechen die Units den Modulen. C bietet ein sehr ähnliches Konzept, nämlich die Unterscheidung von Header- und Implementierungsdateien mit den entsprechenden Dateiendungen .h und .c.

Die Header-Datei beschreibt die Schnittstelle des Moduls. Die Schnittstelle ist das, was von dem Modul nach außen hin sichtbar sein soll. In Java wären dies die als `public` deklarierten Methoden, Variablen und Konstanten. Im Gegensatz zu Java unterscheidet C allerdings stärker zwischen Deklaration und Implementierung. Daher befindet sich in einer Header-Datei typischerweise nur die sogenannte Signatur einer Funktion, also Funktionsname sowie Art und Anzahl der Parameter, aber kein ausführbarer Code. Die eigentliche Implementierung der Funktion findet in der .c-Datei statt.

Sämtliche in der zugehörigen .h-Datei deklarierten Funktionen müssen implementiert werden. Zusätzliche Funktionen, welche nicht in der Header-Datei deklariert wurden, sind ebenfalls möglich. Solche Funktionen sind nicht nach außen sichtbar, d.h. andere Module können sie nicht benutzen. Dies entspricht in etwa `private` in Java. Globale Variablen können auf dieselbe Weise publiziert, beziehungsweise versteckt werden.

Ein Modul kann ein anderes Modul benutzen, indem es die Header-Datei des anderen Moduls einbindet. Außerdem muss die .c-Datei eines Moduls stets die eigene .h-Datei einbinden. Erst dadurch werden .c und .h zu einer Einheit. Das Hauptmodul enthält die Funktion `main`, welche das Hauptprogramm darstellt. Dieses Modul besteht typischerweise, im Gegensatz zum bisher gesagten, nur aus der .c-Datei. Das Beispiel in Listing 5.1 soll die grundlegende Struktur verdeutlichen.

In den nachfolgenden Abschnitten werden die Details näher erläutert.

LERNERFOLGSFRAGEN

- Wieso wird C-Quelltext in Header- und Codedateien unterteilt?

```

1 //----- INCLUDES -----
2 #include <avr/io.h>      // Input/Output an den Pins
3 #include <avr/interrupt.h> // Interrupts
4
5 #include "foo.h"
6
7 //----- GLOBALS -----
8 int counter = 1;
9
10 //----- FUNCTIONS -----
11 int bar(int a, int b) {
12     return foo(a,&counter)+foo(b,&counter);
13 }
14
15 int main(void) {
16     return bar(2,++counter);
17 }
```

Listing 5.1: Ein typisches C-Programm, `foo.c`

5.1.2 Der Präprozessor

Bevor das Programm durch den *Compiler* übersetzt wird, durchläuft es den *Präprozessor*. Dieser ersetzt vordefinierte Symbole durch ihre Bedeutung. Diese Ersetzung kann zudem an Bedingungen geknüpft werden, wodurch bedingte Kompilierung möglich wird. So kann man *vor* dem Kompilieren Einfluss darauf nehmen, welche Konstanten gesetzt sind und ob manche Programmteile benötigt werden oder nicht. Gerade bei der Arbeit mit eingebetteten Systemen ist dies nützlich, da somit konsequent Speicher gespart wird: Nicht benötigte Teile erscheinen gar nicht erst im kompilierten Code. Dadurch kann das gleiche Programm für verschiedene Plattformen kompiliert werden, wenn man entsprechende Präprozessorbedingungen formuliert hat.

Im Folgenden werden die wichtigsten sogenannten Direktiven vorgestellt. Dabei ist zu beachten, dass diese im Gegensatz zu Befehlen der C-Syntax nicht mit einem Semikolon beendet werden.

Die #define Direktive

DEFINITION

- `#define SYMBOL WERT`
- `#define SYMBOL(x1,x2,...,xN) WERT`

Die `#define` Direktive definiert das Symbol namens `SYMBOL`. Der Präprozessor ersetzt jedes Vorkommen von `SYMBOL` durch `WERT`. Wird die zweite Variante eingesetzt, können beliebig viele Parameter für `SYMBOL` festgelegt werden. Alle Vorkommnisse dieser Parameter im `WERT` werden dann durch die jeweiligen Argumente ersetzt. Ein solches `#define` wird auch als Makro bezeichnet.

```

1 // Korrektes Vorgehen
2 #define PI 3.1415
3 #define TWOPI (2*PI)
4 #define DOUBLE(x) ((x)*2)

5
6 // Ohne Klammerung
7 #define A 2+3
8 #define B a*a
9 uint8_t erg = B;
10 // erg ist 2+3*2+3, also 11
11
12 // Mit Klammerung
13 #define C (2+3)
14 #define D (a*a)
15 uint8_t erg2 = D;
16 // erg2 ist ((2+3)*(2+3)), also 25

```

Listing 5.2: `#define` Beispiel

Beispiel 5.2 zeigt die Definition zweier Symbole: `PI` und `DOUBLE`. Es ist eine gängige Konvention `#define`-Namen immer in Großbuchstaben zu schreiben, Makrovariablen stets zu klammern und diese möglichst nur einmal zu verwenden, um unerwünschte Seiteneffekte zu vermeiden.

`#define` bietet eine Möglichkeit in C Konstanten zu definieren. Konstanten sollten in der Implementierung *niemals* hartkodiert, also als Wert, vorkommen. Stattdessen sollten sie zu Beginn eines Moduls in der hier gezeigten Weise definiert werden. Dies erhöht die Les- und Wartbarkeit.

Die **#ifdef** und **#ifndef** Direktiven

DEFINITION

- **#ifdef SYMBOL**
#endif
- **#ifndef SYMBOL**
#endif

Eine solche Direktive, auch *Include guard* genannt, prüft, ob ein Symbol bereits definiert (**#ifdef**) bzw. noch nicht definiert (**#ifndef**) wurde. Dies ist besonders hilfreich innerhalb von .h-Dateien, um mehrfache Einbindung zu vermeiden. Beispiel 5.3 verwendet die

```
1 // math.h
2 #ifndef _MATH_H
3 #define _MATH_H
4 // Tatsächlicher Inhalt des "math" Moduls
5 #endif
6 // EOF
```

Listing 5.3: **#ifndef** Beispiel

„if not defined“ Direktive um mehrfaches Einbinden der Datei `math.h` zu verhindern. Falls das Symbol `_MATH_H` **nicht** definiert ist, wird es definiert und bis `#endif` fortgefahrt. Ansonsten wird der gesamte Block (hier die gesamte Datei) übersprungen. **#ifdef** funktioniert analog.

Die **#if** Direktive

DEFINITION

- **#if EXPRESSION**
#endif
- **#if EXPRESSION1**
#elif EXPRESSION2
#else
#endif

`#if` wertet den Ausdruck aus und fährt bis `#endif` fort, falls dieser ungleich 0 ist (siehe auch Abschnitt 5.1.13). Die Direktive `#if` kann mit dem Schlüsselwort `#else` um einen zweiten Fall ergänzt werden, der bei Nichtzutreffen der Bedingung ausgeführt wird. Zusätzliche Fälle können mit `#elif` eingefügt werden. Im Kompilat ist nur der zutreffende Fall enthalten, alle anderen werden vollständig entfernt.

```

1 #if SIMULATION==1
2     // C-Code oder weitere Direktiven
3 #elif 0
4     Dieser Text (syntaktisch falscher Code) wird nicht an den
5         Compiler übergeben! Denn 0 ist niemals erfüllt.
6 #else
7     // Anderer Code
#endif

```

Listing 5.4: `#if` Beispiel

In der Sprache C gibt es analog zu dem Präprozessor-Befehl `#if` einen Befehl `if` (Einführung in Abschnitt 5.1.15). Der maßgebliche Unterschied zwischen den beiden Formen der bedingten Anweisung ist, dass die Bedingung des Präprozessorbefehls schon zum Zeitpunkt des Kompilierens feststehen muss. Dies hat aber den Vorteil, dass der resultierende Code kleiner wird, da immer nur eine der Verzweigungen in Code abgebildet wird. Ferner werden, wie Beispiel 5.4 zeigt, im jeweiligen Kontext syntaktisch falsche Codefragmente nicht an den Compiler übergeben.

Die `#include` Direktive

DEFINITION

- `#include "DATEI"`
- `#include <DATEI>`

Der Präprozessor ersetzt den Befehl durch den Inhalt der angegebenen Datei. Dies stellt die Funktionalität des zugehörigen Moduls (DATEI.H) im inkludierenden Modul zur Verfügung. Der Präprozessor beginnt die Suche nach der spezifizierten Datei entweder im Suchpfad oder im aktuellen Verzeichnis. Ein Dateiname in spitzen Klammern impliziert die Suche im Suchpfad, während ein Dateiname in Anführungszeichen die Suche im aktuellen Verzeichnis beginnen lässt. Beispiel 5.5 demonstriert das Einbinden eines Bibliotheksmoduls avr/io.h und einer Projektdatei myTypes.h.

```
1 #include <avr/io.h>
2 #include "myTypes.h"
```

Listing 5.5: `#include` Beispiel

LERNERFOLGSFRAGEN

- Können Sie Beispiele angeben, wie Präprozessor-Direktiven die Les- und Wartbarkeit verbessern können?
- Wieso ist es sinnvoll Headerdateien anderer Module einzubinden und wie werden diese eingebunden?
- Was ist der Unterschied zwischen spitzen Klammern und Anführungszeichen beim Einbinden von Dateien?
- Wie sieht ein typisches Einsatzszenario von `#if`-Direktiven aus?

5.1.3 Kommentare

Wichtige Maßstäbe des Entwicklungsprozesses sind Wart- und Wiederverwendbarkeit der entwickelten Software. Das Kommentieren von Quellcode ist ein wesentliches Hilfsmittel. Einzelne Zeilen werden mit den Zeichen `//` auskommentiert, ganze Kommentarblöcke können mit `/*` eingeleitet und mit `*/` beendet werden. Kommentare sind syntaktisch irrelevant. Listing 5.6 zeigt einige Beispiele.

Zu viele oder triviale Kommentare verschlechtern die Lesbarkeit. Ein nur durch viele Kommentare verständlicher Quelltext ist ein Indiz für einen schlechten Programmierstil. Sinnvoller ist die Verwendung von aussagekräftigen Variablen- und Funktionsnamen. Siehe dazu Abschnitt 5.45.

Kommentare können mit speziellen Schlüsselwörtern versehen werden, aus welchen entsprechende Tools eine Dokumentation generieren können. Ein Beispiel für ein solches Tool ist *Doxxygen*.

```

1 // Kommentar über Befehl
2 befehl();
3
4 befehl(); // Kommentar hinter Befehl
5
6 /*
7 Dies ist ein mehrzeiliger Kommentar,
8 in dem sich viele Informationen
9 unterbringen lassen...
*/
10
11 befehl();

```

Listing 5.6: Ein mit Kommentaren angereichertes C-Programm

Beachten Sie, dass die meisten Kommentare in diesem Dokument dazu dienen, Ihnen wesentliche Konstrukte der Sprache C zu erläutern und daher nicht als Beispiele zur Dokumentation von Quellcode zu verstehen sind.

LERNERFOLGSFRAGEN

- Kommentare werden vom Compiler ignoriert. Warum schreibt man dennoch Kommentare in Quelltextdateien?

5.1.4 Datentypen

Die Größe von Datentypen ist abhängig vom verwendeten Compiler und der Zielplattform. Im *Praktikum Systemprogrammierung* wird der GCC Compiler verwendet. Die Tabelle 5.1 listet die verfügbaren Datentypen, ihre Größe und ihren Wertebereich auf. Da die verwendeten Mikrocontroller nur über wenige kB Speicher verfügen, empfiehlt es sich, auf große Datentypen wie `float` und `long` nach Möglichkeit zu verzichten.

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

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

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 5.1: Datentypen in 16-Bit Architekturen

zu deklarieren. Im Praktikum Systemprogrammierung wird diese Art der Deklaration verwendet.

Der Typ `void` kann nicht instanziert werden, sondern wird für Zeiger unbekannten oder beliebigen Typs verwendet. Zeiger werden in Abschnitt 5.1.7 vorgestellt.

LERNERFOLGSFRAGEN

- Sie wollen eine *Zeitspanne* von maximal einer Minute speichern, wobei Ihr Messinstrument nur auf eine Millisekunde genau auflöst.
 - Welcher Datentyp ist dafür am besten geeignet und warum?
 - Warum ist `float` dafür ungeeignet?
- Man verwendet bei Gleitkomma-Datentypen so gut wie niemals den (Un-)Gleichheitsoperator (`!=,==`).
 - Wieso ist dies sinnvoll?
 - Wie könnte man angemessener prüfen, ob zwei Zahlen gleich sind?

5.1.5 Deklaration und Initialisierung von Variablen

Variablen werden durch Angabe des Datentyps und eines Namens deklariert. Der Name muss im Sichtbarkeitsbereich der Variablen eindeutig sein. Die Initialisierung der Variable kann sofort zur Zeit der Deklaration oder später erfolgen. Listing 5.7 zeigt einige Beispiele.

Sichtbarkeitsbereich (Scope)

In Abhängigkeit von der Stelle im Quelltext, an der eine Variable deklariert wird, ändert sich ihr Sichtbarkeitsbereich (engl. *scope*).

1. Lokale Variablen

Wird eine Variable innerhalb eines Blocks deklariert, so ist sie nur dort sichtbar. Ein Block wird definiert, indem die zugehörigen Anweisungen in geschweifte Klammern gefasst werden. Beispielsweise stellt jede Funktion einen eigenen Block dar. Außerhalb eines Blocks ist eine zugehörige lokale Variable nicht sichtbar oder definiert.

Die C99-Spezifikation sieht vor, dass eine lokale Variable bei jedem Betreten des Blocks, in welchem sie definiert ist, neu initialisiert wird (Inpersistenz)¹. Statische Variablen können mit dem Schlüsselwort `static` deklariert werden und behalten auch nach Verlassen des Sichtbarkeitsbereiches ihren Wert. Listing 5.8 illustriert dies. In Abschnitt 5.1.5 wird näher auf statische Variablen eingegangen.

2. Globale Variablen

Variablen, die im gesamten deklarierenden Modul sichtbar sein sollen, werden im Modul, aber außerhalb von Funktionen deklariert. Üblicherweise steht die Deklaration am Anfang eines Moduls nach den `#include` Anweisungen. Globale Variablen sind in allen Funktionen des Moduls sichtbar, daher können sie von allen Funktionen gelesen und (sofern nicht `const`, siehe Abschnitt 5.1.12) geschrieben werden. Variablen können projektweit sichtbar gemacht werden, indem sie im Interface (.h-Datei) deklariert werden.

Storage Specifiers

Ein *Storage Specifier* ist ein Schlüsselwort, welches Eigenschaften einer Variable festlegt. Es ist wichtig zwischen diesen und den in Abschnitt 5.1.12 vorgestellten *Type Qualifiers* zu unterscheiden. Erstere beziehen sich auf die deklarierte Variable, wobei letztere sich auf ihren Typ beziehen.

Im Folgenden werden die wichtigsten Storage Specifier vorgestellt.

Das `static` Schlüsselwort Häufig sind Seiteneffekte bei Funktionen erwünscht, um Zustandsänderungen im (hardwarenahen) Programm widerzuspiegeln. Variablen, deren Sichtbarkeitsbereich verlassen wird, werden durch den Compiler aus dem Programmstack entfernt und verlieren daher ihren Wert. Daher verwendet man oft globale Variablen, um zustandsabhängige Funktionen zu ermöglichen. Dies hat jedoch den Nachteil, dass jede andere Funktion diese Variablen lesen und schreiben kann, was unerwünscht sein kann. Dazu gibt es den `static` Specifier, der den Compiler anweist eine lokale Variable nicht auf dem Programmstack, sondern im Datenbereich des Programms zu speichern.

¹Dies wird aktuell jedoch nicht von allen Compilern für nicht-Funktions-Blöcke unterstützt. Insbesondere sind Variablen teilweise außerhalb ihres Blocks sichtbar.

```

1 // Ohne Initialisierung
2 uint8_t var1;
3
4 // Mit Initialisierung (dezimal)
5 uint16_t var2 = -74;
6 // Mit Initialisierung (hex durch Voranstellen von 0x)
7 uint16_t var3 = 0x2A; // entspricht dezimal 42
8 // Mit Initialisierung (binär durch Voranstellen von 0b)
9 uint16_t var4 = 0b11010010; // Entspricht dezimal 210
10 // Mit Initialisierung (oktal durch Voranstellen von 0)
11 uint16_t var5 = 013; // Entspricht dezimal 11
12
13 // Mehrere Variablen anlegen
14 uint8_t var6, var7, var8;
15
16 // Mehrere Variablen initialisieren
17 uint8_t var9 = 2, var10 = 3, var11 = 7;

```

Listing 5.7: Variablen Deklaration mit und ohne Initialisierung

Dadurch bleibt der Wert dieser Variablen auch nach Verlassen ihres Sichtbarkeitsbereichs erhalten, ist jedoch nur in diesem Block sichtbar. In diesem Fall ist der sogenannte Gültigkeitsbereich echt größer als der Sichtbarkeitsbereich.

Das extern Schlüsselwort Wird eine globale Variable in mehreren Modulen (bzw. Dateien) verwendet, muss diese für jedes Modul deklariert werden. Dazu wird sie mit dem Schlüsselwort **extern** deklariert, was dem Compiler mitteilt, dass sich die Definition an einer anderen Stelle befindet, und diese erst beim Linken bekannt sein wird.

So kann eine eindeutige Variable mehrmals deklariert werden, ohne das der Compiler mehr als einmal Speicher für diese reserviert. Die Deklaration ohne das Schlüsselwort (Definition) sollte in dem Modul stattfinden, welches die Semantik bzgl. der Variable definiert.

```

1 extern int fluxCapacitor; // Variable wird nicht explizit
2   angelegt
3 int myFunction(void) {
4   int var3 = 47; // Nur innerhalb der Funktion sichtbar
5   static int var4 = 0; // Behält ihren Wert auch nach Verlassen
6     der Funktion
7   var3 = var3 + 1; // Hat immer den Wert 48 an dieser Stelle
8   var4 = var4 + 1; // Bei jedem Aufruf der Funktion wird var4
9     um 1 erhöht
10  fluxCapacitor = var3; // Schreibe 48
11  return var3+var4;
12 }
13 // ...
14 myFunction(); // Gibt (47+1)+(0+1) = 49 zurück
15 myFunction(); // Gibt (47+1)+(1+1) = 50 zurück
16 myFunction(); // Gibt (47+1)+(2+1) = 51 zurück
17 fluxCapacitor = 2; // Legaler Befehl
18 var4 = 40; // Illegaler Befehl, da var4 hier nicht sichtbar ist

```

Listing 5.8: Variablen Deklaration mit Storage Specifiers

LERNERFOLGSFRAGEN

- Was ist der Unterschied zwischen statischen, lokalen und globalen Variablen?
- Wovon hängt die Lebensdauer von Variablen ab und wovon ihre Sichtbarkeit?
- Welches Ergebnis erwarten Sie, wenn Sie eine Variable als `extern` deklarieren ohne sie an einer anderen Stelle ohne dieses Schlüsselwort zu deklarieren? Wieso kann dies kompiliert aber nicht gelinkt werden?

5.1.6 Typecasts

Zuweisungen der Art `VariableVonTyp1=VariableVonTyp2` sind unter Umständen Programmierfehler und werden vom Compiler abgefangen. Allerdings sind solche Zuweisungen manchmal beabsichtigt. Dann soll der Wert der zweiten Variablen als Wert vom Typ der ersten Variable *interpretiert* werden. Beispielsweise lassen sich `unsigned char`-Werte sowohl als Zeichen als auch als Zahl interpretieren. Durch einen expliziten Typecast wird verhindert, dass der Compiler eine solche Zuweisung als Fehler wertet. Bei kompatiblen Typen führt der Compiler implizite Typecasts durch. Bei expliziten Typecasts ist Vor-

sicht geboten, weil man die Typprüfung des Compilers umgeht. Listing 5.9 zeigt die Verwendung.

```

1 // Zwei Variablen verschieden Typs
2 signed char var1;
3 unsigned char var2 = 214;
4
5 // Erzeugt Warnung, da die "signedness" nicht übereinstimmt
6 var1 = var2;
7
8 // Typecasting:
9 var1 = (signed char)(var2); // var1 ist nun -42
10 /* Dabei wird der Wert von var2 als vorzeichenbehafteter Wert
   betrachtet, der den gültigen Wertebereich überschreitet und, da zwischen 128 und 255, eine negative
   Zahl darstellt */

```

Listing 5.9: Typecasting

LERNERFOLGSFRAGEN

- Was ist ein Typecast?
- Können Sie den Einsatz von Typecasts motivieren?
- In machen Fällen werden Ausdrücke implizit gecastet, ohne dass der Programmierer dies explizit hinschreiben muss. Wieso ist dies nicht immer der Fall bzw. sinnvoll?

5.1.7 Zeiger (Pointer)

In den vorhergehenden Abschnitten wurden Variablentypen und die zugehörigen Gültigkeitsbereiche eingeführt. Der Compiler übersetzt die Variablen in entsprechende Adressen im Speicher und nutzt diese direkt. Entsprechend gibt es in C die Möglichkeit, direkt auf Speicherbereiche zuzugreifen bzw. sich den zu einer Variable gehörigen Speicherbereich ausgeben zu lassen. Diese Adresse wird als *Zeiger* auf die entsprechende Variable bezeichnet.

Wird einer Variablen Deklaration ein Stern hinzugefügt (also z.B. `char *ptr;`), wird ein Zeiger erzeugt. Der Compiler erwartet, dass in diesem Zeiger eine Adresse auf eine Variable vom Typ der Definition (hier `char`) abgelegt wird. Man kann sowohl die im

Zeiger gespeicherte Adresse verändern, als auch auf den an der Adresse hinterlegten Wert zugreifen. Die Verwendung wird in Listing 5.10 veranschaulicht. Bei der Verwendung von Zeigern spielen zwei Operatoren eine wichtige Rolle:

- & liefert eine Speicheradresse („Referenzierung“). Es wird also nicht direkt auf die Variable/Funktion/etc. zugegriffen, stattdessen wird ihre Adresse im Speicher zurückgeliefert.
- * liest das Ziel eines Zeigers aus („Dereferenzierung“). Damit wird direkt auf die Speicherstelle zugegriffen, auf die der Pointer zeigt.

