RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

LEHRSTUHL
INFORMATIK 11
SOFTWARE FÜR
EINGEBETTETE
SYSTEME

Praktikum Systemprogrammierung

Begleitendes Dokument

Lehrstuhl für Informatik 11 - RWTH Aachen

8. November 2011

Inhaltsverzeichnis

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

Folgende Emailadresse ist für Kritik, Anregungen oder Verbesserungsvorschläge verfügbar: psp@embedded.rwth-aachen.de

1 Allgemeines

Dieses Dokument ist eine Hilfe für das Erarbeiten der Aufgaben des Praktikum Systemprogrammierung. In den folgenden Kapiteln werden nützliche Hinweise 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 Evalutationsboard platziert ist.
Kapitel 3, „Einführung in AVR Studio“, erläutert den Installationsprozess und die Verwendung der Entwicklungsumgebung des Mikrocontrollers.
Kapitel 4, „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 5, „Hinweise zum Debuggen“, erklärt den Vorgang des Debuggens und stellt die Tools vor, welche AVR Studio bereitstellt um den Programmierer bei der Fehlersuche zu unterstützen.
Kapitel 6, „Dokumentation mit Doxygen“, stellt das Programm Doxygen vor, welches direkt aus dem Quellcode Dokumentationen erstellen kann.

1.2 Praktikumsrichtlinien

Zum Bestehen des Praktikums ist die erfolgreiche Teilnahme an allen sechs Praktikumsversuchen sowie die Teilnahme an der Einführungsveranstaltung Voraussetzung. Zur erfolgreichen Teilnahme müssen die Hausaufgabe und die Präsenzaufgaben am Ende des Praktikumstermins entsprechend der Anforderungen lauffähig sein. Außerdem wird von Ihnen erwartet, dass Sie Ihren Arbeitsplatz aufgeräumt und sauber verlassen.

Es besteht die Möglichkeit bei triftigen Gründen maximal einen Versuch oder die Einführungsveranstaltung nicht zum vorgegebenen Zeitpunkt zu bestehen bzw. zu besuchen und diese(n) zu einem späteren Zeitpunkt nachzuholen. Die triftigen Gründe sind nachzuweisen (z.B. ärztliches Attest bei Erkrankung).
Die Aufgabenstellungen zu den jeweiligen Versuchen werden im Voraus im L^2P-Lernraum der RWTH zur Verfügung gestellt. Es ist Ihre Aufgabe, diese bis zum Versuchstermin zu bearbeiten und eine kurze Präsentation (ca. 5-7 Minuten / 3-5 Folien als PDF - andere Formate nur nach Absprache / maximal 8 MB Dateigröße) vorzubereiten. Die

Präsentation soll den Lösungsweg erläutern und auf mögliche Probleme eingehen. Die Präsentation darf dabei nicht nur aus Quelltext bestehen. Sie dürfen jedoch kurze Codebeispiele oder besser eine Darstellung als Pseudocode verwenden. Bei jedem Versuch wird ein Teil der Teams aufgefordert, ihre Präsentation in einem kurzen Vortrag vorzustellen. Daher werden einige Teams mehr als eine Präsentation halten.

Sollten die Hausaufgaben oder Präsentationen nur unzureichend vorbereitet sein, kann das Team nicht an diesem Praktikumsversuch teilnehmen und somit nicht zum vorgegebenen Zeitpunkt den Versuch bestehen. Dies kann zum Ausschluss vom Praktikum führen.

Die Bearbeitung der Aufgabenstellung kann zwischen den Teampartnern zu gleichen Teilen aufgeteilt werden, jedoch müssen alle Studierende innerhalb des Teams die komplette Lösung verstehen und erklären können. Die Aufgabenstellung ist von den beiden Studierenden eines Teams selbstständig zu lösen. Zuwiderhandlungen werden als Täuschungsversuch gewertet, der zum Ausschluss vom Praktikum führt. Es ist ausdrücklich verboten Code ganz oder in Teilen von einem anderen Team zu übernehmen.

Die Lösungen der Aufgabenstellungen und die Präsentation sind mittels der dafür eingerichteten Website spätestens 24 Stunden vor Praktikumsbeginn auf den Server des Lehrstuhls zu übertragen. Die eingesandten Unterlagen werden Ihnen zu Beginn jedes Versuches zur Verfügung gestellt. Der Code kann gesammelt als Archiv eingesandt werden. Diese Archive dürfen nicht passwortgeschützt sein. Die Präsentation soll getrennt (also nicht in einem Archiv) im PDF Format übertragen werden. Während der Versuche ist ausschließlich auf dem Netzlaufwerk und nicht auf der lokalen Festplatte zu arbeiten. Sollten mehrere Versionsstände Ihrer Software existieren, so ist die endgültige Version eindeutig zu kennzeichnen.

Ihre Unterlagen werden auf Plagiate kontrolliert. Sie erklären sich einverstanden, dass dies auch mit Hilfe einer Plagiatsuchmaschine durchgeführt werden kann. Dabei wird Ihre Arbeit auf einem Server gespeichert. Im Fall, dass zwei oder mehr Teams den finalen Code nicht selbstständig erstellt haben wird dies als Täuschungsversuch gewertet und alle beteiligten Teams vom Praktikum ausgeschlossen.