Während das Zeichen * bei der Variablen Deklaration den Typ der deklarierten Variablen angibt, bildet der ** Operator* mit der Variable einen Ausdruck, und zwar die Dereferenzierung einer Variablen.

```

1 // Normale Variable
2 int var = 168;
3
4 // Pointer auf Variable var
5 int *p = &var;
6
7 // Wert an der Adresse, auf die der Pointer zeigt
8 // (* ist hier keine Multiplikation!)
9 int wertAnPositionP = *p;
10
11 /* Wert an der nächsten Adresse (Adresse auf die der Pointer
   zeigt + 1). Der Inhalt an Stelle p+1 ist möglicherweise
   undefiniert. */
12 int wertAnPositionPplus1 = *(p+1);
13
14 // var den Wert 87 zuweisen
15 *p = 87;
```

Listing 5.10: Zeiger

LERNERFOLGSFRAGEN

- Was ist ein Zeiger (Pointer) auf eine Variable?
 - Betrachten Sie den Code in Listing 5.11.
 - Wieso wird diese Funktion mit hoher Wahrscheinlichkeit zu Fehlern führen?
 - Können Sie die Funktion korrigieren, so dass sie eine gültige Speicherstelle zurückliefert? (Es ist nicht notwendig, dass bei jedem Aufruf eine neue Speicherstelle zurückgegeben wird)
- Hinweis: Siehe Abschnitt 5.1.5.
- Was versteht man im Zusammenhang mit Zeigern unter den Begriffen „Referenzierung“ und „Dereferenzierung“?

```
1 int *newInt(int initialValue) {  
2     int val = initialValue;  
3     int *ptr = &val;  
4     return ptr;  
5 }
```

Listing 5.11: Fehlerhafter Code

5.1.8 Funktionszeiger

Funktionszeiger sind eine Teilkasse von Zeigern: Sie enthalten die Adressen von Funktionen anstatt der von Variablen. Es ist wichtig zu beachten, dass beim ATmega 644 der Speicher nach dem Konzept der Harvard-Architektur aufgebaut ist. Das bedeutet, dass der Programmspeicher vom Datenspeicher getrennt ist. Daher haben Funktionszeiger und Zeiger unterschiedliche Bezugssysteme. Die Verwendung von Funktionspointern ist in Listing 5.12 dargestellt.

Wie man sehen kann, ist die explizite Dereferenzierung von Funktionszeigern nicht nötig, da C für das Aufrufen von referenzierten Funktionen eine Kurzschreibweise erlaubt.

```

1 long funktion1(long m, long n) {
2     if (m > 0) {
3         if (n > 0)
4             return funktion1(m-1,funktion1(m,n-1));
5         else
6             return funktion1(m-1,1);
7     } else
8         return n+1;
9 }
10
11 // Zeiger auf funktion1 anlegen
12 long (*zeiger)(long, long) = &funktion1;
13 long a; // Variable für das Ergebnis
14 a = (*zeiger)(7,6); // Funktionsaufruf (explizite Deref.)
15 a = zeiger(7,6);    // Funktionsaufruf (Kurzschreibweise)

```

Listing 5.12: Zeiger auf Funktion

LERNERFOLGSFRAGEN

- Können Sie ein Einsatzgebiet für Funktionszeiger angeben?
- Angenommen, Sie arbeiten auf einem ATmega 644, und haben einen Zeiger `fp` der auf eine Funktion zeigt und einen Zeiger `p` der auf eine Variable zeigt. Wieso kann es sein, dass der Wert beider Zeiger gleich ist (also `fp == p`)?
- Sie haben in der Vorlesung *Programmierung* das Konzept objektorientierter Programmierung kennengelernt. Können Sie ein Kernkonzept der OOP angeben, welches Gebrauch von Funktionszeigern macht?

5.1.9 Arrays

Von jedem Datentyp lassen sich nicht nur einzelne Variablen, sondern auch Arrays (Felder) erzeugen. Felder unterscheiden sich von den bisher vorgestellten Variablen lediglich darin, dass sie einen Index besitzen. Auf diese Weise können zusammenhängende Daten leichter behandelt werden.

Die Verwendung von Arrays ist sinnvoll für tabellen- oder listenartige Daten. Insbesondere bei einer größeren Anzahl an Datensätzen ist es vom Programmieraufwand her leichter über ein Array zu iterieren, als auf jede Variable einzeln zuzugreifen. Ein Array kann mehr als eine Dimension haben. Listing 5.13 zeigt ein Beispiel. Beachten Sie, dass

der Index eines Arrays immer bei 0 beginnt.

```

1 // Deklaration
2 char text[4];      // Eindimensionales Array
3 int matrix[2][3];   // Zweidimensionales Array
4
5 // Initialisierung per Zugriff auf einzelne Elemente
6 text[0] = 'i';
7 text[1] = 'x';
8 text[2] = 'i';
9 text[3] = 0;
10 matrix[0][0] = 5;
11 matrix[0][1] = 7;
12 matrix[1][0] = 11;
13 // ...
14
15 // Deklaration und Initialisierung in einem Schritt
16 char text2[4] = {'i', 'x', 'i', 0};
17 char text4[4];
18 int matrix2[2][3] = {{1, 2, 3}, {0, -1, 7},};
19
20 // Iteration über ein Array. Die verwendete
21 // for-Kontrollstruktur dient nur zur Veranschaulichung
22 // und wird später näher erläutert.
23 unsigned int i;
24 for (i = 0; i < 4; i++) {
25     text4[i] = text1[i];
26 }
```

Listing 5.13: Arrays

Der Compiler übersetzt Arrays, indem er feste Bereiche im Speicher für das Array reserviert und in den Arrayvariablen die erste Adresse des reservierten Speicherbereichs ablegt. Somit sind Arrays eine Kurzschreibweise für die Verwendung von Pointern auf festen Speicherbereichen.

LERNERFOLGSFRAGEN

- Wieviel Speicher belegt ein Array von 21 `short` Werten?
- Was ist der Unterschied zwischen `int foo[12];` und `foo[12];`? Welche Gefahren bestehen hier im Zusammenhang mit dem Index?

5.1.10 Zeichenketten (Strings)

Einzelne Zeichen lassen sich als Variable vom Typ `char` speichern. Dabei wird der zugehörige ASCII-Wert des zu speichernden Buchstabens in einer 8 Bit-Speicherstelle abgelegt.

```

1 char c = 'A'; // Initialisierung mit dem Zeichen A
2 c = 98;           // Zuweisung des Zeichens b (ASCII)
3 c = c - 'a';    // Zuweisung der Zahl 1 (nicht des Zeichens 1)
```

Listing 5.14: Zeichen

Falls mehr als ein Zeichen (also eine Zeichenkette) abgelegt werden soll, hat man die Möglichkeit dies in einem Array vom Typ `char` zu machen. Dabei kann die Länge manuell bei der Deklaration angegeben werden oder leer gelassen werden. Im letzten Fall ermittelt der Compiler selbst die Länge der Zeichenkette. Die Besonderheit im Vergleich zu anderen Arrays liegt in der sogenannten Nullterminierung. Das letzte Element des `char`-Arrays ist eine Null, die das Ende der Zeichenkette markiert. Das Erzeugen einer Zeichenkette ist in Listing 5.15 veranschaulicht.

```

1 // 13 Byte (für 12 Zeichen) Speicher reservieren
2 char strHelloWorld[] = "Hello World!";
```

Listing 5.15: Erzeugen einer Zeichenkette

In Anbetracht der oft knappen Ressourcen von Mikrocontrollern sind Strings vergleichsweise speicherintensiv, da sie als Array von `char`-Datentypen angelegt werden. Viele Zeichenketten, wie z. B. die Ausgaben auf dem LCD ändern sich zur Laufzeit jedoch nicht. Daher empfiehlt es sich konstante Zeichenketten mit dem Schlüsselwort `PROGMEM` `char` zu deklarieren. Dadurch legt der Compiler diese nicht im SRAM, sondern im Flash-Speicher ab. Auf dem ATmega 644 steht erheblich mehr Flash-Speicher zur Verfügung als SRAM. Dieses Vorgehen hilft Speicherplatz einzusparen. Listing 5.16 veranschaulicht das Erzeugen von Zeichenketten.

```

1 // Im Header:
2 extern const PROGMEM char strHelloWorld[];
3 // In einer C-Datei:
4 const PROGMEM char strHelloWorld[] = "Hello World!";
```

Listing 5.16: Ablegen einer Zeichenkette im Programmspeicher

LERNERFOLGSFRAGEN

- Der Datentyp `char*` stellt einen Zeiger auf eine Variable dar, die genau ein Zeichen speichern kann.
 - Warum wird `char*` aber auch für Zeichenketten beliebiger Länge verwendet?
 - Woran erkennt man bei einer solchen Zeichenkette die Länge?
- Wozu gibt es den Typ `PROGMEM char`?
 - Welchen Unterschied gibt es hier im Vergleich zum Typ `char`?
 - Angenommen Sie haben eine Variable `PROGMEM char str[10]`. Ist es sinnvoll auf `str[3]` zuzugreifen? Wieso (nicht)?

5.1.11 Eigene Datentypen erstellen

Die C-Syntax stellt mehrere Schlüsselwörter zur Verfügung, mit denen neue Datentypen erstellt werden können.

Zusammengesetzte Datentypen

Mit dem Schlüsselwort `struct` kann ein neuer Datentyp erzeugt werden, der aus anderen Datentypen zusammengesetzt ist. Ein einfaches Beispiel ist in Listing 5.17 dargestellt. Die Deklaration von Variablen benötigt ebenfalls das Schlüsselwort. Der Zugriff auf die Attribute des Datentypen erfolgt über den Punktoperator, wie in Listing 5.18 gezeigt.

```

1 struct Student {
2     int immatriculation;
3     unsigned long matrNumber;
4 };

```

Listing 5.17: Erzeugen einer zusammengesetzten Datenstruktur

Darüber hinaus gibt es in C den Strukturoperator `->`. Mit ihm ist es möglich über einen Zeiger auf die einzelnen Attribute einer Struktur direkt (ohne den Dereferenzierungsoperator) zuzugreifen. Wie in Listing 5.19 zu sehen ist, kann auf diese Weise eine in einer Struktur hinterlegte Funktion aufgerufen werden.

```
1 // Deklarieren
2 struct Student arthur;
3 // Zugriff
4 arthur.immatriculation = 1979;
5 arthur.matrNumber = 123456;
6
7 // Weitere Variable deklarieren
8 struct Student tricia;
9 // Zugriff
10 tricia.immatriculation = 1979;
11 tricia.matrNumber = 123457;
```

Listing 5.18: Beispiel für die Verwendung zusammengesetzter Datentypen

```
1 struct MyClassStruct {
2     int attr1;
3     void (*attrFunc)(void);
4 };
5 // Erzeugung einer Instanz
6 struct MyClassStruct obj;
7 // Zeiger auf diese Instanz
8 struct MyClassStruct* p = &obj;
9 // Zugriff auf ein Attribut (hier: attr1)
10 (*p).attr1 = 5;
11 // Kurzschreibweise für den Zugriff auf attr1
12 p->attr1 = 5;
13 // Aufruf der attrFunc Methode
14 ((*p).attrFunc)();
15 // Kurzschreibweise des Aufrufs der attrFunc Methode
16 p->attrFunc();
```

Listing 5.19: Verwendung des Strukturoperators

Aufzählungstypen

Die C-Syntax stellt das Schlüsselwort `enum` bereit, um eigene Aufzählungstypen zu definieren. Solche Aufzählungen vermeiden Verwechslungen ihrer Elemente und besitzen einen eigenen Wertebereich. Dadurch wird der Quelltext meist deutlich besser lesbar. Eine einfache Definition und die Verwendung des Typs sind in Listing 5.20 dargestellt. Aus Gründen der Speichereffizienz werden diese Typen vom Compiler intern in primitive/numerische Datentypen aufgelöst. Diese automatische Auflösung kann man beeinflussen und den numerischen Wert *optional* von Hand zuweisen, wie die Listings zeigen.

```

1 enum FehlerTypen {
2     ERROR_MEMORY,           // == 0
3     ERROR_OUTPUT,          // == 1
4     // ERROR_FILE wird explizit ein Wert zugewiesen
5     ERROR_FILE = 5,        // == 5
6     ERROR_NONE             // == 6
7 };
8
9 // Deklaration
10 enum FehlerTypen fehler;
11
12 // Initialisierung
13 fehler = ERROR_NONE;

```

Listing 5.20: Deklaration eines Aufzählungstypen

Typnamen

In der Sprache C gibt es die Möglichkeit, mit Hilfe der Anweisung `typedef` neue Datentypen zu definieren. Diese neuen Datentypen werden in der Regel verwendet, um den Quellcode besser lesbar zu machen. Außerdem ist es durch diesen Befehl möglich, die Deklaration von zusammengesetzten Datentypen zu verkürzen, so dass nicht immer das Schlüsselwort `struct` bei einer Deklaration benötigt wird. Die Verwendung der Anweisung `typedef` wird in Listing 5.21 veranschaulicht.

```

1 // Ganzzahl als int definieren
2 typedef int Ganzzahl;
3 // Erlaubt:
4 Ganzzahl var1 = 42;

```

Listing 5.21: Verwendung von `typedef`

```

1 // Struktur definieren
2 typedef struct Bsp {
3     char attr;
4 } Beispiel;
5 // Erlaubt:
6 Beispiel var2;
7 // Äquivalent zu:
8 struct Bsp var3;

```

Listing 5.22: Verwendung von **typedef** bei **structs**

Da **struct** Typen einen eigenen Namensraum besitzen, ist sogar der in Listing 5.23 gezeigte **typedef** legal und äquivalent zu 5.22.

```

1 // Struktur definieren
2 typedef struct Beispiel {
3     char attr;
4 } Beispiel;
5 // Erlaubt:
6 Beispiel var2;
7 // Äquivalent zu:
8 struct Beispiel var3;

```

Listing 5.23: Namensraum von **typedef** und **struct**

Vereinigung

Das Schlüsselwort **union** erlaubt es verschiedene Typen zu einem zusammenzufassen. Syntaktisch sieht dies aus wie eine **struct**-Definition, allerdings ist die Größe des Verbundes nur so groß wie der größte enthaltene Typ. Da alle Attribute an derselben Speicheradresse beginnen, können diese nicht unabhängig geändert werden. Listing 5.24 zeigt einen Verbundtypen für eine IPv4-Adresse.

Unions dienen unter anderem dazu, Werte von Typen umzuinterpretieren, ähnlich wie explizite Typecasts, da die Attribute einer **union** den selben Speicherplatz belegen, was im Vergleich zum syntaktisch ähnlichen **struct** weniger Speicherplatz benötigt.

```

1 union IpAddr {
2     unsigned char block[4];
3     unsigned long ip;
4     unsigned char first_block;
5 };
6 IpAddr localhost = {{127,0,0,1}};
7 IpAddr lh2;
8 lh2.ip = localhost.ip; // == 0x0100007F
9 // lh2.block == {127,0,0,1}
10 // lh2.first_block == 127

```

Listing 5.24: union Beispiel

LERNERFOLGSFRAGEN

- Angenommen, Sie wollen eine rudimentäre Klasse implementieren, wie sie etwa in C++ oder Java implementiert ist, welches hier vorgestellte Datenprimitiv eignet sich dafür?
- Die Verwendung eines `enums` definiert mehrere Bezeichner, welche alle zu einer Zahl auflösen. Welchen Vorteil sehen Sie im Vergleich zur Verwendung mehrerer `#define` Direktiven? Sind beide Ansätze gleich ausdrucksstark?
- Sowohl `structs` wie auch `unions` können mehrere Attribute besitzen. Was ist der signifikante Unterschied zwischen diesen Primitiven?
- Eine `union` wird häufig auch als *besserer* - oder *sicherer Cast* bezeichnet. Haben Sie eine Erklärung dafür?
- Warum ist es nicht möglich in Beispiel 5.24 mit einem Union Attribut `second_block` direkt auf den zweiten Eintrag von `block` in `IpAddr` zuzugreifen?

5.1.12 Type qualifiers

Type Qualifiers sind Schlüsselwörter, mit denen sich Eigenschaften von Typen ausdrücken lassen. Diese lassen sich auch in zusammengesetzten Typen, etwa Pointern, auf einzelne Teile des Typs anwenden. Hierbei ist darauf zu achten, dass C es erlaubt, das Schlüsselwort links oder rechts des eigentlichen Typs zu notieren. Zunächst sollen zwei wichtige Qualifier vorgestellt werden, um dann auf die spezielle Verwendung mit zusammengesetzten Typen eingehen zu können.

volatile

Beim Debuggen ergibt sich im Zusammenhang mit Variablen ein wichtiges Problem: Der Compiler optimiert den Code, was sehr häufig das Wegfallen einiger Variablen bedeutet, falls dadurch effizienterer und äquivalenter Code entsteht. Diese können dann beim Debuggen nicht mehr betrachtet und ausgelesen werden. Das Schlüsselwort **volatile** schließt die deklarierte Variable von allen Optimierungen des Compilers aus. Typischerweise wird dies in Verbindung mit IO-Registern verwendet, bei der sich der Wert einer Variable ändern kann, ohne dass dies für den Compiler anhand des geschriebenen Codes ersichtlich wäre.

const

Eine Variable kann mit dem Schlüsselwort **const** als konstant markiert werden. Dadurch wird dem Compiler mitgeteilt, dass der mit dieser Variable assoziierte Speicherbereich nicht unter Verwendung dieser Variable modifiziert werden darf. Listing 5.25 erläutert die Wirkung dieses Schlüsselworts.

```

1 // Die Variable x hat den Typ 'int const' und wird mit 27
2     initialisiert
3 // 'int const' ist exakt der gleiche Typ wie 'const int'
4 int const x = 27;
5 // Der Versuch die Zahl 15 an x zuzuweisen wird vom Compiler
6     mit einem Fehler quittiert
7 x = 15; // Illegal
8 // Die Variable y hat den Typ 'int', ist also nicht konstant
9 int y = 0;
10 // Der Wert von x kann jedoch gelesen werden und in eine nicht-
11     konstante Variable kopiert werden
y = x;
12 // Dadurch behält y ihren Typ, ist also weiterhin änderbar
y += 15;

```

Listing 5.25: Konstante Variablen erlauben keine Zuweisungen.

Zusammengesetzte Typen mit Qualifiern

Um einen Zeiger zu qualifizieren, gibt es zwei Möglichkeiten. Zunächst kann der Zeiger selbst konstant sein, oder aber auf einen konstanten Wert zeigen. Hierbei ist die Seite, auf der der Type Qualifier notiert wird, relevant. Beispielsweise ist bei der Deklaration **const int* var** intuitiv nicht klar, ob nun **var** konstant ist, oder ob ***var** konstant ist, **var** also auf einen konstanten Speicherabschnitt zeigt (das erste ist korrekt). Daher ist es empfehlenswert, den Qualifier auf der rechten Seite des qualifizierten Typs zu notieren. Verschiedene qualifizierte Typen werden in Listing 5.26 dargestellt.

```

1 // Zwei Konstanten, initialisiert mit 13, bzw. 29
2 int const x = 13;
3 int const y = 29;
4 // Eine Variable
5 int z = 0;
6 // Ein veränderlicher Zeiger auf eine Konstante
7 int const* px = &x;
8 // Ein konstanter Zeiger auf eine Konstante
9 int const* const py = &y;
10 // Das Ändern von px ist legal, da px selbst nicht konstant ist
11 px = py;
12 // Modifizieren des Wertes auf den px zeigt ist jedoch nicht mö
13     glich
14 *px = 89; // Illegal!
15 // Das Ändern von py ist NICHT legal, da py konstant ist
16 py = px; // Illegal!
17 // Ein konstanter Zeiger auf eine veränderliche Variable
18 int* const pz = &z;
19 *pz = 89; // Legal!
20 pz = 0; // Illegal!
21 /* Ein von der Optimierung ausgeschlossener Funktionszeiger,
   der auf eine Funktion zeigt, die einen Zeiger auf einen
   konstanten char und einen int erwartet. Der Rückgabewert ist
   ein Pointer auf einen konstanten Pointer, welcher auf einen
   veränderlichen short zeigt. */
short* const* (*volatile func)(char const* c, int i);

```

Listing 5.26: Zusammengesetzte Typen mit Type Qualifiers

Die Variablen Deklaration und damit auch die korrekte Aussprache komplizierter Konstrukte orientiert sich an der Hierarchie der C-Operatoren. Eine hervorragende Erklärung hat T.Birnthal in *C-Deklarationen lesen und schreiben* verfasst.²

²<http://www.ostc.de/c-declaration.pdf>

LERNERFOLGSFRAGEN

- Was ist der generelle Unterschied zwischen *Type Qualifiern* und *Storage Specifiern*?
- Welche Type Qualifier sind Ihnen bekannt?
- Mithilfe der `#define` Direktive kann man bereits Konstanten definieren, ohne Platz im Datenbereich des laufenden Programms zu belegen.
 - Wozu braucht man den `const` Qualifier?
 - Was ist der Unterschied zwischen `const` Variablen und `#define` Direktiven?
- Können Sie einen Typ angeben, in dem das `const` Schlüsselwort (als Schlüsselwort) vorkommt, der aber bei einer Variablen Deklaration eine nicht-konstante Variable spezifiziert?
- Es ist sehr verbreitet, Type Qualifier auf der linken Seite des qualifizierten Typs anzugeben.
 - Können Sie erklären, warum dies eine fragwürdige Praxis ist?
 - Welche Probleme können dadurch entstehen?
 - Welche Alternativen gibt es?
- Gibt es Unterschiede zwischen den folgenden drei Typen?
 1. `const int*`
 2. `int const*`
 3. `int* const`

5.1.13 Operatoren

Die Sprache C definiert einige Operatoren, um mathematische bzw. logische Operationen sowie Variablenzugriffe durchzuführen. Im Folgenden werden die wichtigsten dieser Operatoren jeweils mit einem kurzen Beispiel vorgestellt. In allen Beispielen wird angenommen, dass die Variable `unsigned char var25` den Wert 25 bzw. `0b00011001` besitzt.

Der = Operator

Mit Hilfe des Zuweisungsoperators kann z. B. einer Variablen ein Wert zugewiesen werden.

Beispiel: `var = 76; // var wird 76 zugewiesen`

Der == Operator

Mit Hilfe des Vergleichsoperators `==` kann die Gleichheit zweier Operanden überprüft werden. Dieser Operator wird häufig innerhalb von bedingten Anweisungen verwendet.

Beispiel: `var = 25 == 5; // var ist 0, da 25 nicht gleich 5 ist`

Der != Operator

Mit Hilfe des Vergleichsoperators `!=` kann die Ungleichheit zweier Operanden überprüft werden. Dieser Operator wird häufig innerhalb von bedingten Anweisungen verwendet.

Beispiel: `var = 25 != 5; // var ist 1, da 25 ungleich 5 ist`

Der > bzw. < Operator

Mit Hilfe der Vergleichsoperatoren `>=`, `>`, `<=` und `<` kann die Größe von zwei Operanden verglichen werden.

Beispiel: `var = 25 < 7; // var ist 0, da 25 nicht kleiner ist als 7`

Arithmetische Operatoren

Die arithmetischen Operatoren `+`, `-` und `*` entsprechen den gleichnamigen Abbildungen des zugehörigen Restklassenrings $\mathbb{Z}/2^{2^n}\mathbb{Z}$ mit $n = 3,4,5,6$ (`char`, `short`, `long`, `long long`). Der `/` Operator bildet hier eine Ausnahme, er berechnet das reelle Ergebnis und runden es nach unten ab.

Beispiel: `var = 103 / 4; // var ist 25, da 103/4 gleich 25.75 ist`

Mithilfe von `/` lässt sich auch eine aufrundende und kaufmännisch runde Division implementieren, indem zuvor auf den Dividenden der Divisor-1 bzw. der Divisor/2 (falls dieser gerade ist) addiert wird. Hierfür gibt es jedoch auch eine umfangreiche Sammlung mathematischer Funktionen in `math.h`.

Der % Operator

Mit Hilfe des Modulo-Operators `%` wird der Rest einer ganzzahligen Division berechnet.

Beispiel: `var = 25 % 2; // var ist 1, da 1 kongruent 25 mod 2 ist und 1<2`

Der ++ bzw. -- Operator

Mit Hilfe des Inkrement-Operators `++` bzw. des Dekrement-Operators `--` kann eine Variable um eins erhöht bzw. verringert werden. Dieser Operator kann sowohl vor als auch hinter seinem Operanden stehen. Steht der Operator vor dem Operand, spricht man von einem Präinkrement bzw. -decrement, steht er hinter dem Operand, bezeichnet man ihn als Postinkrement bzw. -decrement. Bei Verwendung der Prä-Variante wird die Operation ausgeführt **bevor** der Wert des Ausdrucks bestimmt wird, bei der Post-Variante wird zunächst der Wert des Ausdrucks bestimmt und **danach** die Operation ausgeführt.

Beispiel: `var = var25++; // var ist 25, var25 erhält den Wert 26`

Beispiel: `var = ++var25; // var und var25 sind 26`

Der << bzw. >> Operator

Mit Hilfe des Bitshiftoperators wird der Inhalt eines Operanden (*nicht-zyklisch*) bitweise nach links bzw. rechts verschoben.

Beispiel: `var = 0b11001 >> 2; // var ist 0b110 (6)`

Nach dieser Zuweisung hat die Variable `var2` den Wert `0b00000110`. Der angegebene Wert wurde um 2 Stellen nach rechts verschoben. Dadurch sind die beiden niedrigstwertigen Bits (engl. least significant bits - lsb) verloren gegangen. Von links wurden Nullen aufgefüllt.

Der & Operator

Mit Hilfe des bitweisen Und-Operators werden zwei Operanden bitweise miteinander verundet.

Beispiel: `var = 0b11001 & 0b1111; // var ist 0b1001 (9)`

Der | Operator

Mit Hilfe des bitweisen Oder-Operators werden zwei Operanden bitweise miteinander verodert.

Beispiel: `var = 0b11001 | 0b1111; // var ist 0b1111 (31)`

Der ^ Operator

Mit Hilfe des bitweisen exklusiven Oder-Operators werden zwei Operanden bitweise exklusiv miteinander verodert.

Beispiel: `var = 0b11001 ^ 0b1111; // var ist 0b10110 (22)`

Der ~ Operator

Mit Hilfe der bitweisen Negation wird ein Operand bitweise invertiert.

Beispiel: `var = ~0b11001; // var ist 0b11100110`

Siehe zu diesem Operator die weiterführenden Hinweise in Abschnitt 5.3.7.

Der && Operator

Mit Hilfe des logischen Und-Operators werden zwei Ausdrücke miteinander verknüpft. Im Unterschied zur bitweisen Operation unterscheidet `&&` nur zwischen „0“ und „nicht 0“ für beide Operanden.

Beispiel: `var = 8 && 1; // var ist 1`

Daher ist `a && b` das gleiche wie `(a != 0) & (b != 0)`.

Der || Operator

Mit Hilfe des logischen Oder-Operators werden zwei Ausdrücke miteinander verknüpft. Analog zu `&&` unterscheidet `||` nur zwischen „0“ und „nicht 0“.

Beispiel: `var = 8 || 0; // var ist 1`

Wie `&&` lässt sich auch `||` durch das bitweise Oder ausdrücken. `a || b` ist das gleiche wie `(a != 0) | (b != 0)`.

Der ! Operator

Mit Hilfe der binären Negation wird ein Wert logisch invertiert. Dabei wird die 0 auf 1 abgebildet und jeder andere Wert auf 0.

Beispiel: `var = !8; // var ist 0`

`!a` ist das gleiche wie `a == 0`.

Der ?: Operator

Mit Hilfe des konditionalen Operators wird anhand einer Bedingung aus zwei Operanden ein Wert ausgewählt. Ist die Bedingung erfüllt, wird der erste ausgewählt, ansonsten der zweite.

Beispiel: `var = 0 ? 5 : 25; // var ist 25`

Kurzformen von Zuweisungen

Es ist möglich eine Zuweisung und eine weitere Operation mit dem Operanden der Zuweisung in einer Zeile durchzuführen. Dazu wird der Operator dem Zuweisungsoperator vorangestellt (z. B. `+=`).

Beispiel: `var += 17; // Äquivalent zu var = var + 17;`

Analog existieren Kurzformen zu allen binären nicht-logischen Operatoren: `-=`, `*=`, `%=`, `&=`, etc.

Referenzierung &

Die Referenzierung liefert die Speicheradresse einer Variablen oder einer Funktion zurück.

Beispiel: `unsigned char *ptr = &var25; // ptr zeigt auf var25`

Dereferenzierung *

Die Dereferenzierung ermöglicht den direkten Zugriff auf den Speicher.

Beispiel: `unsigned char var = *ptr; // var ist 25 falls ptr auf var25 zeigt`

Strukturoperator ->

Mit Hilfe der Strukturoperatoren `.` und `->` kann auf Elemente einer Struktur bzw. eines Zeigers auf eine Struktur zugegriffen werden.

Beispiel: `myStructPtr->attr = myStruct.attr; // Kopiere Attribut von einer Strukturinstanz in eine möglicherweise andere`

Operator	Funktion
.	Strukturoperator
->	Strukturoperator für Zeiger
++, --	Inkrement bzw. Dekrement
~	Bitweise Negation
!	Logische Negation
&	Referenzierung
*	Dereferenzierung
%	Modulo Operator
*, /	Arithmetische Operatoren
+, -	Arithmetische Operatoren
>>, <<	Bitweiser Shift
>=, >, <=, <	Größer/Kleiner (gleich) Vergleich
==, !=	Test auf (Un-)Gleichheit
&	Bitweises Und
^	Bitweises exklusives Oder
=	Bitweises Oder
&&	Logisches Und
	Logisches Oder
? :	Konditionaler Operator
=	Zuweisung
+ =, usw.	Kurzform Zuweisung $x = x + y$

Tabelle 5.2: Operatoren

Zusammenfassung

Die Tabelle 5.2 zeigt eine Kurzzusammenfassung der zuvor vorgestellten Operatoren in absteigender Präzedenz (erstgenannte werden vor letztgenannten ausgeführt - falls ungeklemmt).

LERNERFOLGSFRAGEN

- Eine Zuweisung (=) ist im Gegensatz zu vielen anderen Sprachen in C ebenfalls ein Ausdruck.
 - Welche Vorteile sehen Sie darin?
 - Welche Gefahr besteht im Zusammenhang mit Abfragen auf Gleichheit dadurch?
- Wie unterscheiden sich die Strukturoperatoren -> und .?
- Gibt es einen Unterschied zwischen den Operatoren & und && (beziehungsweise | und ||, oder ~ und !)? Können Sie alle Werte (für einen festen Datentyp) angeben, für die sich die Operatoren *jeweils* gleich verhalten?
- Was ist der Unterschied zwischen dem Ausdruck `result*=var` und dem Ausdruck `result=var*`? Falls `result` den Typ `int` hat, welchen Typ muss dann jeweils `var` haben?
- Gibt es einen *logischen* Xor Operator? Falls ja, welchen? Falls nein, können Sie diese Operation aus den Operatoren in C konstruieren?
- Können Sie $(a \& b) + (a \sim b) == (a \mid b)$ nachweisen?

5.1.14 Funktionen

Die C-Syntax erlaubt die Implementierung von Funktionen, die das Programm in logische Bereiche aufteilen. Dabei besteht eine Funktion aus einem eindeutigen Namen, einer Parameterliste mit Vorbedingungen (Funktionssignatur) und einem Rückgabewert, sowie dem eigentlichen Funktionskörper, der die Funktionalität implementiert. Mit dem Schlüsselwort `void` kann auf Parameter oder Rückgabewert verzichtet werden. Mit dem Schlüsselwort `return` kann die Funktion zu jedem Zeitpunkt verlassen werden. Ein eventueller Rückgabewert muss dem Schlüsselwort nachgestellt werden. Nach Beendigung einer Funktion kehrt der Kontrollfluss an die Codestelle zurück, welche die Funktion aufgerufen hat. Listing 5.27 verdeutlicht den Aufbau einer Funktion.

C gehört zu den Programmiersprachen, die eine explizite Deklaration von Funktionen verlangen. Anders als beispielsweise in Java reicht es nicht aus, eine Funktion nur zu implementieren, damit sie anderen Funktionen bekannt ist. Das zeigt sich dadurch, dass in Listing 5.27 die Funktion `main` die Funktion `istGerade` nicht kennt, weil `istGerade` erst nach `main` deklariert wird. Abhilfe schaffen sogenannte Forward-Deklarationen. Dazu wird der Funktionsrumpf zu Beginn der .c-Datei aufgeführt und die eigentliche Implementierung kann später folgen. Dies ist z.B. notwendig, falls nicht mit Header-Dateien

```

1 int main(void) {
2     unsigned n = 5;
3     // Rufe andere Funktion auf
4     char gerade = istGerade(n);
5     return 0;
6 }
7
8 char istGerade(unsigned wert) {
9     if (wert % 2 == 0) {
10         return 'j';
11     } else {
12         return 'n';
13     }
14 }
```

Listing 5.27: Beispiel einer Funktion und ihres Aufrufs

gearbeitet wird und zyklische Funktionsaufrufe vorkommen. Für das obige Beispiel würde also die in Listing 5.28 genannten Zeilen am Anfang von Listing 5.27 genügen, um der Funktion `main` die Funktion `istGerade` bekannt zu machen.

```

1 char istGerade(unsigned wert);
2 int main(void);
```

Listing 5.28: Forward-Deklaration von Funktionen

Bei der Betrachtung der Funktionssignatur parameterloser Funktionen wie `main` fällt auf, dass die leere Parameterliste, im Gegensatz zu vielen Hochsprachen mit C-ähnlicher Syntax wie Java oder C++, nicht mit einem leeren Klammerpaar „()“ sondern mit „`(void)`“ definiert werden. Das leere Klammerpaar deutet in C eine Parameterliste mit *beliebig* vielen Argumenten an, welche in hardwarenahen Anwendungen selten sinnvollen Einsatz findet.

LERNERFOLGSFRAGEN

- Was versteht man unter einer Forward-Deklaration? Wieso ist es sinnvoll, dass diese notwendig sind?
- Wie unterscheiden sich Funktionen welche in der Headerdatei deklariert wurden von solchen, die es nicht wurden?
- Was ist der Unterschied zwischen den zwei Funktionsdeklarationen `void f1(void);` und `void f2();`? Wieviele Argumente erwarten diese Funktionen jeweils?
- Sind die Variablen in der Parameterliste einer Funktion *globale Variablen* oder *lokale Variablen*?

5.1.15 Kontrollstrukturen

Kontrollstrukturen erlauben es dem Programm, auf verschiedene Werte in Variablen oder Eingaben unterschiedlich zu reagieren. Beachten Sie, dass bei Bedingungen in C nicht nur die beiden Werte „wahr“ und „falsch“ existieren, sondern prinzipiell *jeder* Wert (primitiver Datentypen wie `int`, `char*`, `float`, ...) genutzt werden kann. Werte ungleich 0 werden als „wahr“, Werte gleich 0 als „falsch“ interpretiert.

Einfache bedingte Verzweigung

Die mit dem Schlüsselwort `if` eingeleitete Kontrollstruktur erlaubt eine simple Verzweigung anhand eines logischen Ausdrucks. Ist die Bedingung erfüllt, so wird die dem `if` nachfolgende Anweisung bzw. der nachfolgende Block ausgeführt. Zusätzlich kann mit dem `else` Schlüsselwort eine Alternative zum `if`-Block angegeben werden, welche genau dann ausgeführt wird, wenn die Bedingung nicht erfüllt ist. Beispiele sind in Listing 5.29 dargestellt.

Obwohl stilistisch nicht ganz korrekt, also den allgemein üblichen Konventionen nicht entsprechend, hat sich die Notation `else if` (ohne Absatz) durchgesetzt, um mehrere Bedingungen zu testen. Ein Beispiel hierzu findet sich in Listing 5.30.

Verzweigung bei aufzählbaren Datentypen

Aufzählbare Datentypen sind alle ganzzahligen Zahlen-Typen sowie selbst definierte Aufzählungstypen. Listing 5.30 zeigt, wie es grundsätzlich möglich ist derartiges mit `if` Anweisungen zu realisieren. Bei mehreren Bedingungen ist es der Übersichtlichkeit halber sinnvoller mit den Schlüsselwörtern `switch` und `case` zu arbeiten. Listing 5.31 zeigt ein Beispiel.

```

1 // Bei mehreren Befehlen mit Klammern
2 if (var == 5) {
3     // Nur wenn var fünf ist
4     Befehl1();
5     Befehl2();
6 }
7
8 // Bei einem Befehl sind Klammern nicht notwendig - dieses
9 // Beispiel zeigt aber, dass dadurch sehr leicht Fehler
10 // entstehen, da nachstehende Befehle eventuell so aussehen als
11 // gehörten sie zu dem if-Block
12 if (var == 5)
13     Befehl1();
14     Befehl2(); // Dieser Befehl wird immer ausgeführt, also unab-
15     // hängig von var
16
17 if (var == 5) {
18     // Nur wenn var fünf ist
19     Befehl1();
20     Befehl2();
21 } else {
22     // Nur wenn var NICHT fünf ist
23     Befehl3();
24     Befehl4();
25 }
```

Listing 5.29: Verzweigung mit if

```

1 if (wert == 1) {
2 } else if (wert4 == 2) {
3 } else if (wert2 == wert4) {
4 } else {
5 }
```

Listing 5.30: if-basierte Unterscheidung bei aufzählbaren Typen

Beachten Sie, dass die Programmausführung nicht nur einen einzigen zutreffenden `case` ausführt, sondern fortsetzt, wenn sie nicht explizit durch ein `break` abgebrochen wird. `break` verlässt den `switch-case-Block` und setzt die Programmausführung an der nächsten Anweisung nach dem Block fort.

```

1  switch (wert) {
2      case 1:
3          Befehl1(); // Befehle für wert == 1
4          break;
5
6      case 2:
7          Befehl2a(); // Befehle für wert == 2
8          Befehl2b();
9          // Da hier kein break steht wird auch noch Befehl3 ausgefü-
10         hrt - Dieser Effekt wird als Fall-Through bezeichnet und
11         sollte der Übersichtlichkeit halber immer explizit mit
12         einem Kommentar hervorgehoben werden
13
14     case 3:
15         Befehl3(); // Befehle für wert == 3 und wert == 2
16         break;
17
18     default:
19         Befehl4(); // Befehle für alle anderen Fälle
20 }
```

Listing 5.31: switch-case-Konstrukte

for-Schleife

Diese mit dem Schlüsselwort `for` eingeleitete Kontrollstruktur kommt immer dann zum Einsatz, wenn die Anzahl an Wiederholungen im Voraus bekannt ist. Obwohl eine `for`-Schleife auch für eine vor Schleifenantritt unbekannte Anzahl an Iterationen verwendet werden kann, ist es eine für die Wartung hilfreiche Konvention, für solche Schleifen das `while` Konstrukt zu verwenden.

Im Initialisierungssteil wird die Zählervariable (die deklariert werden muss) auf einen sinnvollen Wert gesetzt. Es folgt die Abbruchbedingung und schließlich die Inkrementierungsvorschrift. Die im Körper definierte Funktionalität wird bei jeder Iteration ausgeführt. Listing 5.32 zeigt ein einfaches Beispiel. Auch hier kann mit `break` die Schleife vorzeitig verlassen werden. Soll die Schleife nicht vollständig verlassen, sondern nur die aktuelle Iteration abgebrochen werden, kann mit `continue` erzwungen werden an den Anfang der nächsten Iteration zu springen. Sowohl `break` als auch `continue` sollten aus stilistischen Gründen vermieden werden.

```

1 int i;
2 // (Initialisierung; Abbruchbedingung;
3 // Inkrementierungsvorschrift)
4 for (i=0; i<10; i++) {
5     if (i == 7) {
6         // Beim 7. Durchlauf wird keine Meldung ausgegeben
7         continue;
8     }
9     // Das hier wird 9 mal ausgeführt
10    printString("Das ist Iteration ");
11    printNumber(i);
}
```

Listing 5.32: for-Schleife

Im Gegensatz zu anderen Sprachen, wie z. B. Java, ist die Deklaration von Variablen im Schleifenkopf (z. B. `for (int i=0; i<10; i++)`) nicht in allen C-Standards zulässig und sollte daher nicht verwendet werden.

while-Schleife

Die while-Schleife wird mit dem Schlüsselwort `while` eingeleitet. Diese Schleife wird hauptsächlich verwendet, wenn die Anzahl an Iterationen im Voraus unbekannt ist. Der Körper wird solange ausgeführt, wie die definierte Bedingung erfüllt wird. Ist die Abbruchbedingung bereits zu Beginn der while-Anweisung erfüllt, wird der Schleifenkörper nicht ausgeführt. Listing 5.33 zeigt ein Beispiel.

```

1 int i = 0;
2 while (generateRandomNumber() > 42) {
3     i++;
4 }
```

Listing 5.33: while-Schleife

do-while-Schleife

Die do-while-Schleife, welche mit dem Schlüsselwort `do` eingeleitet wird, verhält sich ähnlich zur while-Schleife, jedoch wird der Schleifenkörper auf jeden Fall mindestens ein Mal durchgeführt, die Abbruchbedingung wird erst nach dem ersten Durchlauf zum ersten Mal überprüft. Listing 5.34 zeigt ein Beispiel.

```

1 char i = 0;
2 do {
3     i++; // Wird mindestens ein Mal ausgeführt.
4 } while (generateRandomNumber() > 42);

```

Listing 5.34: do-while-Schleife

LERNERFOLGSFRAGEN

- Was sind die Unterschiede zwischen `if` und `switch` Anweisungen?
- Was versteht man unter einem Fall-Through?
- In manchen Fällen können `switch` Blöcke durch den Compiler in effizienteren Code umgewandelt werden als äquivalente `if` Blöcke. Haben Sie dafür eine Erklärung?
- Welche Schleifentypen gibt es in C? Sind alle diese Typen gleich ausdrucksstark?

5.1.16 Registerzugriff, Bitshifting & Bitmasken

In der hardwarenahen Programmierung ist es häufig notwendig, ein Hardwareregister direkt zu manipulieren. Dies ist z. B. der Fall wenn einzelne Funktionalitäten über Register konfiguriert werden. Da im Regelfall keine Operation zur Verfügung steht, einzelne Bits eines Registers zu verändern, nutzt man an dieser Stelle Bitmasken.

Eine Bitmaske ist nichts anderes als ein Zahlenwert. Es ist aber meist besser lesbar, diesen Wert nicht dezimal, sondern binär zu notieren. Teilweise ist auch eine hexadezimale oder oktale Darstellung hilfreich. Beispiel 5.35 illustriert dies.

Bei der Angabe von Bitmasken ist es üblich führende Nullen anzugeben, um die Größe des Ergebnisses anzudeuten. Masken auf diese Art zu spezifizieren ist allerdings unüblich, da es fehleranfällig und schlecht wart- und portierbar ist. Daher wird oft der Shiftoperator wie in Listing 5.36 verwendet, um die Maske durch den Compiler ausrechnen zu lassen - dies geschieht zur Compilezeit, verlangsamt also die Ausführungsgeschwindigkeit des

```

1 // Bit 3 (beginnend bei 0) in Register reg setzen
2 reg |= 0b00001000;
3 // Bit 2 (beginnend bei 0) in Register reg nul len
4 reg &= 0b11111011;
5 // Bit 1 (beginnend bei 0) in Register reg invertieren
6 reg ^= 0b00000010;

```

Listing 5.35: Registerzugriffe

```

1 mask1 = (1 << 5); // 0b00100000
2
3 // Die so erzeugten Masken lassen sich verknüpfen um mehrere
   Bits gleichzeitig zu manipulieren
4 mask2 = (1 << 5) | (1 << 3); // 0b00101000
5
6 // Setzt Bit 5 und 3 falls diese in 'value' gesetzt sind
7 var2 |= value & mask2;

```

Listing 5.36: Bitmasken

Programms nicht.