Eingeschickte Dateien gelten als Abgabe. Daher sollen nur endgültige Versionen hochgeladen werden. Die abgegebenen Hausaufgaben müssen frei von Fehlern und Warnungen kompilieren. Dabei muss ein von dem im Praktikum verwendete Kompilierer lesbares Format verwendet werden. Einzelne eventuell noch bekannte Fehler müssen dokumentiert sein und damit sichtbar gemacht worden sind. Die Funktionalität muss in den dafür vorgesehen Methodenrümpfen implementiert werden.

Sie werden gebeten pünktlich zu Ihren jeweiligen Praktikumsterminen zu erscheinen. Sollten Sie zu spät zu einem Termin erscheinen, kann dies dazu führen, dass Sie nicht an diesem Termin teilnehmen können. Eine Abmeldung vom Praktikum ist bis maximal 21 Tage nach der Vorbesprechung möglich (siehe BPO2010). Bei einer späteren Abmeldung wird das Praktikum als nicht bestanden gewertet.
Des Weiteren führt jeder Versuch, die vom Lehrstuhl zur Verfügung gestellten Geräte oder Infrastruktur zu kompromittieren oder zu beschädigen zum Ausschluss vom Praktikum. Wechselmedien (z.B. USB Medien) werden grundsätzlich bei Benutzung in dem zur Verfügung gestellten PC auf Schadprogramme untersucht. Falls eine Infektion festge-

stellt wird, kann das Schutzprogramm diese Datei ohne Ankündigung löschen. Darüber hinaus werden alle Änderungen nach dem Neustart des PCs verworfen. Daher ist während der Praktikumsversuche immer auf dem zur Verfügung gestellten Netzlaufwerk zu arbeiten.

Außerdem gelten die umseitigen Sicherheitsbestimmungen, sowie die Netzordnung der RWTH Aachen.

1.3 Allgemeine Sicherheitsbestimmungen

- Den Anweisungen der Betreuer ist Folge zu leisten.
- Ein unbefugter Aufenthalt im Praktikumsraum ist verboten.
- Arbeiten Sie ausschließlich mit Niederspannungen < 30V.
- Stecken Sie NICHTS in die Steckdose!
- Das anschließen eigener Hardware (z.B. Laptop oder eigener Mikrocontroller) an die Infrastruktur im Praktikumsraum ist nicht gestattet.
- Verwenden Sie ausschließlich die Ihnen beschriebenen und bekannten Geräte.
- Trennen Sie im Fehlerfall (Bauteile werden heiß, Geruchsentwicklung,...) den Versuchsaufbau durch Ausschalten des Netzteils.
- Falls Gefahr für Menschen besteht (elektrischer Schlag), schalten Sie unmittelbar den gesamten Arbeitsplatz durch Ausschalten der Steckerleiste spannungsfrei! (PCs werden dabei mit ausgeschaltet)
- Bauteile können im Fehlerfall heiß werden! Dies ist zu beachten, um Verbrennungen zu vermeiden!
- Wird festgestellt, dass Einrichtungen oder Hilfsmittel sicherheitstechnisch nicht einwandfrei sind, so ist dieser Mangel unverzüglich dem zuständigen Mitarbeiter zu melden. Die Geräte oder Anlagen sind nicht weiter zu verwenden und der Benutzung durch andere Personen zu entziehen sowie auf Gefahren hinzuweisen.
- Änderungen am Aufbau elektrischer Schaltungen und Systeme müssen im spannungslosen Zustand vorgenommen werden. Soweit nicht ausdrücklich anderweitig geregelt, müssen unter Spannung stehende Schaltungen beaufsichtigt bleiben.
- Die Rechner, Geräte und Werkzeuge sind sorgfältig zu behandeln. Beschädigungen an diesen müssen unverzüglich dem zuständigen Mitarbeiter gemeldet werden. Für grob fahrlässig oder vorsätzlich verursachte Schäden ist der Benutzer voll ersatzpflichtig.
- Die Praktikumsräume und deren Einrichtungen sind stets in Ordnung zu halten; insbesondere ist nach Ende des Versuchs der Praktikumsplatz aufzuräumen.

- In den Praktikumsräumen sind Essen, Trinken und Rauchen nicht gestattet.
- Vorhandene Warn- und Hinweisschilder sind generell zu beachten.
- Die Kontakte des Boards sollen so wenig wie möglich berührt werden, um eine Beschädigung durch elektrostatische Entladung zu vermeiden.
- Vor dem Verbinden des Boards mit der Spannungsversorgung sind alle Jumper und Drahtverbindungen auf korrekte Konfiguration zu überprüfen. Achten Sie auf die Polarität der Spannungsversorgung.
- Es sind nur die in der jeweiligen Aufgabenstellung beschriebenen Ports/Pins zu benutzen.
- Die Zugänge zu den Not- oder Ausschaltern, Verteileranlagen, Feuerlöscheinrichtungen, sowie Türen und Durchgängen sind freizuhalten.
- Es ist verboten, fremde Versuchsaufbauten oder Schaltungen zu berühren oder zu verändern. Es darf kein Gerät von einem anderen Versuchsplatz entfernt werden.
- Es ist verboten, die Konfiguration von Praktikumsrechnern zu verändern, Programme zu kopieren, oder Programme zu installieren.
- Die Notrufnummer lautet 113.