Die im vorherigen Schritt erzeugten Masken werden anschließend mit binären Operatoren auf das jeweilige Register angewandt. Dabei können im Grundsatz die beiden folgenden Intentionen unterschieden werden:

Setzen eines Bits Sollen ein oder mehrere Bits in einem Register gesetzt werden, so wird die zugehörige Bitmaske mit dem Register binär Oder-Verknüpft.

LVALUE |= BITMASKE;

Rücksetzen eines Bits Sollen ein oder mehrere Bits in einem Register genullt (zurückgesetzt) werden, so wird die zugehörige Bitmaske invertiert und mit dem Register binär und-verknüpft.

LVALUE &= ~BITMASKE;

Listing 5.37 zeigt verschiedene Varianten von Registermanipulationen.

```

1 // Es existieren mehrere Möglichkeiten Bit 1, 3 und 5 in REGB
2 // zu setzen. Viele davon sind aufwändig und schlecht lesbar.
3 REGB |= 0b00101010;
4 REGB |= (10 << 2) + 2;
5 REGB |= (1 << 5) | (1 << 3) | (1 << 1);
6 // Die zweite Variante sollte in jedem Fall
7 // vermieden werden, da weder wart- noch portierbar
8 // Es sollte die dritte Variante bevorzugt werden
9
10 // Analog: Rücksetzen von Bits
11 REGC &= 0b11010101;
12 REGC &= ~(10 << 2) - 2;
13 REGC &= ~((1 << 5) | (1 << 3) | (1 << 0));
14 // Auch hier sollte die dritte Variante bevorzugt werden
15
16 // Nutzlos bzw. gefährlich ist dagegen:
17 REGD |= (0<<3); // Shiftet und verodert eine 0, also keine Ä
18 // nderung in REGD
19 REGD &= (0<<2); // Setzt das ganze (!) Register auf 0
20
21 // In diesem Beispiel wird geprüft, ob das dritte Bit in PINA
22 // gesetzt ist
23 while (PIN & (1 << 3)) {
24     // ...
25 }
```

Listing 5.37: Bitmasken und Register

LERNERFOLGSFRAGEN

- Was versteht man unter einer Bitmaske?
- Wieso muss beim Setzen von Nullen die Bitmaske zunächst invertiert werden, beim Setzen vom Einsen jedoch nicht?
- Wieso ist es sinnvoll Bitmasken nicht explizit (binär, dezimal, ...) hinzuschreiben, sondern sie mithilfe des Shift- und bitweisen Oder Operators zusammenzusetzen?
- Um eine 1 in einer Bitmaske zu setzen, etwa an Stelle 3, genügt die Veroderung mit dem Ausdruck `(1 << 3)`. Wieso ist es beim Setzen einer 0 nicht analog mit dem Ausdruck `(0 << 3)` möglich?

5.1.17 Inline Assembler

In manchen Fällen kann es erforderlich sein, Assemblercode in eine C Datei einzubetten. Dazu werden die Schlüsselwörter `__asm__ volatile` verwendet. Das Listing 5.38 veranschaulicht eine solche Einbettung. Dabei sollte jeder Befehl in einer eigenen Zeile stehen, und jeweils mit Anführungszeichen versehen werden.

```
1 __asm__ volatile (
2     "ldi r16, 0xFF \n\t"
3     "out DDRB, r16 \n\t"
4     "JMP 0x0000 \n\t"
5 );
```

Listing 5.38: Inline Assembler

LERNERFOLGSFRAGEN

- Wie lässt sich Assembler in C-Code einbetten?
- Wieso ist es sinnvoll dies so sparsam wie möglich einzusetzen?

5.1.18 Zufallszahlen

Für Zufallszahlen steht in C die Standardbibliothek stdlib.h zur Verfügung. Sie enthält die Methoden `void srand (unsigned int seed)` und `int rand (void)`:

Mittels `srand(seed)` wird eine Kette von Pseudozufallszahlen initialisiert, ausgehend vom sogenannten seed-Wert. Daraufhin können beliebig oft Zahlen mit `rand()` abgerufen werden. Die Funktion `rand()` liefert pseudo-zufällige Werte zwischen 0 und `RAND_MAX` $\geq 2^{16} - 1$ zurück.

Wenn Sie Ihre Kette von Zufallszahlen mit demselben seed-Wert beginnen, erhalten Sie die gleiche Kette von Zufallszahlen. Das bedeutet, dass bei einem festkodierten seed-Wert ein Mikrocontroller nach jedem Reset exakt dieselben Ergebnisse liefert, obwohl vermeintliche Zufallszahlen genutzt wurden. Sollte dies zu einem Problem führen, gibt es Möglichkeiten den seed-Wert in Abhängigkeit der Umwelt zu wählen. So kann der seed-Wert z. B. aus dem Wert eines Analog/Digital-Wandler Pins generiert werden, insofern dieser Pin mit keinem Bezugspotential verbunden ist. Durch den offenliegenden Anschluss des Wandlers nimmt dieser elektromagnetische Strahlung aus der Umgebung auf, die einem natürlichen Rauschen unterliegt und hinreichend zufällig ist. Für viele Anwendungen reicht die Initialisierung mit einem festen Wert vollkommen. Listing 5.39 zeigt die Verwendung.

```

1 #include <stdlib.h>
2 int main(void) {
3     srand(4711);
4     int pseudorandom = rand();
5     return 0;
6 }
```

Listing 5.39: Zufallszahlen

LERNERFOLGSFRAGEN

- Was versteht man in Zusammenhang mit Zufallszahlen unter einem Seed?
- Wie können Sie eine *Gleitkomma-Zufallszahl* zwischen 0 und 2 erzeugen?

5.2 Strukturverbesserungen

Große Softwareprojekte erfordern ein Mindestmaß an Übersichtlichkeit. Gerade bei der Arbeit in Teams wird es zunehmend wichtig seinen Code zu strukturieren und für gute

Lesbarkeit zu sorgen. Das fördert die Zusammenarbeit der Teammitglieder untereinander und hilft jedem Programmierer selbst. Codestrukturierung verbessert nicht nur die Übersicht und damit Lesbarkeit des Codes, sondern hilft bei der Erkennung von Fehlern und deren Behebung. Durch bestimmte Konzepte, die im Folgenden vorgestellt werden, können bekannte Fehlerquellen reduziert oder vermieden werden. Zu guter Letzt ist Codestrukturierung ein wesentliches Merkmal für die Wartbarkeit und Erweiterbarkeit von Programmen und somit im Rahmen der hardwarenahen Programmierung besonders wertvoll.

Viele der vorgestellten Methoden sind keine festen Regeln, die von vornherein anwendbar sind. Ziel ist es, die Möglichkeiten zur besseren Strukturierung des geschriebenen Codes aufzuzeigen. Da während der Implementierung häufig unstrukturierte Zwischenstufen entstehen, sind die nachfolgenden Konzepte zur stetig wiederholten Anwendung gedacht. Nehmen Sie sich Zeit Ihr Programm immer wieder selbst zu lesen, zu hinterfragen und mit folgenden Konstrukten immer wieder schrittweise zu strukturieren.

Dieses Kapitel orientiert sich an "*Fowler, Martin : 'Refactoring - Improving the Design of Existing Code', Addison-Wesley 1999*". Dieses Buch kann in der Informatikbibliothek gefunden werden.

5.2.1 Kommentare

Oft ist nach einiger Zeit nicht einmal für den Programmierer verständlich, wie der von ihm geschriebene Quelltext funktioniert. Für Außenstehende ist es noch viel schwieriger fremden Quelltext zu lesen und die Abläufe zu erkennen. In diesem Sinne sind Kommentare in einem lesbaren Quellcode unverzichtbar.

- Folgende Dinge sollten durch Kurzkommentare erläutert werden:
 - Konstanten
 - Definitionen
 - Funktionsköpfe
 - komplexe Datenstrukturen
 - komplizierte Programmabläufe
- Für Funktionen sollte ein mehrzeiliger Kommentar verwendet werden, der zumindest auf die Parameter und den Rückgabewert eingeht, siehe Listing 5.40. In diesem Beispiel finden Sie einige spezielle Tags. Diese Tags werden für das Dokumentationsstool Doxygen, siehe Kapitel 7, benötigt.
- Innerhalb von Funktionen sind Kommentare zulässig.
- Zu vermeiden sind überflüssige Kommentare. Als Faustregel gilt: Code sollte idealerweise ohne Kommentar verständlich sein. Sonst ist der Quelltext dahingehend zu verbessern, dass er verständlich wird. Alles was anschließend unverständlich sein könnte ist mit Kommentaren zu versehen.

- Kommentare werden nicht ausschließlich innerhalb des Codes gelesen. Es gibt eine Reihe von Tools, die aus Kommentaren ein externes Dokumentationsdokument erzeugen. Ein bekanntes Tool wird in Kapitel 7 „Dokumentation mit Doxygen“, vorgestellt.

```

1  /*!
2  * Die Aufgaben der Funktion werden in
3  * einem mehrzeiligen Kommentar erläutert.
4  *
5  * \param var1 Eine Eingangsvariable
6  * \param var2 Eingangsvariable Nummer 2
7  * \return Der Rückgabewert
8  */
9 int tolleFunktion(char var1, void* var2) { // ...

```

Listing 5.40: Dokumentation von Funktionsbestandteilen

5.2.2 Umgang mit Funktionen und Kontrollstrukturen

Die sequenzielle Programmierung erlaubt das Aneinanderreihen langer Befehlsketten. Der nachfolgende Abschnitt soll Ihnen Sensibilität dafür vermitteln, wie große und schwer verständliche Abläufe in Teile zerlegt und geordnet werden sollten.

Extraktion von Funktionen

Ein wichtiges Konzept ist die Auslagerung von Funktionen. Sinngemäß zusammenhängende Codestücke oder solche, die oft wiederkehren, sollten herausgenommen und in eine eigene Funktion ausgelagert werden. Dies verleiht den entsprechenden Codestücken nicht nur einen verständlichen Bezeichner, sondern eliminiert Redundanzen. In Listing 5.41 ist eine unverständliche Funktion gezeigt. Diese ist in Listing 5.42 aufgebrochen; obwohl das zweite Listing insgesamt mehr Zeilen hat, sind die einzelnen Funktionen kürzer und einfacher verständlich. Weitere Verbesserungen sind möglich.

Doppelten Code vermeiden

Doppelter Code bringt diverse Probleme mit sich. Eine Änderung muss an allen redundanten Stellen nachvollzogen werden oder zieht Fehler nach sich. Doppelten Code zu vermeiden geht häufig mit Datenkapselung und Funktionsextraktion einher.

Logische Zusammenhänge beachten

Es gibt häufig mehrere äquivalente Darstellungsweisen eines Problems. Dabei können einige Darstellungen für Menschen komplizierter zu verstehen sein als andere. Zu viele Negationen führen oft zu Fehlern, daher sollten Redundanzen, wie in Listing 5.43 dargestellt, vermieden werden.

```

1 int value;
2 int *pointer;
3 char highBits;
4
5 int main(void){
6     int i;
7     for(i=1; i<10; i++){
8         if(!highBits){
9             value = *pointer & 0b00001111;
10        } else {
11            value = *pointer & 0b11110000;
12        }
13        highBits++;
14        // ...
15        if(highBits){
16            pointer = pointer+1;
17            highBits = 0;
18        } else {
19            highBits = 1;
20        }
21    }
22}
```

Listing 5.41: Überfüllte Funktion

5.2.3 Umgang mit Daten

Es gibt eine Unmenge an Möglichkeiten mit Daten umzugehen. Hier wird hauptsächlich auf die Arbeit mit Variablen eingegangen. Variablen arbeiten unter Umständen mit einer Vielzahl von Werten. Ihre Werte können sich in einer Vielzahl von Interpretationen und Typen unterscheiden. Nicht zuletzt werden Variablen häufig an vielen Stellen des Programms verändert. Daher sollte die Verwendung von Variablen strukturiert werden.

Sichtbarkeit von Variablen

- Aufgrund der begrenzten Ressourcen eines Mikrocontrollers sind globale Variablen möglichst zu vermeiden, da diese dauerhaft Speicherplatz belegen.
- Globale Variablen sind eine typische Fehlerquelle: Jede Funktion kann sie lesen und (sofern nicht `const`) beschreiben. Das Beschreiben einer globalen Variablen durch eine Funktion kann ein Fehlverhalten einer anderen Funktion bewirken. Ein solches Verhalten wird als *Seiteneffekt* bezeichnet, Funktionen mit Seiteneffekten als *seiteneffektbehaftet*. Das Reduzieren globaler Variablen beugt Seiteneffekten vor.

```

1 int value;
2 int *pointer;
3 char highBits;
4
5 void increasePointer(void){
6     if (highBits) {
7         pointer = pointer+1;
8         highBits = 0;
9     } else {
10        highBits = 1;
11    }
12}
13 int readValue(void){
14     if(!highBits){
15         return *pointer & 0b00001111;
16     } else {
17         return *pointer & 0b11110000;
18     }
19}
20 int main(void){
21     int i;
22     for(i=1; i<10; i++){
23         value = readValue();
24         // ...
25         increasePointer();
26     }
27}
```

Listing 5.42: Extrahieren in neue Funktion

- Allgemein sollte die Sichtbarkeit so lokal wie möglich gehalten werden, dadurch werden Fehler bereits zur Entwurfs- bzw. Compilezeit vermieden.
Ein wichtiges Instrument in diesem Zusammenhang ist die Verwendung von „Get-“ und „Setmethoden“, die den Zugriff auf gekapselte lokale Variablen erlauben.

Nützliche Variablen, überflüssige Variablen & Wahl richtiger Bezeichner

In den vorhergehenden Abschnitten gab es viele Beispiele an denen zu erkennen war, dass Variablen in der Programmierung in C unumgänglich sind. Um den erzeugten Code lesbar zu halten, ist es wichtig auf die Benennung der Bezeichner zu achten.

```

1 if (!safe) {
2     noalert = 0;
3 }
```

Listing 5.43: Doppelte Negation

↓

```

1 if (danger) {
2     alert = 1;
3 }
```

Listing 5.44: Vermeidung doppelter Negation

Gute Bezeichner

Gute Bezeichner sind kurz, aber dennoch aussagekräftig. Vermeiden Sie kryptische Bezeichner; z. B. für “last Analog/Digital Conversion“:

- schlecht: `1stadcnv`
- besser: `lastADConversion` oder `lastADC`

Überflüssige Variablen

Temporäre Variablen sind eventuell überflüssig, wie in Listing 5.45 illustriert.

```

1 float circumference(float radius){
2     float temp = 2.0*radius*3.141;
3     return temp;
4 }
```

Listing 5.45: Überflüssige Variable

↓

```

1 float circumference(float radius){
2     return 2.0*radius*3.141;
3 }
```

Listing 5.46: Vermeidung überflüssiger Variablen

Das hier gezeigte Beispiel ist sehr einfach. In komplexeren Zusammenhängen kann es wiederum durchaus die Lesbarkeit bzw. Verständlichkeit des Codes erhöhen, falls temporäre Variablen vorhanden sind. Dies ist ein Kompromiss, zu dem es keine pauschale Lösung gibt.

Wiederverwendung Mehrfachverwendung ist eine Fehlerquelle: Für verschiedene temporäre Werte sollten verschiedene temporäre Variablen verwendet werden. Bereits behandelte Paradigmen, wie das der eindeutigen Bezeichner, dürfen nicht vergessen werden:

```

1 void calculateCube(int length){
2     int temp = length*length;
3     print(temp);
4     temp = temp*length;
5     print(temp);
6 }
```

Listing 5.47: Mehrfachverwendung temporärer Variable

↓

```

1 void calculateCube(int length){
2     int area = length*length;
3     print(area);
4     int volume = length*length*length;
5     print(volume);
6 }
```

Listing 5.48: Trennung temporärer Variablen

Lesbarkeit durch zusätzliche Variablen Das Einführen zusätzlicher Variablen ist nicht grundsätzlich schlecht. Im Zweifelsfall ist abzuwägen, was die Lesbarkeit erhöht. Das folgende Beispiel beschäftigt sich mit der Einführung beschreibender Variablen:

```

1 void status(void){
2     if ((getRunningLevel() == 1) && (((readSensorV()-10) >
3         voltageThreshold) || (readSensorA() > currentThreshold))) {
4         alert();
5     }
}

```

Listing 5.49: Großes if-Konstrukt

↓

```

1 void status(void){
2     int const voltage = readSensorV()-10;
3     bool const voltageTooHigh = voltage > voltageThreshold;
4     int const current = readSensorA();
5     bool const currentTooHigh = current > currentThreshold;
6     int const runlevel = getRunninglevel();
7     bool const danger = voltageTooHigh || currentTooHigh;
8
9     if (runlevel == 1 && danger) {
10         alert();
11     }
12 }

```

Listing 5.50: Einführung beschreibender Variable(n)

Es ist allerdings anzumerken, dass in Listing 5.50 alle Funktionen ausgeführt werden, egal wie der Wert von `runlevel` ist. In Listing 5.49 jedoch werden die Funktionen *lazy* ausgeführt, also nur, falls sie am Ergebnis der Abfrage noch etwas ändern können. Bei zeitintensiven oder seiteneffektbehafteten Funktionen kann dies einen entscheidenden Unterschied darstellen.

Konstanten Die Bedeutung hartkodierter Werte ist nicht immer verständlich, stattdessen sollten besser aussagekräftige Konstanten oder `defines` verwendet werden. Vgl. Listing 5.51

Datenstrukturen Auflistungen, deren Bedeutung nur dem Programmierer selbst verständlich sind, sollten in kommentierte `structs` ausgelagert werden. Für die Bezeichnung und das Festhalten von Zuständen sollten unter Verwendung von `enums` selbsterklärende Bezeichner gewählt werden.

Verwendung von `typedefs` `typedefs` erscheinen auf den ersten Blick unwesentlich, da sie lediglich neue Namen für existierende Namen festlegen. Tatsächlich ist das `typedef` Schlüsselwort aber unerlässlich für verständlichen, portierbaren und wartbaren Code. Aus einem Typnamen sollte hervorgehen, welchen logischen Typ eine Variable hat: Mit `typedefs` können deskriptive Typnamen eingeführt werden.

```
1 float freeFallDistance(int t){  
2     return 0.5 * 9.81 * t * t;  
3 }
```

Listing 5.51: Mysteriöser Wert



```
1 #define EARTH_GRAVITY_ACCELERATION 9.81  
2 float freeFallDistance(int t){  
3     return 0.5 * EARTH_GRAVITY_ACCELERATION * t * t;  
4 }
```

Listing 5.52: Konstantenverwendung

Ein weiterer Vorteil ist, dass eine Änderung des technischen Typs mehrerer Variablen des gleichen logischen Typs bei Verwendung eines `typedefs` eine einzige Änderung erfordert. Listing 5.56 zeigt die modifizierte Version des in Listing 5.55 gezeigten Codes. Im Praktikum Systemprogrammierung wird für alle selbstdefinierten Typen eine Großschriftkonvention verwendet, um Typnamen von Variablennamen zu unterscheiden. Eine gebräuchliche Alternative besteht darin an selbst definierte, skalare Typen den Suffix `_t` anzuhängen.

```

1 unsigned int marvin[2];
2 marvin[0]=2;
3 marvin[1]=245915;
```

Listing 5.53: Seltsames Array

↓

```

1 // Statt unbekannter Zahlenkodierung, verwende enum
2 enum Studiengaenge {
3     DIPLOM, BACHELOR, MASTER
4 };
5 enum StudentIndex {
6     STUDIENGANG = 0,
7     MATRNUMMER = 1
8 };
9
10 unsigned int marvin[2];
11 marvin[STUDIENGANG]=MASTER;
12 marvin[MATRNUMMER]=245915;
```

Listing 5.54: Array mit Bezeichnern

↓

```

1 enum Studiengaenge {
2     DIPLOM, BACHELOR, MASTER
3 };
4 // Statt Array konstruiere aussagekräftiges struct
5 struct Student {
6     enum Studiengaenge studienGang;
7     unsigned int matrNummer;
8 };
9
10 struct Student marvin;
11 marvin.studienGang = MASTER;
12 marvin.matrNummer = 245915;
```

Listing 5.55: Structverwendung

```

1 // Definiere "Studiengang" als anonymes enum
2 typedef enum {
3     DIPLOM, BACHELOR, MASTER // ...
4 } Studiengang;
5 // Legt extra Typnamen für Matrikelnummer und Jahr an
6 typedef unsigned short Matnum;
7 typedef unsigned short Jahr;
8 typedef struct {
9     Studiengang studienGang;
10    Matnum matrNummer;
11    Jahr immatrikulation;
12    Jahr geboren;
13 } Student;
14
15 // Definition ohne Initialisierung
16 Student marvin;
17 marvin.studienGang = MASTER;
18 marvin.matrNummer = 123456;
19 marvin.immatrikulation = 2009;
20 marvin.geboren = 1989;
21
22 // Initialisierung eines konstanten structs
23 Student const marvin = {
24     .studienGang = MASTER,
25     .matrNummer = 123456,
26     .immatrikulation = 2009,
27     .geboren = 1989
28 };

```

Listing 5.56: Structverwendung

Lesbares Bitshifting

In Abschnitt 5.37 wurden bereits Registerzugriffe und der dazugehörige Umgang mit Bitmasken erläutert. Dort wurden viele Möglichkeiten aufgezählt, um Bits an die richtigen Stellen zu verschieben, falls nötig zu negieren und mit weiteren Operatoren auf Register anzuwenden. Zur Verbesserung der Lesbarkeit sollten solche Konstrukte überdacht werden.

Die Les- und Portierbarkeit dieser Ausdrücke kann verbessert werden, indem die einzelnen Bits nicht um ihre Stelligkeit verschoben werden, sondern der Namen des entsprechenden Bits verwendet wird. Die Bibliothek `avr/io.h` hält zu diesem Zweck Definitionen (`#define`) für alle Register und deren Bits, entsprechend der Bezeichnungen im Datenblatt, bereit. Dies zeigt das Listing 5.57.

```

1 #include <avr/io.h>
2 PINA = ( ADCSRA | (1 << 6) ) & ~(1 << 2);

```

Listing 5.57: Schlecht lesbares Bitshifting



```

1 #include <avr/io.h>
2 ADCSRA |= (1 << ADEN); // ADC Enable Bit
3 ADCSRA &= ~(1 << ADIE); // ADC Interrupt Enable Bit

```

Listing 5.58: Bereits lesbar



```

1 // Sollte global verfügbar gemacht werden
2 // "Set Bit"
3 #define sbi(bitset,bit) bitset |= (1 << (bit))
4 // "Clear Bit"
5 #define cbi(bitset,bit) bitset &= ~(1 << (bit))
6
7 //-----
8 // Dann kann verwendet werden:
9 sbi(ADCSRA,ADEN); // ADC Enable Bit
10 cbi(ADCSRA,ADIE); // ADC Interrupt Enable Bit

```

Listing 5.59: Unter Verwendung von Makros

Manchmal verleitet Bequemlichkeit dazu diese disziplinierte, schrittweise Abarbeitung zu vernachlässigen und kürzere Schreibweisen zu verwenden. Dies ist eine häufige Fehlerquelle, da Denkfehler in der booleschen Logik auftreten können und selbst Tippfehler drastische Auswirkungen haben. Es ist daher guter Programmierstil, sich diese regelmäßig benutzten Operationen in ein eigenes Makro auszulagern. Damit verkürzt man die Schreibweise und ermutigt sich selbst zur empfohlenen Disziplin, wie Listing 5.58 zeigt.

5.3 Quelltextkonventionen

Neben den strukturellen Merkmalen helfen Quelltextkonventionen den Code verständlich zu halten. Dies erlaubt effizientere Wartung, Korrektur und Modifikationen desselben. Speziell im Rahmen von Praktika während des Studiums empfiehlt es sich gemäß Quelltextkonventionen zu arbeiten, um so eine bessere Betreuung zu ermöglichen.

Die hier vorgestellten Richtlinien sind nur eine Möglichkeit von vielen und sollen ein Beispiel für Quelltextkonventionen darstellen. Da diese Konventionen dem persönlichen Geschmack des Programmierers unterliegen, gibt es so viele verschiedene Konventionen wie Programmierer. Für ein Projekt sollten sich jedoch alle an dem Projekt beteiligten auf eine Konvention einigen und diese beibehalten.

5.3.1 Dateien

- Implementierung und Interface sollten in separate Code- und Header-Dateien getrennt werden.
- Ein Zeilenumbruch nach 120 Zeichen macht den Code besser lesbar, da horizontales Scrollen vermieden wird.
- Wenn unabhängige Module verwendet werden, empfiehlt es sich Konstanten, Strukturen, etc. dem jeweiligen Modul zuzuordnen. Eine Ausnahme bilden projektweit globale Defines und Typedefs.

5.3.2 Kommentare

- Header-Dateien sollten mit einer kurzer Beschreibung versehen werden, welche zumindest die folgenden Daten enthält:
 - Autor
 - Datum
 - Version
 - falls bekannt, Fehler

Ein Beispiel ist in Listing 5.60 gegeben.

```
1  /*! \file
2   * \brief Kurzbeschreibung dieser Datei
3   *
4   * Dies ist eine etwas längere Beschreibung der Datei.
5   *
6   * \author      Max Musterstudent
7   * \date       2008
8   * \version     3.2
9   * \bug        stürzt ab, wenn man zweimal auf ESC drückt
10  */
```

Listing 5.60: Doxygen-Kommentare

5.3.3 Bezeichner

- Wir unterscheiden 2 Notationen:
 1. Variablen und Funktionen werden anhand der *camelCase* Notation benannt. Grundsätzlich wird jedes Wort komplett klein geschrieben. Wird ein Bezeichner aus mehreren einzelnen Wörtern zusammengesetzt, wird mit einem Kleinbuchstaben begonnen und der erste Buchstabe jedes nachfolgenden Worts groß geschrieben.
Beispiel: device interrupt handler → deviceInterruptHandler
 2. Um Konstanten optisch hervorzuheben ist es sinnvoll, ausschließlich Großbuchstaben zu verwenden und Begriffe mit einem Unterstrich zu trennen. Gleichtes gilt für Aufzählungstypen.
Beispiel: device interrupt id → DEVICE_INTERRUPT_ID
- Zeigerdefinitionen sollten einheitlich mit dem *-Operator am Variablennamen und nicht am Variablentyp (`char *ptr` anstelle von `char* ptr`) durchgeführt werden.
- Neue Datentypen sollten inklusive des Schlüsselwortes `struct` deklariert werden. Bei der Definition sollte das anführende Wort groß geschrieben werden.
Beispiel: MyDeviceStruct, hier lässt man jedoch das Wort „Struct“ typischerweise weg.

5.3.4 Definitionen und Konstanten

- Feststehende Werte sollten mit der `#define`-Direktive definiert werden.
- Definition von Zeichenketten mit `PROGMEM char const` anstelle von `char const`, damit diese nicht im SRAM, sondern im Flashspeicher abgelegt werden.
- Die Präprozessordirektive `#ifndef ... #endif` verhindert ein mehrfaches Einbinden von Quelltext. Das Listing 5.61 zeigt die Syntax und Verwendung der Direktive.

```

1  /* !
2   * Ausführlicher Kommentar
3   */
4  #ifndef _OS_SOMETHING_H
5  #define _OS_SOMETHING_H
6
7  // Der Eigentliche Dateiinhalt
8
9 #endif
10 // Dateiende

```

Listing 5.61: Definition von Konstanten und Symbolen über Präprozessordirektiven

5.3.5 Klammern

- Der Einsatz von Klammern erhöht die Lesbarkeit des Quelltextes.
- Schließende geschwungene Klammern stehen in einer eigenen Zeile. öffnende können in einer eigenen Zeile stehen, müssen es aber nicht³.

5.3.6 Anweisungen

Vermeiden Sie es, mehr als eine Anweisung pro Zeile zu haben. Beispiel:

```

1 unsigned char a, b;
2 a = 3;
3 b = 2 + (a *= 10);

```

Listing 5.62: Mehrere Anweisungen pro Zeile

Dies hat folgende Semantik:

- a und b werden deklariert
- a wird der Wert 3 zugewiesen
- a wird der Wert $a \cdot 10 = 30$ zugewiesen
- b wird der Wert $2 + 30 = 32$ zugewiesen

Die zweite Zuweisung an a ist nicht offensichtlich. Wenn Sie Fehler in Ihrem Programm suchen müssen, erschweren solche Konstrukte die Suche erheblich.

5.3.7 Casten von Variablenarten

Bedenken Sie bei Ihrer Implementierung, dass Ihr C-Code für einen 8 Bit RISC-Prozessor kompiliert wird, was zu anderem Verhalten führt, als sie es aus anderen Programmiersprachen gewohnt sind. Das betrifft insbesondere die Arbeit mit Variablen unterschiedlicher Größe. Zunächst wird folgendes Beispiel betrachtet:

```

1 unsigned char x = 255;
2
3 // Test A
4 lcd_line1();
5 unsigned char y = x + 1;
6 if (y > 200) {
7     lcd_writeString("groesser");
8 } else {
9     lcd_writeString("kleiner"); // <---
10 }

```

³Hier gibt es zwei in etwa gleich verbreitete Konventionen - für ein Projekt sollte sich auf eine geeinigt werden.

```

11 // Test B
12 lcd_line2();
13 if (x+1 > 200) {
14     lcd_writeString("groesser"); // <---
15 } else {
16     lcd_writeString("kleiner");
17 }

```

Listing 5.63: Integer Promotion

Unerwarteterweise zeigen nicht beide Displayzeilen das gleiche Ergebnis an. In Test A wird x um 1 inkrementiert und anschließend in y gespeichert. Diese Variable ist vom Typ `unsigned char`, weswegen das Resultat 256 zu 0 gecastet wird (`unsigned char` ist ein $\mathbb{Z}/256\mathbb{Z}$ Ring). Man kann sich das so vorstellen, dass der Überlauf in der Binärdarstellung `1|00000000` abgeschnitten wird, da kein neuntes Bit im `char` zur Verfügung steht. In Test B wird die Operation zunächst auf 16 Bit ausgeführt, obwohl einer der Werte aus einer 8-Bit Variable stammt, da die Konstante 1 implizit den Typ `int` hat. Es wird in diesem Fall nicht gecastet bevor die Bedingung ausgewertet wird, daher ist das Ergebnis nicht 0, sondern 256. Ähnliches Verhalten veranschaulicht dieses Beispiel:

```

1 unsigned char x = 0xFF;
2 // Es gilt
3 (unsigned char)~x == 0;
4 // Aber
5 ~x == 0xFF00; // != 0

```

Listing 5.64: Integer Promotion 2

Dieses Phänomen hat seine Ursache in der Definition der Sprache C und wird in ISO Standard 9899 als Integer Promotion bezeichnet. Dort heißt es „*If an int can represent all values of the original type, the value is converted to an int; otherwise, it is converted to an unsigned int. These are called the integer promotions.*“ Darauf aufbauend kann auch folgendes Verhalten auftreten:

```

1 unsigned char x = 0x0F; // == 0b00001111
2 x = ((~x) >> 1);
3 // Es gilt
4 x == 0xF8; // == 0b11111000
5 // Denn
6 ((~x) >> 1) == 0x7FF8;

```

Listing 5.65: Integer Promotion 3

Das x wird hier negiert, wodurch, entsprechend der Erklärung aus 5.64, der Wert 0xFFFF entsteht. Beim anschließenden Shiften nach rechts wird zwar eine 0 von links eingeschoben, jedoch werden beim nachfolgenden impliziten Cast zurück in einen `unsigned char` die

5 Einführung in die C Programmierung

ersten Bits abgeschnitten. Es wirkt so, als sei eine 1 statt einer 0 eingeschoben worden. Um Fehler zu vermeiden, müssen Sie diese Zusammenhänge berücksichtigen. Es ist daher eine weitere Konvention, dass alle Arithmetik- und Bitshift-Operationen explizit zum gewollten Typ gecastet werden müssen.

6 Hinweise zum Debuggen

Trotz sorgfältiger Modellierung und Planung eines Programms sind Fehler in der Implementierung nicht auszuschließen. In solchen Fällen ist die aktive Fehlersuche häufig unausweichlich. In diesem Dokument werden einige Verfahren vorgestellt, die es erleichtern können, Fehlverhalten in einem Programm aufzuspüren. Es wird von einer bereits festgestellten Fehlfunktion ausgegangen und gezeigt, wie man ihre Ursache findet.

Aufbau dieses Kapitels

Zunächst wird gezeigt, welche Problemfelder durch Debuggen abgedeckt werden können und welche Verfahren zum Einsatz kommen. Kapitel 6.2 führt in das Debuggen mit den Bedienelementen von Atmel Studio ein. In Kapitel 6.3 werden Probleme beim Debuggen diskutiert, die meistens mit der Optimierung des Compilers zusammenhängen. Schließlich werden in Kapitel 6.4 fehlerhafte Programme vorgestellt, die mit Hilfe der in den vorherigen Kapiteln diskutierten Verfahren korrigiert werden.

6.1 Was ist Debuggen?

Debuggen ist eine Methode, um Fehler in einem Programm aufzuspüren und zu beseitigen. Das Hauptziel des Debuggens ist das Fehlverhalten eines Programms zu der entsprechenden Stelle im Quelltext zurückzuverfolgen, wo die Ursache unmittelbar korrigiert werden kann. Die Abweichung im erwarteten Programmverhalten kann sich unter anderem in fehlerhaften Programmausgaben oder der Nichterreichbarkeit von Programmstellen zeigen. Eine Alternative zum Debuggen ist eine formale Korrektheitsanalyse des Quellcodes, die unter Umständen sehr aufwändig sein kann. Weitere Alternativen zum manuellen Debuggen sind symbolische Ausführung von Programmen und (randomisiertes) Black-Box Testing.

In diesen Unterlagen werden an verschiedenen Stellen Beispiele angegeben, um den Prozess des Debuggens zu veranschaulichen. Diese Beispiele sind zum Teil stark vereinfacht, um den Kern des jeweils dargestellten Verfahrens darzustellen. In der Praxis sind die untersuchten Programme meist deutlich komplexer.

6.1.1 Allgemeines Vorgehen beim Debuggen

Einen Fehler zu finden bedeutet, die fehlerhafte Stelle, die das falsche Verhalten auslöst, einzugrenzen. Dies ist oft ein iterativer Prozess, bei dem die fehlerhafte Programmstelle immer weiter eingegrenzt wird, bis der Fehler eindeutig lokalisiert wurde.

Im ersten Schritt des Debuggens muss überlegt werden, welche Codestellen generell für eine bestimmte Art des Fehlverhaltens in Frage kommen. Wenn eine frühere Version des Programmcodes fehlerfrei funktioniert hat, können z.B. neu hinzugefügte Funktionen Verursacher des Problems sein. Sind solche Kandidaten gefunden, können sie und die Bedingungen, unter denen sie aufgerufen werden, einzeln überprüft werden. Auf diese Weise kann ein konkreter Fehler mit Unterstützung einer Debuggersoftware wie dem Debugger aufgespürt und behoben werden.

6.1.2 Debugger

Im Allgemeinen beinhaltet Debuggen das Analysieren des aktuellen Programmzustands und des Speicherinhalts. In einem Szenario, in dem sich der Debugger sowie das zu debuggende Programm auf demselben Rechner befinden, kann der Debugger direkt auf die nötigen Daten zugreifen. Sollte man von seinem PC ein Programm auf externer Hardware (z.B. auf einem Mikrocontroller) debuggen wollen, so muss zwischen diesen beiden Geräten eine Schnittstelle existieren, die dem Debugger einen Zugriff auf die Informationen ermöglicht. In diesem Praktikum wird ein JTAGICE 3 Interface verwendet, um eine solche Verbindung herzustellen.

Atmel Studio enthält einen integrierten Debugger, der auch für den ATmega 644 Mikrocontroller verwendet werden kann. Der Debugger wird über die Debug-Symbolleiste, welche in Abbildung 6.1 dargestellt wird, und über die Menüelemente im Debug-Dropdownmenü gesteuert.



Abbildung 6.1: Die Debug-Symbolleiste von Atmel Studio

6.2 Debugging-Methoden

Zwei grundlegende Konzepte des Debuggens sind die Kontrollfluss- und die Speicherüberwachung. Das erste im Folgenden vorgestellte Konzept ist die Überwachung des Kontrollflusses, bei der der Ablauf des Programmes nachvollzogen wird. Dazu gehört auch die Betrachtung des aus dem C-Code kompilierten Assemblercodes, welcher die tatsächlich ausgeführten Instruktionen angibt. Danach wird die Überwachung des Speichers vorgestellt, aus der unter anderem auf die aktuelle Belegung von Variablen geschlossen werden kann. In der Praxis wird eine Kombination aus diesen Maßnahmen verwendet.

6.2.1 Überwachen der Programmausführung

Um den Ablauf des Programmes nachzuvollziehen, kann es schrittweise durchgegangen oder während der Ausführung angehalten werden. Dies kann zur Aufspürung von Unstimmigkeiten im Programmfluss verwendet werden. Im angehaltenen Zustand markiert

6 Hinweise zum Debuggen

der AVR Debugger die aktuelle Position der Programmausführung im Quellcode (Abbildung 6.2).

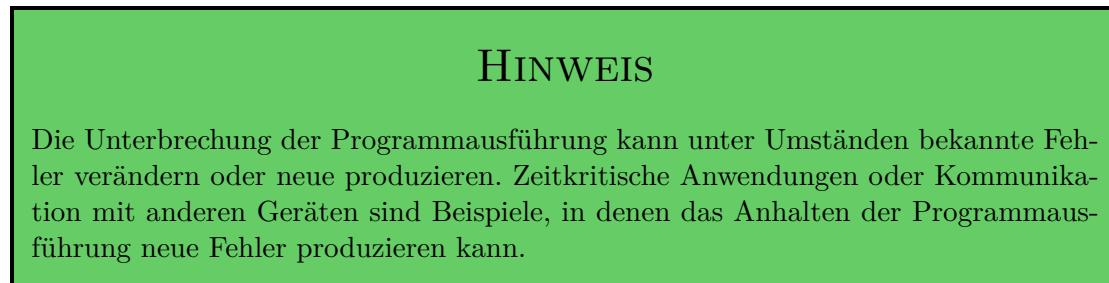


Abbildung 6.2: Markierung der aktuellen Position der Programmausführung

Pausieren des Programms

Während das Programm ausgeführt wird, kann es mit Hilfe des Pause-Knops (Abbildung 6.3) angehalten werden, um zu überprüfen, an welcher Stelle im Quelltext sich die Programmausführung zurzeit befindet.



Abbildung 6.3: Der Pause-Knopf

Verwendung Die Programmpause kann verwendet werden, wenn eine zu erwartende Ausgabe nicht erfolgt ist oder das Programm nicht auf Eingaben reagiert. Sollte der Programmablauf eine Endlosschleife enthalten, so wird das Programm wahrscheinlich genau dort pausiert werden. In diesem Fall können die Abbruchbedingungen der Schleife auf Richtigkeit und Erfüllbarkeit überprüft werden.

Darüber hinaus kann die Programmpause eingesetzt werden, um die weiter unten vorgestellten Techniken verwenden zu können.

Breakpoints

Ein Breakpoint markiert eine Zeile im Code, an der die Programmausführung automatisch angehalten werden soll. Der Debugger pausiert das Programm, sobald er an der Codezeile ankommt, die mit einem Breakpoint versehen wurde. Dabei werden die Anweisungen dieser Zeile noch nicht ausgeführt. Um einen Breakpoint zu setzen, muss in das weiße Feld links neben der entsprechenden Codezeile geklickt werden. Abbildung 6.4 zeigt den Vorgang exemplarisch. Ein weiterer Klick auf diesen Knopf entfernt den *zuvor in dieser Zeile* gesetzten Breakpoint. Es ist ebenfalls möglich mehrere Breakpoints an verschiedenen Stellen im Quelltext zu definieren, die unabhängig voneinander verarbeitet werden.

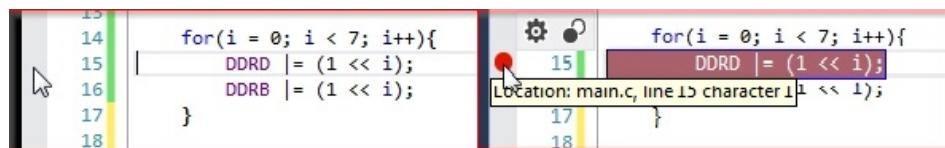
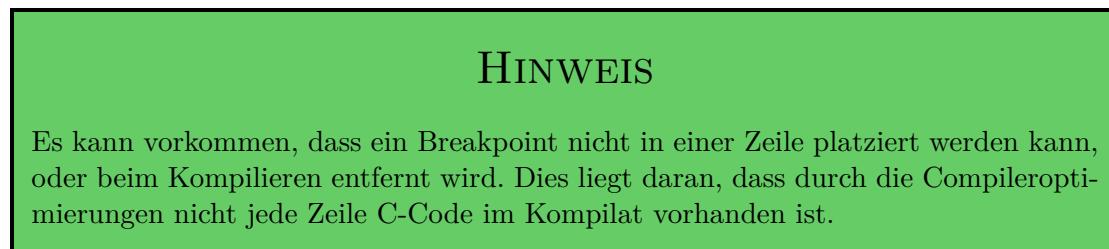


Abbildung 6.4: Setzen eines Breakpoints

Im Breakpoint Menü findet sich eine Übersicht aller momentan vorhandenen Breakpoints. Es befindet sich am unteren Bildschirmrand und ist alternativ über die Tastenkombination Alt + F9 aufrufbar. Hier können einzelne Breakpoints gelöscht und eingeschaltet werden.

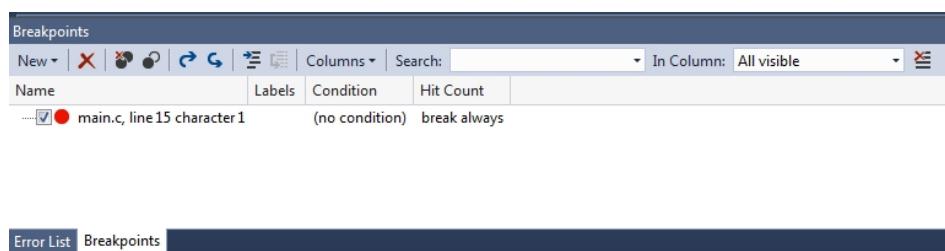


Abbildung 6.5: Breakpoint Menü von Atmel Studio

Verwendung Mit Breakpoints wird überprüft, ob Bereiche des Codes ausgeführt werden können oder ob sie unerreichbar sind. Sie werden ebenfalls verwendet, um vor einer

Codestelle, bei der ein Fehler vermutet wird, den Programmablauf anzuhalten, so dass das verdächtige Verhalten des Programms von dort an mit einer Schritt-für-Schritt-Ausführung fortgesetzt (siehe Abschnitt 6.2.1) oder die aktuelle Speicherbelegung angesehen werden kann.

Breakpoints erleichtern es, zur Laufzeit zu bestimmten Codestellen zu gelangen, um diese näher zu untersuchen. Ohne Breakpoints müsste in Einzelschritten zu der gewünschten Stelle navigiert werden, was unter Umständen viel Zeit in Anspruch nehmen kann.

Run To Cursor Mit dem Run-To-Cursor-Knopf (Abbildung 6.6) kann man im Debugmodus direkt zu der Codezeile laufen, auf der sich der Cursor momentan befindet. In diesem Fall wird vom Debugger ein einmaliger Breakpoint in die entsprechende Zeile gesetzt und die Ausführung gestartet. Angehalten wird nur dann, wenn diese Zeile auch erreichbar ist. Dieser Knopf steht nur zur Verfügung, wenn die Programmausführung momentan pausiert ist.



Abbildung 6.6: Der Run-To-Cursor-Knopf

Schritt-Für-Schritt-Ausführung

Die Schritt-Für-Schritt-Ausführung des Programms ermöglicht eine feinkörnige Kontrolle des Programmflusses. Hierzu bieten die Bedienelemente zur schrittweisen Abarbeitung verschiedene Funktionen an.