1.4 Haftung

Im L^{2}P-Lernraum wird das Dokument *Bedienungsanleitungen und Sicherheitshinweise* zur Verfügung gestellt, welches den korrekten Umgang mit der zur Verfügung gestellten Hardware erläutert.

Der Benutzer haftet für unsachgemäße Bedienung, mutwillige oder grob fahrlässige Zerstörung und Verlust. Der Lehrstuhl haftet nicht für persönliche oder materielle Schäden, die nachweislich durch fahrlässiges Verhalten oder Vorsatz entstanden sind. Ersatzansprüche gegen den Lehrstuhl sind ausgeschlossen. Einrichtung und Gegenstände sind schonend zu behandeln. Beschädigungen sind, unabhängig davon, ob sie vorgefunden oder selbst verursacht worden sind, sofort dem zuständigen Mitarbeiter zu melden.

1.5 Programmierrichtlinien

Um die Lesbarkeit des Codes und die Fehlersuche während des Versuches zu minimieren, muss sich an folgende Richtlinien gehalten werden. Am Ende jedes Punktes findet sich ein Verweise 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 `constants.h`, `defines.h`, `struct.h`, `enums.h`, `typedef.h`, um das Projekt übersichtlich zu halten. Siehe Kapitel 4.1.1.

- Wiederkehrende Funktionen sollen ausgelagert werden, um Codeduplikationen zu vermeiden. Siehe Kapitel 4.2.2.
- Kommentare sollen kurz und aussagekräftig gehalten werden. Siehe Kapitel 4.2.1.

  Kommentiert werden müssen mindestens:

  – Funktionen
  – wichtige Variablen, Konstanten, Defines und Datenstrukturen

  Zu Dokumentation siehe auch Kapitel 6: „Dokumentation mit Doxygen“.
- Variablen sollen mit möglichst geringem Gültigkeitsbereich deklariert werden. Gloable Variablen sollen vermieden werden. Siehe Kapitel 4.1.5.
- Globale Variablen sollen nur mit Hilfe von get()- und set()-Methoden manipuliert werden.
- Für eigene Aufzählungstypen sollen `enums` verwendet werden. Siehe Kapitel 4.1.11.
- Als Kontrollstruktur bei aufzählbaren Datentypen (`enums`) sollen Switch-cases verwendet werden. Siehe Kapitel 4.31.
- Bitmasken sollen durch gezieltes bitshifting erzeugt werden unter Zuhilfename der durch den avr-gcc bereitgestellten *defines*. Siehe Kapitel 4.2.3.
- Mit Zeigern sollte vorsichtig und überlegt umgegangen werden. Bei einer Zeigerdefinitionen soll das Sternchen an den Variablennamen und nicht an den Datentypen geschrieben werden. Siehe Kapitel 4.1.7.
- Aussagekräftige *defines* für feste Werte erhöhen die Übersicht des Programmcodes. Dabei muss auf ausreichende Klammerung geachtet werden, um unerwünschtes Verhalten zu vermeiden. Siehe Kapitel 4.1.2
- Variablen müssen aussagekräftig benannt werden. Siehe Kapitel 4.2.3

2 Hardware

In diesem Kapitel wird die im Praktikum Systemprogrammierung verwendete Hardware vorgestellt. Dabei handelt es sich um einen ATmega 644 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 Datenblatt eingesehen werden, auf welches im L^{2}P-Lernraum 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. Dazu müssen Bitmasken verwendet werden. Siehe Kapitel 4.1.16. 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-Wiederstände verwendet. Pull-up 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 Evaluationsboard 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 verbunden wird. Dadurch ergibt sich die Situation, dass bei Druck 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.

2.1.2 Interrupts

Es gibt externe und interne Ereignisse, die einen Interrupt auslösen können. Einen 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 Interrupts 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 verschieden 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:

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

Je größer der Prescaler ist, desto langsamer zählt der jeweilige Timer. Der Wert des Prescalers kann jedoch nicht frei programmiert werden. Durch Hardware bedingt sind nur einige bestimmte Zweierpotenzen als Werte möglich. Alle Timer des ATmega 644 unterstützen die Prescaler-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` durchnummeriert, und können beliebig gelesen und geschrieben 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 Evaluierungsboard

Das Evaluierungsboard 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.

2.2.1 Funktionen des Evaluierungsboards

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

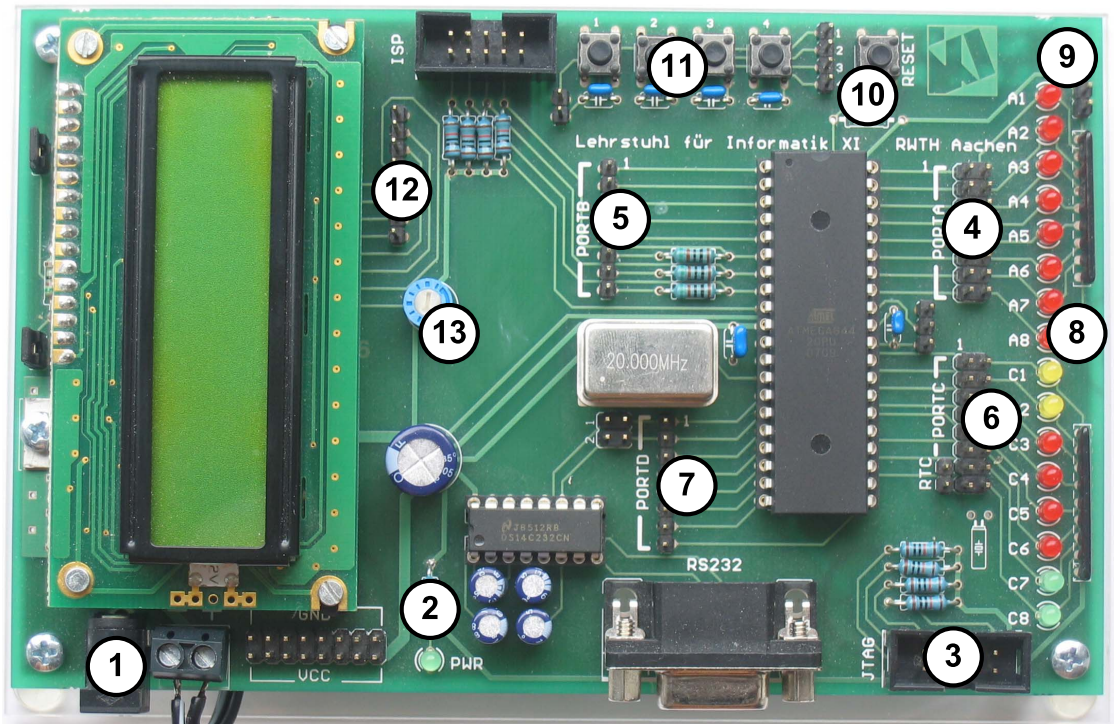


Abbildung 2.1: Das im Praktikum verwendete Evaluierungsboard

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 Evaluierungsboard 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 L^2P-Lernraum ein fertiger Treiber in Form einer Header-Datei samt zugehöriger Implementierung (`ldc.c`, `lcd.h` und `lcd_customchars.h`) vorhanden. Die wichtigsten Funktionen werden hier kurz vorgestellt:

Initialisierung

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

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 übergebene Position. Gültige Werte für `row` sind {1,2}. Gültige Werte für `column` sind $\{1 \ldots 16\}$.

Löschen des Displays

- `lcd_clear(void)`
  Löscht den Displayinhalt und setzt den Cursor wieder auf die Anfangsposition zurück.

- `lcd_erase(unsigned char 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 0-16, die als "0"-"F" ausgegeben werden.

- `lcd_writeHexByte(uint8_t number)`
  Schreibt die Eingabe im hexadezimalen Format auf das Display.

- `lcd_writeHexWord(uint16_t number)`
  Schreibt die Eingabe mit führenden Nullen im hexadezimalen Format auf das Display.

- `lcd_writeHex(uint16_t number)`
  Schreibt die Eingabe 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 prog_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 $\{0 \ldots 100\}$). Vor Aufruf der Funktion muss der Cursor an den Anfang einer Zeile gesetzt werden.

2.2.3 Relevante Headerdateien von AVR-GCC

Neben den vom Lehrstuhl zur Verfügung gestellten Headerdateien werden auch einige von AVR-GCC genutzt. Die wichtigsten für dieses Praktikum sind:

- `avr/io.h`: Enthält eine große Zahl an `#define`-Anweisungen, die zum Beispiel Konstanten wie `PORTA`, `DDRB`, `MCUCR`, etc. zu den mikrocontrollerspezifischen Werten auflösen.
- `avr/interrupt.h`: Enthält Definitionen, die das einfache Anlegen von Interrupt Service Routinen ermöglichen. So wird hier z. B. das Makro `ISR()` definiert, mit dem Funktionen als ISR implementiert werden. Auch `sei()` und `cli()` zum globalen An- bzw. Abschalten der Interrupts werden hier definiert.
- `avr/pgmspace.h`: Enthält Funktionen, die den Zugriff auf den Programmspeicher erlauben. Während der Laufzeit ist nur unter speziellen Bedingungen ein Schreibzugriff auf den Programmspeicher möglich, standardmäßig ist ein derartiger Zugriff jedoch aus Sicherheitsgründen verboten. Es ist aber möglich, Konstanten wie z. B. Strings als `prog_char[]` anstelle des normalen `char[]` zu definieren und diese damit in den (wesentlich größeren) Programmspeicher auszulagern, anstatt sie im Arbeitsspeicher zu halten. Genauere Informationen zur Verwendung von `prog_char` können in Kapitel 4.1.10 nachgeschlagen werden.