Abbildung 6.7: Die Knöpfe zur Schritt-Für-Schritt-Ausführung

1. Die *Step-Into* Funktion sorgt dafür, dass der nächste Befehl ausgeführt wird. Ist die folgende Anweisung ein Funktionsaufruf, so wird diese aufgerufen und die Programmausführung zu Beginn dieser Funktion angehalten.
2. Sollte die nächste auszuführende Anweisung ein Funktionsaufruf sein, so wird bei Betätigen des *Step-Over-Knops* die Unterfunktion ohne Schritt-für-Schritt Ausführung bearbeitet, danach springt der Debugger in die nächste Codezeile des aktuellen Kontrollflusses.
3. Der *Step-Out-Knopf* ermöglicht es schließlich, das Debuggen eines Unterfunktionsaufrufs abzubrechen und in den aufrufenden Kontext zurückzukehren.

Die Schritt-Für-Schritt-Ausführung des Programms wird meist in Kombination mit den Methoden zur Überwachung des Speichers angewendet. Dadurch kann beobachtet werden, wie sich einzelne Programmbefehle auf die Werte der Variablen auswirken.

6.2.2 Disassembler

Atmel Studio bietet die Möglichkeit den vom Compiler erzeugten Assemblercode, mit-
samt der zugehörigen mnemonischen Anweisungen, zu betrachten. Je höher die Optimie-
rungsstufe ist, desto stärker kann dieser Code strukturell vom ursprünglichen C-Code
abweichen, wobei die Semantik stets erhalten bleibt. Bei Betrachtung des Assembler-
codes kann exakt nachvollzogen werden, in welcher Reihenfolge der Prozessor welche
Befehle ausführt. Im Debug-Modus kann die Assembleransicht über den Befehl „Disas-
sembly“ (Alt + F8) geöffnet werden. Eventuell gesetzte Breakpoints stehen weiterhin zur



Abbildung 6.8: Der Disassembly-Knopf

Verfügung. Analog zur C-Ansicht markiert ein Pfeil die als nächstes auszuführende
Programmzeile. Der Programmcode kann anschließend auf die gleiche Weise wie der C-Code
durchlaufen werden. Das Debuggen des Assemblercodes ist aufgrund der Komplexität
oft die aufwändigste, jedoch gleichzeitig die beste Option, um ein mögliches Problem im
Detail zu verstehen.

Beispiel

Die folgende Abbildung zeigt links ein kleines Programm, welches in einer Endlosschleife
den Wert von Port A einliest, dessen Fakultät berechnet und das Ergebnis auf die Ports
B und C ausgibt. Das Programm wurde ohne Optimierung kompiliert. Auf der rechten

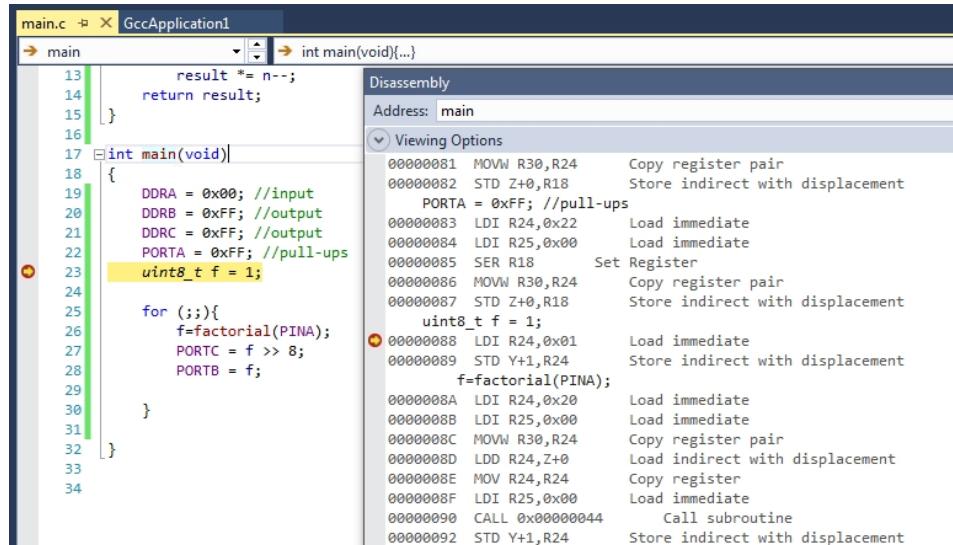


Abbildung 6.9: C-Code (links) und Assembleransicht des Programms (rechts)

Seite ist die Assembleransicht des Programms zu sehen. Über den Assemblerbefehlen

sind die ursprünglichen C Befehle notiert, aus denen der Compiler den Assemblercode generiert hat. Eine solche Zuordnung ist je nach Grad der Optimierung nicht immer möglich. Mit Hilfe des Handbuchs kann die Korrektheit der Assemblerbefehle überprüft werden. Beispielsweise wurde die Codezeile `uint8_t result = 1` durch zwei Assembleranweisungen ersetzt, welche im Folgendem erklärt werden.

```
-- Lade den Wert '1' (0x01) in das Register R24
LDI R24, 0x01
-- Speichere den Wert des Registers R24 an Adresse,
-- die durch die Variable Y+1 referenziert wird
STD Y+1, R24
```

6.2.3 Überwachung des Speichers

Um den aktuellen Zustand des Mikrocontrollers zu überprüfen, bietet sich die Überwachung der verschiedenen Speicher und der Peripherie an. Entsprechende Fenster lassen sich über die in Abbildung 6.10 hervorgehobenen Knöpfe öffnen und schließen. Im Folgenden werden mehrere Werkzeuge bzw. Fenster vorgestellt, mit denen man Speicherbereiche betrachten kann.

HINWEIS

Alle diese Werkzeuge haben gemeinsam, dass sie nur dann einen definierten Wert anzeigen, wenn die Programmausführung pausiert ist. Eine Änderung des jeweils betrachteten Speicherteils wird in roter Farbe dargestellt.



Abbildung 6.10: Die Bedienelemente zur Speicherüberwachung

Variablenüberwachung

Sobald das Programm durch Erreichen eines Breakpoints oder manuelles Pausieren angehalten wurde, lassen sich die Werte von Variablen auslesen, die sich im Sichtbereich des momentanen Ausführungskontextes befinden. Dazu muss die Variable mit der rechten Maustaste angeklickt, und „Add Watch“ aus dem Kontextmenü gewählt werden. Es öffnet sich ein neues Fenster namens „Watch“, in dem die Werte sowie Typ und Adresse aller beobachteten Variablen aufgelistet werden.

HINWEIS

In einigen Fällen wird der Inhalt einer Variablen nicht angezeigt. Auch dies hängt meist mit Compileroptimierungen zusammen. Weitere Hinweise mit Lösungsvorschlägen zu diesem Thema können in Abschnitt 6.3.2 gefunden werden.

Beispiel Ein Beispiel für eine solche Überwachung ist in Abbildung 6.11 dargestellt. Das Watch-Fenster zeigt die momentan überwachten Variablen `Endian`, `value` und `result` an. Die Variable `Endian` wurde im Zuge der Optimierung entfernt. Die zweite überwachte Variable `value` ist vom Typ `uint8_t` (ohne Vorzeichen, 8 Bit lang) und hat derzeit den Wert 1. Außerdem ist erkennbar, dass sie sich an der Adresse 4347 (0x10fb) im SRAM befindet.

Watch 1		
Name	Value	Type
Endian	Optimized away	Error
value	1	uint8_t{data}@0x10fb ([R28]+1)
result	Unknown identifier	Error

Watch 1 Watch 2 Processor Status Memory1 I/O

Abbildung 6.11: Das Watch-Fenster

In diesem Beispiel kann die Variable `result` nicht beobachtet werden. Das hängt damit zusammen, dass lokale Variablen zur Laufzeit auf dem Stack abgelegt werden und somit nur existieren, wenn diese Funktion gerade bearbeitet wird. Siehe Kapitel 6.3.2

Verwendung Die Variablenüberwachung wird typischerweise eingesetzt, wenn sich der Programmfluss oder die Ausgaben anders entwickeln, als erwartet. Beispielsweise können komplizierte Ausdrücke in bedingten Anweisungen Fehler enthalten, die nur schwer nachvollziehbar sind. Um einen solchen Fehler in einem Programmstück zu finden, ist es oft hilfreich die Werte aller beteiligten Variablen zu kennen, um den Programmverlauf selbst nachrechnen zu können. Das Beispiel in Listing 6.1 zeigt eine umfangreiche Bedingung, in der durch die Analyse der Variablen zur Laufzeit eventuelle Fehler aufgespürt werden können.

```

1 unsigned char inA = PINA;
2 unsigned char inB = PINB;
3 inB = inB/45 + 7 - PINC;
4 if (inA & (0x41 ^ PIN_DEFINE) != inB*inB) {
5   lcd_writeString("Messwert ok!");
6 }
```

Listing 6.1: Komplizierte if-Abfrage

Anzeige des Speichers

Insbesondere bei hardwarenahen Anwendungen gibt es Situationen, in denen größere Datenmengen auf dem SRAM abgelegt werden, denen keine Variable zugeordnet ist. So kann beispielsweise ein empfangener Datenstrom variabler Länge in einem Ringpuffer gespeichert sein, dessen Anfang und Ende durch die Zeiger `head` bzw. `tail` angegeben werden. Um Fehler in einem solchen Datenstrom zu finden, gibt es die Möglichkeit sich den entsprechenden SRAM-Bereich anzeigen zu lassen. Die Speicheranzeige ist exemplarisch in Abbildung 6.12 zu sehen. Die Speicheranzeige ist wie folgt zu lesen: Jede Zeile listet mehrere Bytes des jeweiligen Speichers auf. In diesem Beispiel ist oben links der Speicher „IRAM“ ausgewählt, sodass die angezeigten Werte dem Inhalt des SRAM entsprechen. In der linken Spalte wird die Adresse des ersten Bytes der jeweiligen Zeile angezeigt, danach folgt eine byteweise Wiedergabe des Speichers in hexadezimaler Darstellung. In der dritten Spalte findet man die gleichen Bytes als ASCII Zeichen. Die Anzahl der Bytes, die in einer Zeile angezeigt werden, hängt von der Breite des Fensters ab (hier wurde *Cols: Auto* eingestellt). Die Speicheranzeige bietet darüber hinaus die Möglichkeit, Werte in dem Speicher zu manipulieren. Im Kontextmenü finden sich zudem verschiedene Optionen, um die Darstellung der Werte anzupassen.

Beispiel In Abbildung 6.12 steht an Adresse 0x0022 des SRAM der Wert 0xFF und an Adresse 0x005E der Wert 10.

Neben Programmspeicher und SRAM können ebenfalls EEPROM, I/O und die 33 Register angezeigt werden. Zur Beobachtung der Prozessorregister bieten sich die weiter unten in diesem Abschnitt beschriebenen Ansichten besser an.

Verwendung Die Verwendung des Memoryfensters ist immer dann sinnvoll, wenn längere Blöcke eines Speichers, die nicht durch eine Variable direkt angegeben werden können, durchsucht werden müssen. Der Programmspeicher wird erst interessant, wenn dieser zur Laufzeit manuell beschrieben wird, etwa durch die Implementierung eines Bootloaders.

Anzeige des Prozessorstatus und der Register

Die Prozessor- und Registeransicht stellt den Zustand des Prozessors und den Inhalt der 32 general purpose Register während der Ausführung dar. Das in Abbildung 6.13

6 Hinweise zum Debuggen

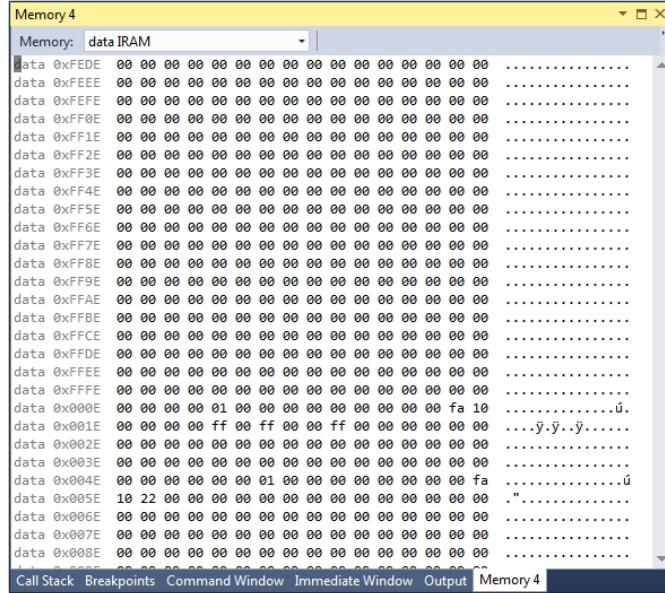


Abbildung 6.12: Die Anzeige des Mikrocontrollerspeichers

gezeigte Fenster befindet sich automatisch auf der rechten Seite der Umgebung und ist verfügbar, sobald die Ausführung pausiert wird. In dem Beispiel kann abgelesen werden, dass derzeit das Interruptflag (Bit 7) des SREG auf 0 und das Zeroflag (Bit 1) auf 1 stehen. Durch einen Mausklick auf die Kästen in der Zeile Status Register können die Bits gesetzt oder gelöscht werden.

Ferner kann der Stack Pointer von Interesse sein, welcher momentan auf der Adresse 0x10F8 steht, was darauf hindeutet, dass derzeit 7 Byte des Stacks belegt werden¹.

Verwendung Bei Ansteuerung des Mikrocontrollers in höheren Programmiersprachen ist der Inhalt der general purpose Register für den Programmierer meistens nicht relevant, da diese vom Compiler zugewiesen und verwaltet werden. Ausnahmen hiervon entstehen dann, wenn in den durch den Compiler festgelegten Programmfluss eingegriffen wird oder eigene Anweisungen in Assembler eingefügt werden.

Hilfreich kann der Program Counter in Kombination mit dem Memoryfenster sein. Bei der Implementierung eines Betriebssystems mit mehreren Stacks kann das Beobachten des Stack Pointers ebenfalls beim Auffinden von Fehlern helfen.

Bei der Arbeit mit dem Disassembler sind X, Y, Z, sowie der Stack Pointer und die 32 Register besonders interessant.

¹Der ATmega644 hat einen SRAM von 0x1000 Byte (4KB). Dieser beginnt bei Adresse 0x100, damit ist die höchste Adresse 0x10FF. Detailliertere Informationen zu diesem Thema können dem Datenblatt des Mikrocontrollers ATmega 644 entnommen werden.

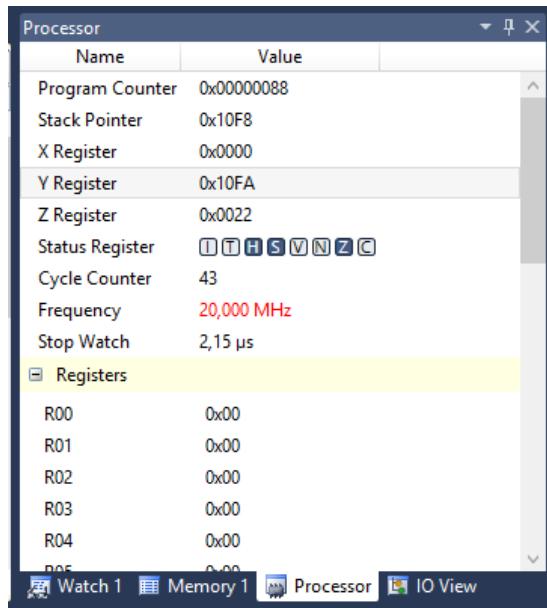


Abbildung 6.13: Die Anzeige des Prozessorstatus und der Register

Der X, Y und Z-Pointer Das Registerpaar R30 und R31 kann zu einem einzigen logischen Register zusammengefasst werden – dem Z-Pointer. Der Z-Pointer kann spezielle Aufgaben übernehmen, indem er als Adressangabe fungiert. Die Speicherzelle im SRAM, auf die der Z-Pointer zeigt, kann für Lade- bzw. Speichervorgänge verwendet werden. Anstatt die Speicheradresse der benötigten Zelle direkt im Programmcode anzugeben, kann sie vor der nötigen Operation zunächst in den Z-Pointer geladen werden. Der Hauptvorteil davon besteht darin, dass mit den Registern R30 und R31, wie mit den anderen Registern auch, Arithmetik betrieben werden kann. Das bedeutet insbesondere, dass das vorherige Laden der nötigen Adresse im Code entfällt und der Z-Pointer mit einfachen Operationen zur Laufzeit manipuliert werden kann.

Neben dem Z-Pointer gibt es noch den X-Pointer bzw. Y-Pointer. Diese werden durch die Registerpaare R26, R27 (X-Pointer) bzw. R28, R29 (Y-Pointer) dargestellt. Im Allgemeinen haben diese dieselben Eigenschaften wie der Z-Pointer.

I/O-Anzeige

Mithilfe der IO-View kann beispielsweise festgestellt werden, ob ein Pin als Ein- oder Ausgang gesetzt, ob ein Taster gedrückt ist oder auch welcher Wert sich momentan im Timer-Register befindet. Dort werden alle Register des Mikrocontrollers aufgelistet. Geöffnet werden kann die IO-View über „Debug“ → „Windows“ → „I/O View“. Diese Ansicht erlaubt sogar das Anzeigen und Ändern von internen Registern, die die Konfiguration des Prozessors selbst betreffen.

Hinweis: Die Inhalte der Register werden (wie alle anderen Debug-Informationen in Atmel Studio) nur aktualisiert, wenn das Programm angehalten wird.

Beispiel In Abbildung 6.14 kann man die genaue Konfiguration der zu Port B gehörenden Register sehen. Die Datenrichtung aller zu Port B gehörigen Pins ist auf 0 gesetzt. Dies erkennt man zum einen an dem Wert 0x00 sowie den acht unausgefüllten weißen Kästchen im unteren Teil des Fensters. Das heißt, dass derzeit alle zu Port B gehörigen Pins als Eingang konfiguriert sind. Durch Anklicken eines der Kästchen kann die Belegung eines bestimmten Bits in einem der Register manipuliert werden.

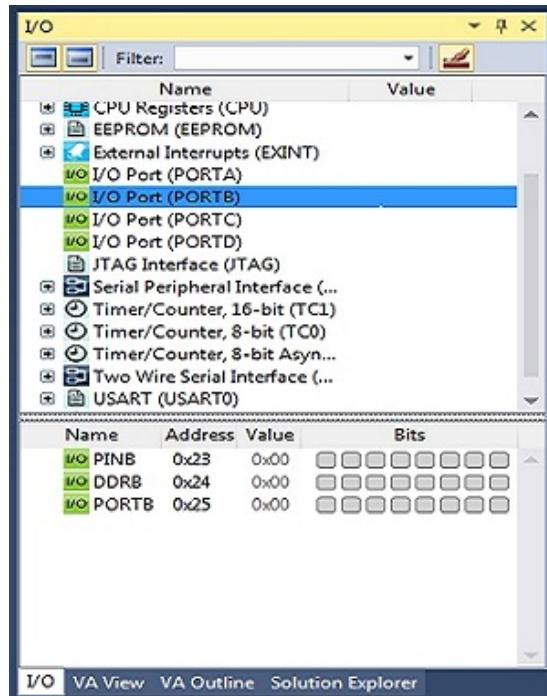


Abbildung 6.14: Die I/O-Anzeige

Verwendung Die Verwendung der I/O-Anzeige ist vielfältig, und immer dann besonders hilfreich, wenn man die Konfiguration eines bestimmten Gerätes (etwa eines Timers) zur Laufzeit überprüfen und gegebenenfalls ändern möchte. Zudem können von der Außenwelt eingehende Signale überwacht werden. Zum Beispiel kann während des Debuggens bequem getestet werden, ob ein Interrupt korrekt ausgeführt wird, wenn sich ein Pin an Port B ändert.

6.2.4 Nicht überwachbare Funktionalität

Beim Debuggen muss immer beachtet werden, dass das Verhalten des Programms, in dem nach Fehlern gesucht wird, durch Eingriffe verändert werden kann. Insbesondere bei

zeitkritischen Anwendungen, wie z. B. Interrupts oder einer synchronen Kommunikation, führt dies oft zu neuen Fehlern. In einem zeitkritischen Kommunikationsprotokoll führt das Anhalten des Programms notwendigerweise zum Abbruch der Kommunikation, da das Programm keine Befehle mehr verarbeiten kann. Um solche Anwendungen trotzdem debuggen zu können, muss auf andere Werkzeuge wie z. B. ein Oszilloskop oder einen LogicAnalyzer zurückgegriffen werden.

Um Debugging-Nachrichten oder Werte von Variablen auszugeben, kann das LC-Display verwendet werden. Dies garantiert, dass das Programm weiterläuft und die Belegung von Variablen und das Eintreten von Ereignissen dennoch überwacht werden kann. Je nach Anwendungsfall kann es jedoch zu kritischen Verzögerungen kommen. Die Ausgabe auf dem Display kostet zusätzliche Zeit. Im Fall einer asynchronen Datenübertragung, in der eingehende Datenbytes durch ein Interrupt behandelt werden, kann die Ausgabe auf dem Display die Laufzeit des Interrupts erheblich verlängern. Je verursachter Verzögerung kann es hierbei zu Datenverlust kommen. Aufgrund der zeitlichen Verzögerung hat die Verwendung des Displays zum Debugging auch den Nachteil, dass angezeigte Variablen zum Zeitpunkt der Ausgabe wieder veraltet sein könnten.

6.3 Probleme beim Debugging

Nicht immer lässt sich das entwickelte Programm auf dem Mikrocontroller ohne weiteres debuggen. Es können verschiedene Probleme auftreten, die das Debuggen erschweren oder in manchen Fällen sogar unmöglich machen. Diese Probleme gliedern sich ebenso wie die Debugging-Methoden in die Bereiche der Ablaufüberwachung und der Speicherüberwachung.

Die meisten dieser Probleme werden durch Compileroptimierungen verursacht, weshalb die Reduzierung der Optimierungsstufe eine erste Maßnahme sein sollte. Weitere Informationen zur Compileroptimierung finden Sie in Kapitel 3.4.

ACHTUNG

Das Herabsetzen der Optimierungsstufe verlangsamt die Abarbeitung des Programms und kann in bisher funktionsfähigen Teilen der Software zu Problemen mit dem Timing führen. Wenn das Programm, welches debuggt werden soll, durch das Herabsetzen der Optimierungsstufe neuartiges Fehlverhalten zeigt, sollte die Optimierung wieder auf eine höhere Stufe gestellt werden.