3 Einführung in AVR Studio

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

3.1 Verwendete Software

In den Praktikumsversuchen kommen zwei verschiedene Programme zum Einsatz:

- **AVR Studio**
  AVR Studio ist eine komplette Entwicklungsumgebung für AVR-Mikrocontroller, d.h. es stellt einen Code-Editor, einen Compiler und einen Debugger zur Verfügung. AVR Studio kann auf http://www.atmel.com kostenlos heruntergeladen werden. Auf den Arbeitsplatzrechnern ist die Version 4.19 installiert. Die Installation der Software auf Ihrem Heimrechner ist ebenfalls notwendig, da der entwickelte Code vor Beginn des Versuchs gründlich getestet werden sollte.

- **AVR-Toolchain** Zusätzlich wird die AVR-GCC Toolchain in der Version 3.3.0 von Atmel verwendet, welche AVR Studio um einen C-Compiler erweitert und es so ermöglicht, den Mikrocontroller in C zu programmieren. Die Toolchain kann ebenfalls von http://www.atmel.com bezogen werden. Diese Software ist nötig, da AVR Studio zunächst nur Assembler verarbeiten kann. Alternativ kann die AVR-GCC Bibliothek WINAVR verwendet werden, welche unter Umständen schneller arbeitet als die AVR-GCC Toolchain. Eine genaue Beschreibung zur Installation von WINAVR kann in Abschnitt 3.3.4 gefunden werden.

Toolchain zur Entwicklung von ATmega-Programmen

Die Entwicklung von Programmen auf einem ATmega-Mikrocontroller in der Programmiersprache C läuft folgendermaßen ab:

1. Erstellen des Quellcodes (*.c* und *.h* - Dateien) mit AVR Studio
2. Kompilieren der Dateien zu Objektdateien in Maschinencode (*.o* - Dateien) mit der AVR-GCC Bibliothek für den AVR Studio-Compiler
3. Linken der Maschinencodedateien zu Programmdateien (*.hex* oder *.elf* - Dateien) mit dem Linker
4. Laden der Programmdateien auf den Mikrocontroller mit dem AVR Studio über die ISP- oder JTAG-Schnittstelle

(5.) Debuggen des laufenden Programms über die JTAG-Schnittstelle mit AVR Studio

Es folgen einige Hinweise zu Teilen der Toolchain:

AVR-GCC Die AVR-GCC Bibliothek für den AVR Studio-Compiler ermöglicht es, den Mikrocontroller unter C zu programmieren und sorgt dafür, dass die Register des Mikrocontrollers im Quellcode direkt angesprochen werden können. Es ist zu beachten, dass einige Register nicht unter demselben Namen wie im Datenblatt des Mikrocontrollers ATmega 644 verfügbar sind. Beim Kompilieren wird der Code eingelesen und seine Struktur erkannt, um Maschinencode zu generieren. Je nach gewählter Einstellung wird die Struktur des Programmes vor dem Erstellen des Maschinencodes aus Optimierungszwecken automatisch verändert. In Kapitel 3.5 wird das Vorgehen des Compilers erklärt. Dies kann zu unerwartetem Verhalten führen, wie in Kapitel 5.4.2, „Hinweise zum Debuggen“, genauer ausgeführt wird.

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 sehr gebräuchlich sind: Die *In System Programming* (ISP) und die *Joint Test Action Group*-Schnittstelle (JTAG). Mit beiden Schnittstellen ist es möglich den Mikrocontroller zu programmieren und zu konfigurieren (Fuses setzen). Die JTAG-Schnittstelle ist zusätzlich in der Lage alle Speicherbereiche des Mikrocontrollers während des Betriebs auszulesen und sowohl die Speicherbereiche als auch den Programmablauf zu manipulieren. Dies ermöglicht das Debuggen des Mikrocontrollers. Zur Verwendung der *JTAG-Schnittstelle* wird ein zusätzliches Gerät (ein so genannter JTAG Programmer) benötigt, welches die Kommunikation zwischen dem Computer und dem Mikrocontroller ermöglicht.

3.2 Verwendung des AVR Studios

In diesem Kapitel wird AVR Studio vorgestellt. Zunächst werden die Elemente der IDE erläutert, im Anschluss wird der Prozessablauf der Projekterstellung und ausführung erläutert. Abschliessend werden die Debugfunktionen von AVR Studio vorgestellt.

3.3 Aufbau der IDE

Der Aufbau der Entwicklungsumgebung (siehe Abbildung 3.1) ist an andere IDEs (Integrated Development Environment) angelehnt. Neben den üblichen Menü- und Symbolleisten finden Sie folgende Arbeitsbereiche:

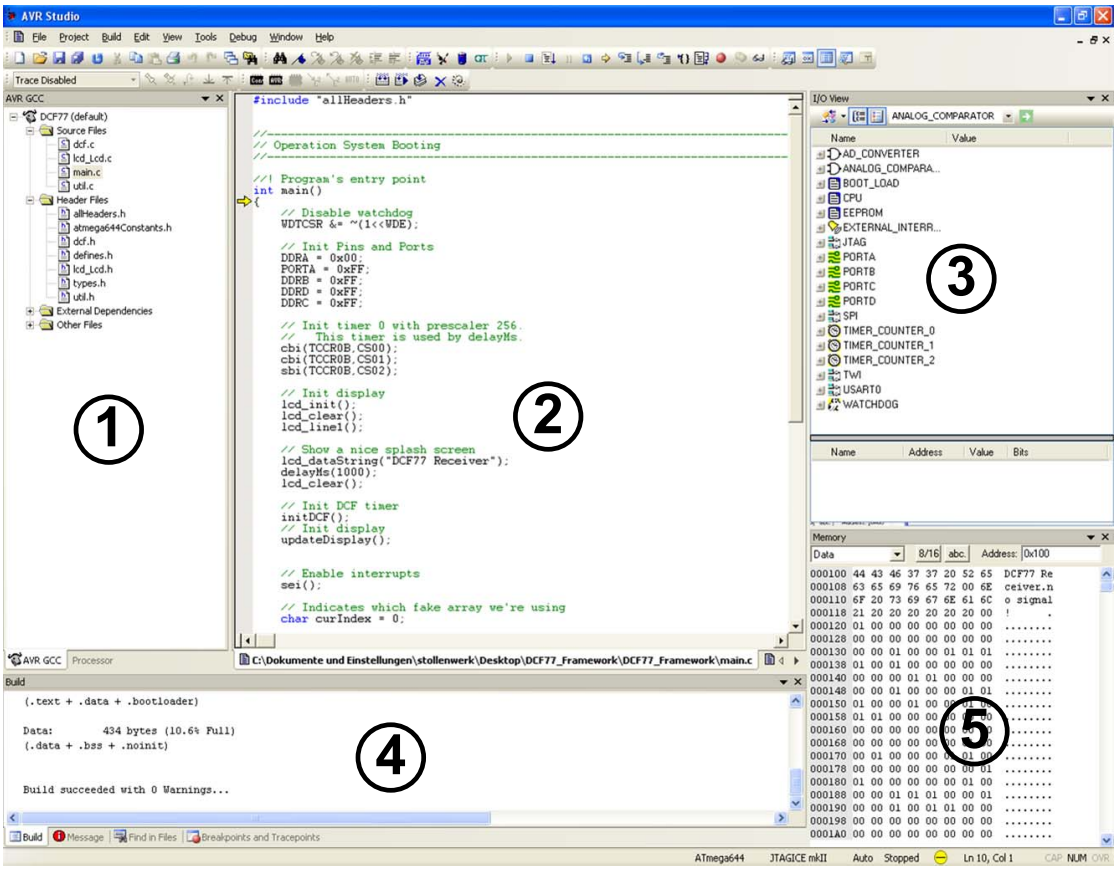


Abbildung 3.1: AVR Studio

1. *Workspace Tabs*: Da der Programmquellcode über mehrere Dateien modularisiert sein kann, wird nicht nur mit einer Datei pro Dokument gearbeitet, sondern mit einem sog. Projekt. In diesem Navigator sind alle Dateien zu finden, die zu dem aktuellen Projekt gehören. Daher werden bei der Arbeit mit einer IDE selten einzelne Dateien sondern meistens gleich ganze Projektverzeichnisse geöffnet. Welche Dateien Teil eines Projekts sind, speichert AVR Studio in Projektdateien mit der Endung „aps“. Am unteren Rand gibt es weitere Reiter, die im Abschnitt über Debuggen erläutert werden.

2. *Editor*: In diesem Bereich wird der Quellcode editiert. Am unteren Rand finden sich Reiter, mit deren Hilfe zwischen mehreren geöffneten Dateien gewechselt werden kann. Mit einem Rechtsklick in eine Codezeile kann per Kontextmenü ein Haltepunkt gesetzt werden.

3. *IO-View*: Hier wird der Zustand des Mikrocontrollers detailliert dargestellt (sofern der Mikrocontroller im Debug-Modus betrieben wird). Es können z. B. die Werte der Timer, die Zustände der Ein- und Ausgangspins sowie die Inhalte wichtiger

Register angezeigt werden.

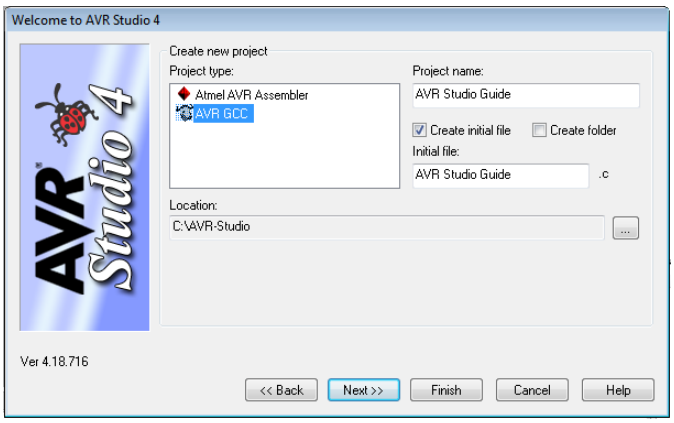
4. *Output Tabs*: In diesem Fenster können verschiedene Systemmeldungen eingesehen werden. Dies betrifft vor allem Meldungen des Compilers, der hier über Warnungen und Fehler im Quellcode informiert. Es kann aber auch ein Reiter mit Suchfunktion geöffnet werden oder ein Reiter welcher Informationen über vorhandene Haltepunkte (Breakpoints) enthält.

5. *Memory*: Hier können die Inhalte aller Speicher (Flash, SRAM und EEPROM) des Mikrocontrollers abgelesen werden.

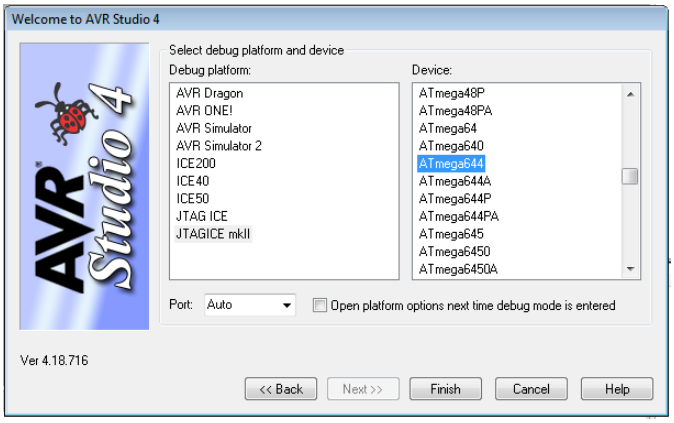
Optionen für die Anpassung des Layouts, weil z. B. ein Fenster fehlt oder ein anderes Fenster nicht benötigt wird, können über den Menüpunkt „View“ in der Menüzeile gefunden werden.

3.3.1 Anlegen eines neuen Projektes

Ein neues Projekt kann unmittelbar nach dem Programmstart angelegt werden. Dabei fragt AVR Studio nach dem Namen des neuen Projektes, der direkt als Verzeichnisname für die Projektdateien und als Name der Projektdatei vorgeschlagen wird. Sowohl „Create initial file“ als auch „Create Folder“ sollten eingeschaltet sein. Weiterhin sollte darauf geachtet werden, dass unter „Project type“ der Eintrag „AVR GCC“ ausgewählt ist.



Auf der nächsten Seite wird der zu programmierende Mikrocontroller und die Umgebung zum Debugging ausgewählt. Als „Device“ sollte immer „ATmega 644“ gewählt werden. Während des des Praktikums ist hier „JTAG ICE mkII“ einzustellen.



Zur Umstellung des Projektes auf ein anderes *Device*, können diese Einstellung jederzeit über „Debug -> Select Platform and Device“ im Hauptmenü geändert werden.

3.3.2 Auswahl der Optimierungsstufe

Der genutzte Compiler (AVR-GCC) hat die Möglichkeit den Code bei der Übersetzung aus der Sprache C in Maschinencode zu optimieren. Einige Beispiele zu den Details dieser Optimierungsmaßnahmen werden in Kapitel 3.5 vorgestellt und erläutert. Im Grundsatz kann festgehalten werden, dass die Optimierung dazu führt, dass der erzeugte Maschinencode kleiner wird und das Programm schneller abgearbeitet wird.

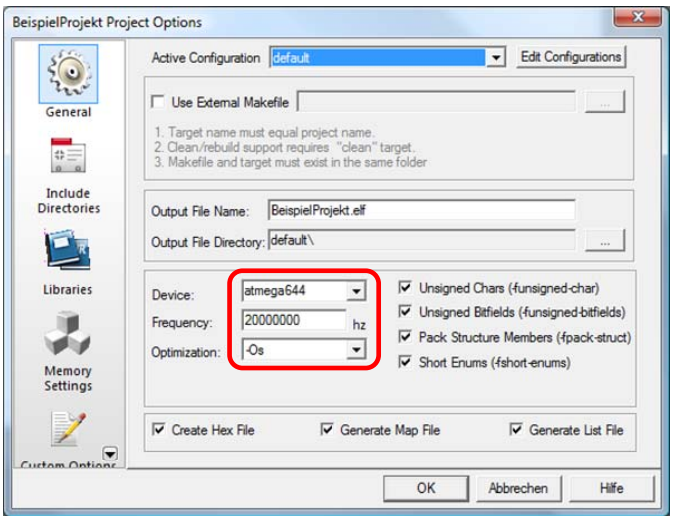
Die Optimierungsstufe lässt sich im AVR-GCC Konfigurationsmenu einstellen, welches sich über den Button „Edit Current Configuration Options“ aus der Haupt-Menüleiste (siehe Markierung) aufrufen lässt.

Hier können Sie die gewünschte Optimierungsstufe einstellen. Dabei steht

- „-O0“ für keine Optimierung
- „-Os“ für Größenoptimierung
- „-O1“ für leichte Geschwindigkeitsoptimierung
- „-O2“ für mittlere Geschwindigkeitsoptimierung
- „-O3“ für starke Geschwindigkeitsoptimierung

Meist empfiehlt sich der Wert „-O2“, da dieser einen guten Kompromiss aus Größe und Geschwindigkeit bietet, ohne dass sich das Aussehen des erzeugten Assembler-Code zu

stark von Ihrem C-Code unterscheidet. Denn je höher die Optimierungsstufe gewählt wird, desto mehr Freiheiten hat der Compiler das Aussehen des Codes zu verändern, was unter Umständen das Debuggen erschweren kann. Nähere Informationen bezüglich der Umgestaltung des Codes, durch die Optimierung des Compilers, können in Kapitel 3.5 gefunden werden.