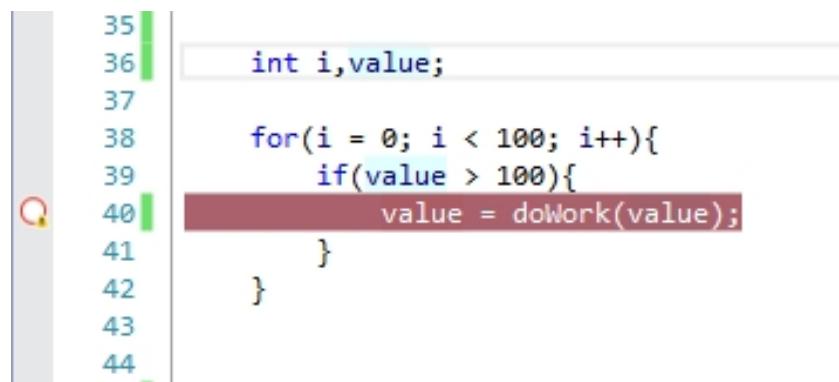
Im Folgenden werden zu den einzelnen Kategorien der Debugging-Probleme häufige Beispiele aufgezeigt, deren Ursachen erklärt und abschließend Vorschläge zur Lösung dieser Probleme gemacht.

6.3.1 Probleme bei der Programmüberwachung

Die Probleme bei der Überwachung des Programmablaufs stammen daher, dass der Debugger keinen Zusammenhang zwischen dem C-Quellcode des Programms und dem auf dem Mikrocontroller laufenden Maschinencode finden kann.

Falsch gesetzte und deaktivierte Breakpoints

Beim Klicken auf den Toggle-Breakpoint-Knopf kann es vorkommen, dass ein Breakpoint nicht in die aktuell ausgewählte Zeile gesetzt wird. Die Ursache hierfür besteht darin, dass diese Zeile keine Entsprechung im aktuell auf dem Mikrocontroller ausgeführten Programm hat. Dies kann der Debugger erst feststellen, wenn er bereits aktiv ist. Ist er inaktiv, werden zu Beginn des Debuggens alle Breakpoints überprüft. Diejenigen Breakpoints, denen keine Maschinenbefehle im Speicher des Mikrocontrollers entsprechen, werden deaktiviert. Abbildung 6.15 zeigt einen deaktivierten Breakpoint.



The screenshot shows a code editor with a sidebar and a main pane. The sidebar has a vertical scroll bar. The main pane displays the following C code:

```
35
36     int i,value;
37
38     for(i = 0; i < 100; i++){
39         if(value > 100){
40             value = doWork(value);
41         }
42     }
43
44
```

A red rectangular highlight surrounds the line of code starting with "value = doWork(value);". On the far left, there is a vertical grey bar with a small red circle containing a yellow question mark icon near the top, indicating a deactivation point or a warning.

Abbildung 6.15: Ein deaktivierter Breakpoint

Kann der Debugger keinen Breakpoint in die gewünschte Zeile setzen, so setzt er ihn stattdessen in die nächste Zeile, zu der er einen Maschinenbefehl kennt. Bereits gesetzte, aber deaktivierte Breakpoints werden nicht verschoben, es liegt in der Verantwortung des Programmierers sich um solche Breakpoints zu kümmern.

Übersprungene Anweisungen

Gelegentlich kann es vorkommen, dass einige Zeilen oder ganze Abschnitte des Codes bei der Schritt-Für-Schritt-Ausführung übersprungen werden. Der Grund für dieses Verhalten ist derselbe wie der für verschobene Breakpoints - der Compiler optimiert den Code so, dass die Assemblerbefehle nicht mehr den ursprünglichen C-Anweisungen zugewiesen werden können.

Ursachen für Probleme der Programmüberwachung

Es gibt zwei Ursachen dafür, wenn es dem Debugger nicht möglich ist, einen Zusammenhang zwischen dem aktuell ausgeführten Maschinencode und dem Quellcode zu erkennen. Die erste mögliche Ursache sind Präprozessordefinitionen, die verhindern können, dass ein bestimmter Teil des Quellcodes überhaupt kompiliert wird.

Beispiel 1 In diesem Beispiel werden alle Anweisungen zwischen den Zeilen `#if DEBUGGING` und `#endif` nicht kompiliert und sind daher vom Debuggen ausgenommen.

```
1 #define DEBUGGING 0
2 #if DEBUGGING
3 ... // Vom Präprozessor verworfener Code
4#endif
```

Listing 6.2: Nicht kompilierter Code, der vom Präprozessor verworfen wurde

Die zweite mögliche Ursache ist die Compileroptimierung. Es kann vorkommen, dass der Compiler Teile des Codes entfernt, mit anderen Teilen vereinigt oder tauscht. In diesen Fällen gibt es für manche C-Anweisungen keine Entsprechung im Maschinencode, weswegen das Debuggen solcher Stellen nicht möglich ist.

Beispiel 2 In diesem Beispiel soll ein Breakpoint in die Zeile 4 gesetzt werden. Bei der Optimierung des Codes wird das komplette if-Konstrukt jedoch vom Compiler entfernt.

```
1 int i = 0;
2 if (i == 13) // Das if-Konstrukt wird vom Compiler entfernt
3 {
4     i++;      // Hier kann kein Breakpoint gesetzt werden
5 }
```

Listing 6.3: Wegoptimierter Code

Eine mögliche Lösung des Problems besteht darin, die Variable `i` als `volatile` zu deklarieren. Diese Vorgehensweise wird im Abschnitt 6.3.2 genauer erläutert.

Lösungsvorschläge

Wenn Teile des Codes wegen Präprozessoranweisungen nicht kompiliert werden und deshalb nicht debuggt werden können, bleibt dem Entwickler keine andere Wahl, als die Präprozessoranweisungen entsprechend zu ändern.

Falls kein Zusammenhang zwischen C und Assembler hergestellt werden kann, lässt sich der Code dennoch debuggen, indem der Assemblercode Schritt für Schritt überprüft wird.

6.3.2 Probleme bei der Speicherüberwachung

Probleme bei der Überwachung der Speicherinhalte von Variablen treten oft auf, weil der Compiler die jeweiligen Variablen so weit optimiert hat, dass sie zur Laufzeit keine feste Speicheradresse mehr belegen.

Optimized away

Wenn im Watch-Fenster als Wert einer Variablen „Optimized away“ (Abbildung 6.16) erscheint, so wurde die betroffene Variable im Zuge der Optimierung entfernt.

Name	Value	Type
i	Optimized away	Error
j	Unknown identifier	Error

Abbildung 6.16: Variablen, deren Wert nicht überwacht werden kann

Unknown identifier

Eine Variable, die in dem Watch-Fenster durch „Unknown identifier“ (Abbildung 6.16) markiert ist, befindet sich nicht im momentanen Laufzeitkontext. Man betrachte Beispiel 6.4:

```

1 void firstFunction(void)
2 {
3     int start; // Wird nur lokal verwendet
4     ...
5 }
6 void secondFunction(void)
7 {
8     int i; // Variable start ist hier nicht bekannt
9     ...
10 }
```

Listing 6.4: Sichtbarkeitsbereiche der lokalen Variablen

Wenn die Variable `start` beim Debuggen der Funktion `secondFunction` überwacht wurde, existiert sie nicht mehr, sobald die Funktion verlassen wird. Befindet sich der Debugger außerhalb dieser Funktion und die Variable `start` wurde nicht aus dem Watch-Fenster entfernt, kann für sie kein aktueller Wert ermittelt werden. Solche Variablen werden als „*Unknown identifier*“ markiert.

Ursachen für Probleme bei der Speicherüberwachung

Wie bereits beschrieben, entstehen solche Probleme für gewöhnlich dann, wenn bestimmte Variablen im Zuge der Compileroptimierung aus dem Arbeitsspeicher entfernt wurden. Dafür sind mehrere Gründe denkbar: Wird eine Variable besonders häufig gelesen oder geschrieben, wird sie vom Compiler in einem Prozessorregister abgelegt. So kann der Zugriff auf die Variable (im Vergleich zu den im SRAM gespeicherten Variablen) schneller erfolgen. Die gleiche Überlegung bietet sich für Variablen an, die nur während eines kurzen Programmabschnitts gebraucht werden, wie etwa Laufvariablen von Schleifen, und keinen persistenten Speicherplatz benötigen. Für das Watch-Fenster zählen Register nicht als gültige Speicheradressen für Variablen.

Beispiel In diesem Beispiel wird die Variable `i` im Watch-Fenster mit „*Optimized away*“ verzeichnet, weil sie nach der Optimierung nur in den Prozessorregistern gespeichert wird.

```
1 int i;
2 for (i = 0; i < LOOP_MAX; i++)
3 {
4     ... // Anweisungen
5 }
```

Listing 6.5: Wegoptimierte Variable

Lösungsvorschläge

Wenn Codezeile, die man debuggen möchte, durch die Compileroptimierung entfernt werden, können darin vorkommende Variablen durch das Schlüsselwort `volatile` von der Optimierung ausgenommen werden. In diesem Fall werden ebenfalls alle Anweisungen, die auf diesen Variablen arbeiten, von der Optimierung ausgeschlossen. Eine genauere Beschreibung von `volatile` kann im Kapitel 5.1.12 „Einführung in die C Programmierung“ gefunden werden.

Das Beispiel 2 aus dem Abschnitt 6.3.1 nochmal unter Verwendung des Schlüsselwortes `volatile`:

```

1 int volatile i = 0; // Die Variable i nicht mehr optimieren!
2 if (i == 13)          // Die Zeile wird auch nicht optimiert
3 {
4     i++;              // Jetzt kann hier ein Breakpoint gesetzt
5     werden
}

```

Listing 6.6: Verwendung von volatile

Um die Compileroptimierung global auszuschalten, lässt sich mithilfe des Dropdown-Menüs 6.17 in der Toolbar von AtmelStudio eine *Build-Configuration* auswählen. *Debug* steht dabei für die Optimierungsstufe -O0 und *Release* für die Optimierungsstufe -O2.

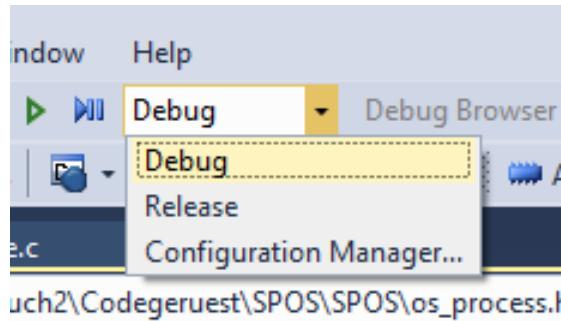


Abbildung 6.17: Verändern der Optimierungsstufe

6.4 Fallbeispiele aus dem Praktikum Systemprogrammierung

Fehler können verschiedene Ursachen haben. In diesem Kapitel werden nur einige der häufigsten davon dargestellt und es wird exemplarisch gezeigt, wie sie lokalisiert und beseitigt werden können. Es soll vor allem die Methodik des Debuggens beispielhaft erklärt werden. Die hier angeführten Fehler sind zwar konstruiert, geben aber teilweise stark vereinfachte, reale Probleme aus der Praxis wieder.

6.4.1 Fehler durch falsche Datentypen

Steht dem Entwickler nur eine eingeschränkte Menge von Ressourcen zur Verfügung, ist es notwendig, möglichst sparsam damit umzugehen. Aus diesem Grund sieht man sich oft gezwungen, den kleinstmöglichen Datentyp für Variablen zu wählen, der die gestellten Anforderungen erfüllt. In manchen Fällen entstehen dadurch nur schwer auffindbare Fehler.

Dies wird an folgendem Beispiel verdeutlicht (Optimierung deaktiviert):

```

1 #define LENGTH 256
2
3 uint8_t i;
4 uint8_t array[LENGTH];
5
6 for (i = 0; i < LENGTH; i++) {
7     array[i] = receive_char();
8 }
9 lcd_writeInt(array[0]);
10 ...
11 ...

```

Listing 6.7: Schwer auffindbarer Typfehler

Es sollen einige Byte empfangen werden und anschließend wird das erste Byte auf dem Display ausgegeben. Nach der Ausführung dieses Programms stellt man fest, dass nach einiger Zeit auf dem Display immer noch nichts zu sehen ist. Wenn angenommen werden kann, dass die relevanten Komponenten auf der Versuchsplatine richtig verbunden sind und die Übertragung der Daten korrekt verläuft, bleiben auf den ersten Blick nur wenige mögliche Ursachen denkbar:

- Die Funktion `receive_char` ist fehlerhaft und terminiert nicht.
- Die Funktion `lcd_writeInt` ist fehlerhaft und gibt nichts aus.
- Ein anderer, parallel laufender Prozess löscht das Display unmittelbar nach der Ausgabe der Daten.

Im ersten Schritt des Debuggens sollte ein Breakpoint in die Zeile 9 gesetzt werden, um die Funktionsweise von `lcd_writeInt` schrittweise zu überwachen (Taste F11). Die Überlegung ist simpel: Kann das Programm hier angehalten werden, widerlegt dies den ersten Punkt in der Liste der möglichen Fehler, da `receive_char` offensichtlich terminiert ist. Die letzten zwei Ursachen können in einem Schritt unmittelbar überprüft werden.

Man startet das Programm erneut und stellt fest, dass der Breakpoint nie erreicht wird. Somit liegt die Fehlerursache nicht in der Ausgabe. Die erste der aufgestellten Hypothesen muss daher näher untersucht werden. Daher wird ein neuer Breakpoint in die Zeile 7 gesetzt. Nach dem Neustart hält der Debugger den Programmablauf an der korrekten Stelle an und der Benutzer kann mit *StepOver* (F10) leicht überprüfen, dass `receive_value` nach dem Empfangen von Daten wieder verlassen wird.

Es bleibt zu untersuchen, ob dies immer der Fall ist, oder ob es Werte für `i` gibt, für die das Programm „hängen“ bleibt. Nach Entfernen des Breakpoints in der Zeile 7 lässt man das Programm eine Weile laufen und hält es dann mit *Pause* (Strg + F5) wieder an. Es kann überprüft werden, dass `receive_value` in jedem Schleifendurchlauf korrekt verlassen wird. Offensichtlich liegt das Problem nicht in der Funktion `receive_value`.

Aber da die Variable *i* jedes Mal erhöht wird, müsste die Schleife früher oder später terminieren. Aus diesem Grund sollen im nächsten Schritt die Werte von *i* betrachtet werden. Dazu muss mit der rechten Maustaste auf *i* geklickt und *Add Watch: i* aus dem Kontextmenü ausgewählt werden. Das Watchfenster öffnet sich und zeigt als den aktuellen Wert von *i* den Wert 250 an. Der Benutzer drückt 5-mal auf *StepOver* (F10) und sieht, wie sich *i* jedes Mal um eins erhöht. Im nächsten Durchlauf müsste die Schleife abbrechen. Stattdessen ändert sich der Wert von *i* auf 0. Damit wurde die Fehlerursache gefunden - in der Variable *i* hat es einen *Überlauf* (engl. Overflow) gegeben, der verhindert, dass die Bedingung *i < LENGTH* verletzt wird. Eine genaue Betrachtung des Datentyps von *i* offenbart schließlich den eigentlichen Fehler: Eine *unsigned char* Variable mit dem Wertebereich [0..255] ist immer kleiner als *LENGTH* (256), womit die Schleife nicht terminieren kann.

Daraus ergibt sich unmittelbar eine mögliche Lösung - der Datentyp von *i* muss geändert werden, beispielsweise in *unsigned short*.

6.4.2 Unerwartetes Verhalten durch Optimierung

Compiler sind üblicherweise so konstruiert, dass sie selbstständige Optimierung von Quellcode durchführen. Die wichtigste Anforderung - die Korrektheit - ist dabei nicht zu verletzen. Von sehr seltenen Fehlern in Compilern abgesehen, wird diese Anforderung unter normalen Umständen erfüllt. In hardwarenahen Anwendungen greift man jedoch oft eigenhändig weit in einen Arbeitsbereich hinein, der ansonsten vom Compiler alleine verwaltet wird. Es werden beispielsweise Register oder Variablen geändert, ohne dass dies im Code explizit gefordert wird (durch eine Zustandsänderung der Außenwelt oder einen parallel laufenden Task). Solche Dinge kann der Compiler nicht in seinem Optimierungsalgorithmus vorhersehen, sodass die Optimierung manchmal fehlerhaften Maschinencode generiert, obwohl der ursprüngliche C-Code korrekt ist.

Dies wird am folgenden Beispiel, in dem zwei parallel laufende Tasks vorkommen, verdeutlicht:

```

1 // Defines & globale Variablen
2 unsigned char* pb = NULL;
3
4 void task1(void) {
5     unsigned char b = 0xBB;
6     unsigned char a = 0xAA;
7     pb = &b;           // Adresse von b an task2 übergeben
8     while (a == 0xAA){} // Warten bis a sich ändert
9     lcd_writeString("a changed!");
10}
11
12 void task2(void) {
13     while (!pb){} // Warten auf task1
14     pb--;          // Zeigt auf die Adresse VOR b (wo a liegt)
15     *pb = 0;         // In a wird eine 0 geschrieben

```

Listing 6.8: Durch Optimierung verfälschtes Programmverhalten

Was gemeint ist

Die Variable `pb` ist global und dient der Kommunikation zwischen `task1` und `task2`. Wird `task1` ausgeführt, so wird zunächst die Variable `b` auf den Stack gelegt und an diese Stelle der Wert `0xBB` geschrieben. Anschließend wird auf die Stelle *davor* (das Stackprinzip) die Variable `a` gespeichert. An der entsprechenden Stelle im Speicher sieht man den Wert `AABB` (Abbildung 6.18). Als nächstes wird die Speicherstelle von `b` mit Hilfe von `pb` an `task2` übergeben und gewartet bis `a` sich ändert.

`task2` wartet seinerseits bis ihm die Adresse von `b` mitgeteilt wird und berechnet daraus die Adresse von `a` (die Speicherstelle *davor*). Damit „weiß“ `task2` wo `a` liegt und schreibt an diese Adresse eine `0` (Abbildung 6.19). Sobald `task1` wieder ausgeführt wird, müsste die `while`-Schleife verlassen und der Text „`a changed!`“ auf dem LCD ausgegeben werden.

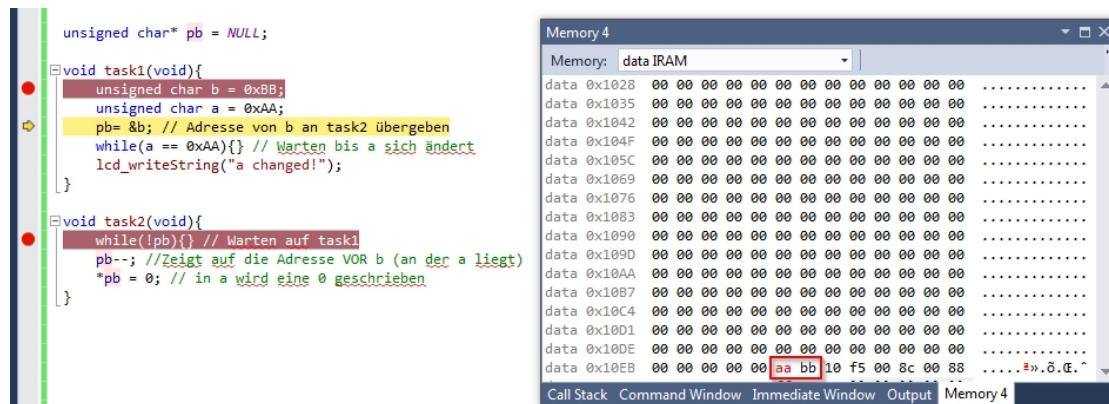


Abbildung 6.18: task1: SRAM ohne Kompileroptimierung

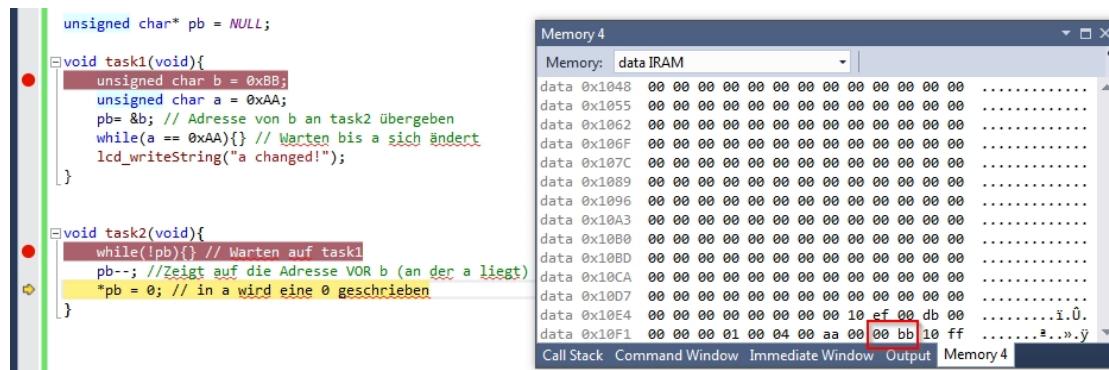


Abbildung 6.19: task2: SRAM mit Kompileroptimierung

Was tatsächlich geschieht

Der Compiler analysiert den Code in `task1` und erkennt, dass die Adresse von `b` bekannt sein muss, daher wird sie unter Umständen an einer anderen Stelle im Code benutzt. Auf die Adresse von `a` trifft dies jedoch nicht zu, da sie in keiner globalen Variable gespeichert wird. Vereinfacht betrachtet, stellt der Compiler zu diesem Zeitpunkt fest, dass sich `a` niemals ändern wird. Aus diesem Grund können alle Vorkommen von `a` durch den Wert `0xAA` ersetzt werden, mit dem `a` initialisiert wurde. Die Bedingung der `while`-Schleife reduziert sich zu `0xAA == 0xAA`, was durch `true` ersetzt werden kann. Infolgedessen wird der Befehl `lcd_writeString` entfernt, da er nicht erreichbar ist. Der „optimierte“ `task1` sieht wie folgt aus:

```
1 void task1(void) {
2     unsigned char b = 0xBB;
3     pb = &b;
4     while (true) {}
5 }
```

Listing 6.9: Reduzierter Code aus Listing 6.8

Fehlersuche

Zunächst wird der Benutzer feststellen, dass die erwartete Zeichenkette nicht ausgegeben wird. Der Versuch, einen Breakpoint in Zeile 10 zu setzen, wird scheitern, da zu diesem Befehl kein Assemblercode existiert. Der nächste Schritt könnte sein, einen Breakpoint in die Zeile 9 zu setzen, um die Schleife und den Wert von `a` beobachten zu können. Im Watchfenster wird man jedoch feststellen, dass `a` aufgrund der Optimierung nicht beobachtbar ist. Dies erlaubt bereits eine gute Hypothese, warum sich das Programm nicht so verhält, wie erwartet. Sucht man zusätzlich im Memoryfenster auf dem Stack von `task1` nach BB, stellt man fest, dass der Wert AABB dort nicht mehr vorhanden ist. Es liegt sofort auf der Hand, dass die Anweisung in Zeile 16 für `task1` wirkungslos bleibt und dieser nicht mehr terminieren kann.

Mögliche Lösungsansätze wären beispielsweise, `a` mit dem Schlüsselwort `volatile` zu deklarieren oder die Adresse von `a` in einem globalen Zeiger zu speichern.

7 Dokumentation mit Doxygen

Doxygen ist ein Open-Source-Werkzeug zur Dokumentation von Quellcode. Es wird dazu verwendet, übersichtliche Dokumentationen über den Aufbau eines Programms direkt aus dem Quelltext zu generieren. Basierend auf Markierungen (sog. Tags) in Kommentaren werden unter anderem verwendete Variablen, Funktionen und Dateien dokumentiert und zusammen als HTML-, L^AT_EX- oder RTF-Dokument ausgegeben.

7.1 Doxygen im Praktikum

Im Praktikum Systemprogrammierung müssen keine Dokumentationen mit Doxygen von den Studierenden erstellt werden. Stattdessen wird zusammen mit den Vorbereitungsunterlagen zu jedem Versuch eine Doxygen-Dokumentation im l2p zur Verfügung stehen. Die jeweilige Dokumentation soll das Verständnis der zu implementierenden Variablen, Konstanten, Funktionen usw. unterstützen.

Die Strukturen und Zusammenhänge innerhalb der im Laufe des Praktikums entwickelten Software werden von Versuch zu Versuch komplexer. Daher ist es nicht möglich, jede einzelne Methode explizit in den Unterlagen zu erklären. Stattdessen werden diese Informationen in die Doxygen-Dokumentation ausgelagert, welche eine Beschreibung jeder im Projekt vorhandenen Header-Datei enthält.

HINWEIS

In diesem Praktikum ist die Verwendung von Doxygen-Kommentaren **im eigenen Code nicht verpflichtend**. Es wird jedoch dringend empfohlen, eine strukturiertere Methodik für die Erstellung von Kommentaren zu einzuhalten, wie sie z.B. von Doxygen gefordert wird. Wenn Sie sich aus diesen Gründen mit Doxygen auseinandersetzen, finden Sie in den nächsten Kapiteln eine kurze Einführung.

7.2 Ausgabe von Doxygen

Abbildung 7.1 zeigt die Startseite der Doxygen-Dokumentation (Datei `index.html`) zu einem fiktiven Versuch X.

Wird in diesem Dokument beispielsweise die Datei `os_core.h` ausgewählt, so wird ihr Inhalt detaillierter dargestellt (Abb. 7.2). Hier werden alle implementierten Methoden

7 Dokumentation mit Doxygen

erst aufgelistet und anschließend zusammen mit ihren Parametern (und ggf. auch Rückgabewerten) erklärt. So ergibt sich die Möglichkeit, den Aufbau des ganzen Projekts besser zu veranschaulichen. Dies kann den Aufwand für eine eigene Implementierung wesentlich verringern.

Es ist also sehr hilfreich, die vorhandene Dokumentation in den Prozess der Entwicklung mit einzubeziehen, da darin wichtige Informationen zu den konkreten Programmierproblemen (Namen der Variablen oder Methoden, Hilfsfunktionen, Konstanten usw.) abgebildet sind.

The screenshot shows the start page of a Doxygen-generated documentation for the SPOS system. At the top, there is a navigation bar with tabs for "Main Page", "Data Structures", and "Files". The "Main Page" tab is currently selected. On the left, a sidebar menu under the heading "SPOS" lists "SPOS Manual", "Data Structures", and "Files". The main content area is titled "SPOS Manual" and contains the following text:

Praktikum Systemprogrammierung: Operating System (SPOS)
SPOS is an operation system that was developed for education purposes at the Embedded Systems Laboratory (i11), RWTH Aachen University.
All provided materials are copyrighted by the i11 and licensed only for use in the course 'Praktikum Systemprogrammierung'.

This documentation describes the functionalities of the given SPOS framework. It contains some basic information about the data types, variables and functions of SPOS.

Remark

This document is designed as a guideline to help you with your own implementation of SPOS. It is not necessary that you implement every data type, variable or function in this Doxygen documentation. You just need to implement the entities mentioned and explained in the PDF lab documentation for the respective lab term. This Doxygen document may contain additional help functions or variables that can, but do not have to be implemented and used by you in your implementation.

However, the signatures for the functions in your PDF lab documentation are consistent with this document, so you can use it as a reference for your homework. Note that this document is generated for a version of SPOS after exercise 3. For the purpose of writing your own programs in progs.c the `lcd.h` might be of special importance to you. Furthermore is it vital for you to precisely study `defines.h`.

At the bottom right of the main content area, there is a footer note: "Generated on Fri May 13 2016 13:48:23 for SPOS by **doxygen** 1.8.10".

Abbildung 7.1: Die Startseite einer beispielhaften Doxygen-Dokumentation

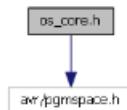
7 Dokumentation mit Doxygen

os_core.h File Reference

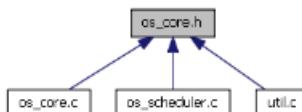
Core of the OS. More...

```
#include <avr/pgmspace.h>
```

Include dependency graph for os_core.h:



This graph shows which files directly or indirectly include this file:



Macros

```
#define os_error(str) os_errorPStr(PSTR(str))
```

Handy define to specify error messages directly.

Functions

```
void os_init_timer(void)
```

Initializes timers. More...

```
void os_init(void)
```

Initializes OS. More...

```
void os_errorPStr(char const *str)
```

Shows error on display and terminates program. More...

Detailed Description

Contains functionality to for core OS operations.

Author

Lehrstuhl Informatik 11 - RWTH Aachen

Date

2013

Version

2.0

Function Documentation

```
void os_errorPStr(char const * str)
```

Terminates the OS and displays a corresponding error on the LCD.

Parameters

str The error to be displayed

Here is the call graph for this function:

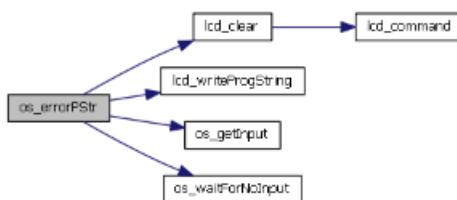


Abbildung 7.2: Detailansicht der `os_core.h`

7.3 Verwendung von Doxygen

Auf der Doxygen-Homepage (<http://www.doxygen.org>) stehen sowohl das Tool Doxygen, als auch eine grafische Benutzerschnittstelle zum Download bereit. Doxygen wird für die Betriebssysteme Windows, Linux und Mac OS X angeboten.

7.3.1 Konfiguration

Die grafische Benutzerschnittstelle bietet einen Assistenten an, der einen Großteil der Einstellungen eigenständig vornehmen kann: den *Doxywizard* zur Erzeugung von *Doxyfiles* (in Anlehnung an den Begriff *Makefiles*). In einem Doxyfile werden alle Einstellungen gespeichert, die für die Erzeugung einer Dokumentation berücksichtigt werden sollen. Der Assistent (Abb. 7.3) führt den Benutzer durch die verschiedenen Konfigurationsmöglichkeiten.

Nachdem ein Doxyfile konfiguriert wurde, muss es gespeichert werden. Die zuletzt verwendeten Doxyfiles werden zum schnellen Laden im *File*-Menü hinterlegt.

Doxygen benötigt auch ein Arbeitsverzeichnis, von dem aus es relative Pfadangaben auflöst. Hier kann das Projektverzeichnis mit dem Doxyfile genutzt werden.

7.3.2 Erzeugen der Dokumentation

Mit einem Klick auf *Run doxygen* im Register *Run* wird eine Dokumentation des Programmquellcodes generiert und im vorgegebenen Projektverzeichnis abgelegt. Für den Fall, dass die Ausgabe im HTML-Format erfolgt, kann die erstellte Datei *index.html* in einem beliebigen Browser geöffnet werden (Abb. 7.4).

Ohne spezielle Kommentare wird die Dokumentation im Wesentlichen aus alphabetischen Verzeichnissen von Datei- und Funktionsnamen bestehen. Im Folgenden werden Techniken zur Kommentierung von Quellcode vorgestellt, mit deren Hilfe eine umfassende Dokumentation erzeugt werden kann.

7 Dokumentation mit Doxygen

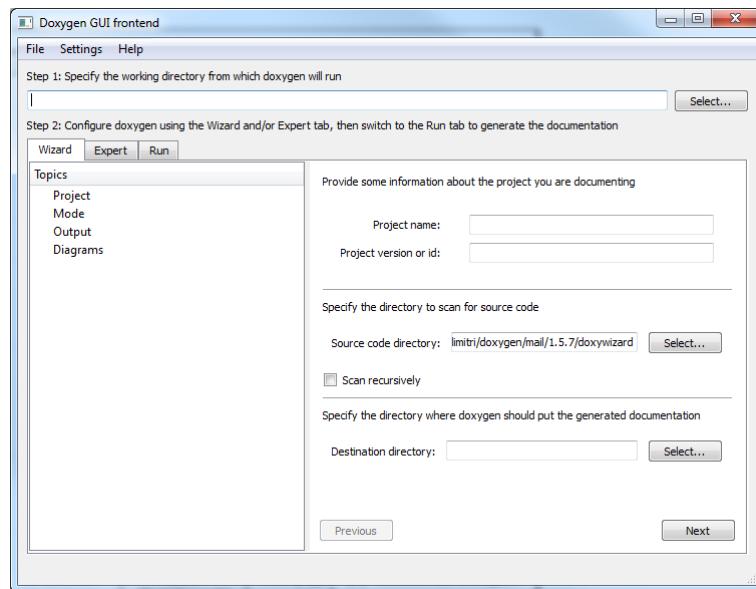


Abbildung 7.3: Der Doxygen-Assistent

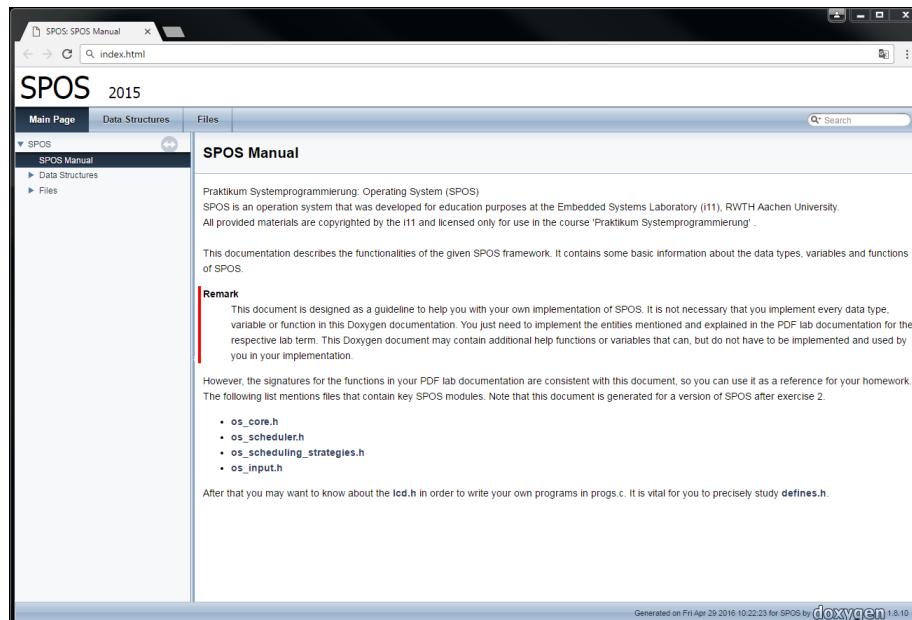


Abbildung 7.4: Die Startseite der neu erstellten Dokumentation

7.4 Doxygen-Kommentare

7.4.1 Grundlagen

Für die Erstellung einer Dokumentation verwendet Doxygen speziell gekennzeichnete Kommentare. Wenn ein Kommentar für die Dokumentation verwendet werden soll, wird einfach als erstes Kommentarzeichen ein '!' geschrieben.

```

1 // Normaler Kurzkommentar
2 //! Doxygen-Kurzkommentar
3
4 /*
5  * Ausführlicher normaler
6  * Kommentar
7  */
8
9 /*!
10  * Ausführlicher
11  * Doxygenkommentar
12 */

```

Listing 7.1: Unterschied zwischen normalen Kommentaren und Doxygen-Kommentaren

Das Dokumentationswerkzeug Doxygen erstellt Beschreibungen zu den Programmierschnittstellen eines Codes. Daher macht es nur Sinn, Doxygen-Kommentare für solche Objekte zu erstellen, die in einer Dokumentation von Interesse sind. Dies sind im Wesentlichen Dateien, Funktionen, Konstanten und Datentypen. Doxygen ordnet von sich aus Kommentare dem nächsten solchen Code-Objekt zu.

Die nächsten Codebeispiele aus einer .h- und einer .c-Datei zeigen den Unterschied zwischen einem Kurzkommentar und einem ausführlichen Kommentar. Die aus diesen Codebeispielen erzeugte Doxygen-Dokumentation ist in Abb. 7.5 dargestellt.

Code in der Header-Datei:

```

1 //! Function used to allocate private memory.
2 MemAddr os_malloc(heap* heap, size_t size);

```

Listing 7.2: Kurzer Kommentar in der Header-Datei

Code in der C-Datei:

```

1  /*
2   * Allocates a chunk of memory on the medium given by the
3   * driver and reserves it for the current process.
4   *
5   * \param heap    The driver to be used
6   * \param size    The amount of memory to be allocated in
7   *                Bytes. Must be able to handle a single byte
8   *                and values greater than 255.
9   * \returns      A pointer to the first Byte of the
10  *               allocated chunk.\n 0 if
11  *               allocation fails (0 is never a valid address)
12  *
13 MemAddr os_malloc(Heap* heap, size_t size ){
14     ...
15 }
```

Listing 7.3: Ausführliche Funktionsbeschreibung in der C-Datei

Die Unterscheidung zwischen Kurzkommentar und ausführlichem Kommentar kann dahingehend genutzt werden, dass in der H-Datei nur die Kurzkommentare hinterlegt werden, um die H-Datei selbst als Schnell-Referenz verwenden zu können, während detaillierte Informationen bei der Implementierung selbst zu finden sind. So wird Doxygen auch im Rahmen des Praktikums eingesetzt.

7.4.2 Kommentieren von Funktionen

Zur Dokumentation von Funktionen stehen besondere Tags für die Signatur und den Rückgabewert zu Verfügung. Diese Tags sollten im ausführlichen Kommentar verwendet werden.

- \param [Parametername] [Beschreibung] - Alle Parameter in der Parameterliste sollten so kommentiert werden. Hier sollte auch der gültige Definitionsbereich (falls dieser eingeschränkt ist) angegeben werden.
- \return [Beschreibung] - Wenn der Rückgabewert nicht „void“ ist, sollte mit Hilfe dieses Kommentars eine Beschreibung des Rückgabewertes gegeben werden. Dazu gehört ggf. auch eine Beschreibung des Verhaltens im Fehlerfall.

7.4.3 Kommentieren von Dateien

Der folgende Doxygen-Kommentar ist ein Beispiel für einen Dateikommentar.

7 Dokumentation mit Doxygen

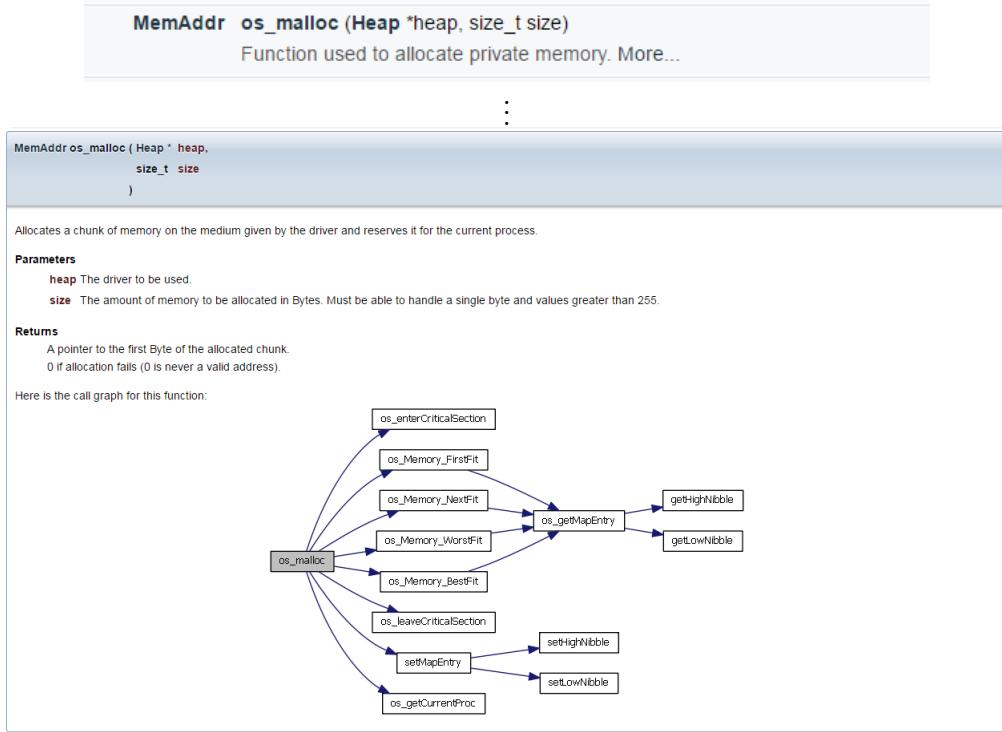


Abbildung 7.5: Die aus dem Beispielcode erzeugte Doxygen-Dokumentation

```

1  /*! \file
2   * \brief Data structures to be used project wide.
3   *
4   * Simply a set of structs for diverse purposes.
5   *
6   * \author Donald Knuth
7   * \author Alan Turing
8   * \author Lehrstuhl für Informatik 11 - RWTH Aachen
9   * \date 2007
10  * \version beta
11  * \ingroup module_01
12 */

```

Listing 7.4: Doxygen-Kommentare am Dateianfang

Diese Tags haben folgende Bedeutung:

- \file - Kündigt an, dass der folgende Teil des Kommentars ein Dateikommentar ist.
- \author [Name] - Jeweils ein Autor der Datei.

- \date [Zeitangabe] - Eine Zeitangabe zur Einordnung des Entstehungszeitpunkts.
- \version [Versionsnummer] - Die aktuelle Versionsnummer.

Dateikommentare sollten nur für Header-Dateien angelegt werden, Doxygen fasst jeweils die gleichnamigen .h- und .c-Dateien in einer Dokumentation zusammen.

7.4.4 Spezielle Tags

Folgende Tags können nach eigenem Ermessen benutzt werden, um die Dokumentation übersichtlicher zu strukturieren:

- \brief [Kurzkommentar] - Hiermit kann man einen Kurzkommentar in einen ausführlichen Kommentar einbetten.
- \n - Fügt einen erzwungenen Zeilenwechsel ein (Dieser Tag wird in den Fließtext integriert, anstatt am Zeilenanfang zu stehen).
- \defgroup [Gruppensymbol] [Gruppenname] - Definiert eine Gruppe („Module“ in der erzeugten Doxygen-Dokumentation), zu der beliebige andere Objekte hinzugefügt werden können. Dies dient im Wesentlichen der besseren Organisation der Dokumentation, da hier zusammengehörige Teile des Projekts gruppiert werden können.
- \ingroup [Gruppensymbol] - Fügt das Objekt einer mit \defgroup erstellten Gruppe von Dokumenten hinzu.
- \sa [Object 1], [Object 2], ..., [Object n] - (see also) Fügt Verweise zu den angegebenen Objekten hinzu. Ein Objekt kann in diesem Zusammenhang eine Funktion, eine Variable, eine Datei oder sogar eine URL sein.
- \mainpage - Markiert den aktuellen Kommentar als Text, der auf der Startseite der Dokumentation erscheinen soll.
- \internal - Markiert das aktuelle Objekt als ein internes Objekt, auf das nicht öffentlich zugegriffen werden soll.

8 Weiterführende Literatur

In diesem Dokument werden nur die wichtigsten Aspekte der jeweiligen Themen beleuchtet. In den folgenden Links werden weitere Informationsquellen aufgezeigt, die beim tiefergehenden Verständnis hilfreich sein können. Weitere Links sind auch im Moodle zu finden.

8.1 AVR-Mikrocontroller

- Ein sehr umfangreiches Tutorial, das auf die Verwendung mikrocontrollerspezifischer Funktionen unter C eingeht, findet sich auf
<http://www.mikrocontroller.net/articles/WinAVR>
- Die Website des Herstellers. Dort finden sich umfangreiche Datenblätter und Werkzeuge zu den hier verwendeten ATmega 644 Mikrocontrollern.
<http://www.atmel.com>

8.2 C-Programmierung

- <http://www.rn-wissen.de/index.php/C-Tutorial>
- <http://de.wikibooks.org/wiki/C-Programmierung>
- http://www.rn-wissen.de/index.php/Fallstricke_beи_der_C-Programmierung
- <http://www.cppreference.com/wiki/c/start>

8.3 Doxygen

Auf der Doxygen-Homepage finden sich ausführliche Tutorials und Codebeispiele. Zudem wird zusammen mit dem Doxygen-Setup ein sehr ausführliches und leicht verständliches Handbuch installiert. <http://www.doxygen.nl/>