Zusätzlich sollte in diesem Dialog der verwendete Mikrocontroller („Device“) und die Quarzfrequenz („Frequency“) auf „ATmega 644“ bzw. „20000000“ Hz eingestellt werden.

3.3.3 Ein Programm kompilieren und starten

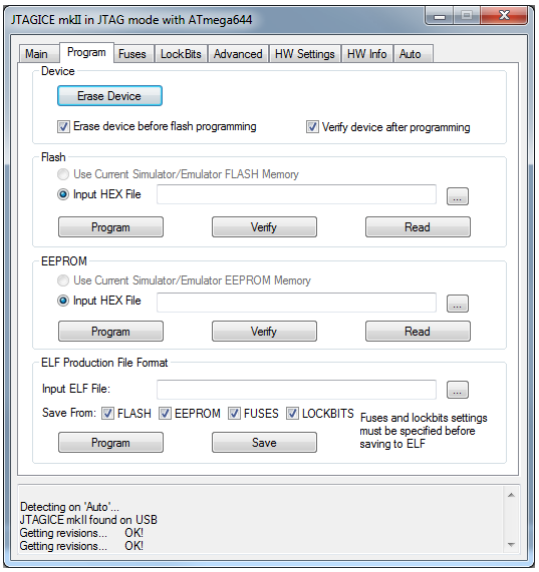
Nachdem der Quellcode geschrieben wurde, kann das Programm kompiliert und auf den Mikrocontroller geladen werden. Dies wird durch die Befehle „Build“ (F7) und „Start Debugging“ bzw. „Build and Run" (Strg + F7) realisiert. Dabei wird das Programm im Debug-Modus gestartet und kann zur Laufzeit über den Befehl „Break“ (Strg + F5) unterbrochen werden. Falls es bei der Überprüfung des Quellcodes zu Fehlern gekommen ist, unterbricht der Compiler mit einer Fehlermeldung. Ein Doppelklick auf die entsprechende Zeile führt zu der entsprechenden Stelle im Quellcode.

Ist das Programm syntaktisch korrekt, kompiliert und geladen, kann die Ausführung mit „Run" (F5) begonnen werden. Um das Verhalten des Mikrocontrollers zur Laufzeit zu beobachten, gibt es zahlreiche Funktionen. Zum einen die sog. Watch-Listen („View -> Watch“), zum anderen die bereits erwähnte „I/O View“. Im Debug-Modus kann man Variablen etc. nur einsehen bzw. ändern, wenn das Programm angehalten wurde. Dies geschieht entweder per Haltepunkt oder „Break “ (Strg+F5). Es kann vorkommen, dass ansonsten ein falscher Wert angezeigt wird.

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):

Im sich anschließend öffnenden Fenster können im Reiter „Program“ 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-File auf den Mikrocontroller zu flashen.



Die Fehlermeldung „Object file not found on expected location“ Die Fehlermeldung „GCC plug-in: Error: Object file not found on expected location *Ordner\Dateiname.elf*“, welche beim Kompilieren auftreten kann, kann mehrere mögliche Ursachen haben:

- Eine potentielle Ursache für diesen Fehler ist, dass in einer Header-Datei, die zum aktuellen Projekt gehört, eine globale Variable zugleich deklariert und initialisiert wird. Die Abhilfe in diesem Fall besteht darin, die Initialisierung in der zugehörigen `.c`-Datei durchzuführen und die globale Variable in der Headerdatei mit dem Schlüsselwort `extern` zu deklarieren.

- Eine weitere potentielle Ursache ist, dass der JTAG ICE mkII abgestürzt ist. Hier besteht die Lösung darin, AVR Studio zu schließen, den JTAG ICE mkII aus- und wieder einzuschalten, und dann AVR Studio wieder zu öffnen. Dabei sollte exakt diese Reihenfolge eingehalten werden.

- Die letzte potentielle Ursache ist ein Bug in AVR Studio. Ein Workaround besteht darin, im Menü „Build“ den Menüeintrag „Rebuild All“ auszuwählen. Wenn dies nicht funktioniert, muss ein neues Projekt angelegt und die alten Quelldateien in dieses importiert werden.

Sollten alle oben genannten Hinweise keine Abhilfe schaffen, sollte das Problem im Forum des L^{2}P-Lernraum dieses Praktikums geschildert werden.

3.3.4 Die alternative Toolchain WINAVR

Alternativ zu der AVR-Toolchain kann die AVR-GCC Bibliothek WINAVR verwendet werden. WINAVR arbeitet unter Umständen schneller als die native AVR Toolchain, deshalb wird diese Bibliothek an dieser Stelle noch einmal erwähnt. Seit Version 4.19 von AVR Studio wird diese jedoch nicht mehr automatisch erkannt. WINAVR kann von der Webseite http://winavr.sourceforge.net/ heruntergeladen und installiert werden. Um die Bibliothek in einem Projekt von AVR Studio zu verwenden, sind einige kleine Änderungen an den Projekteinstellungen nötig, welche unter dem Menüpunkt *Project* → *Configuration Options* zu finden sind.

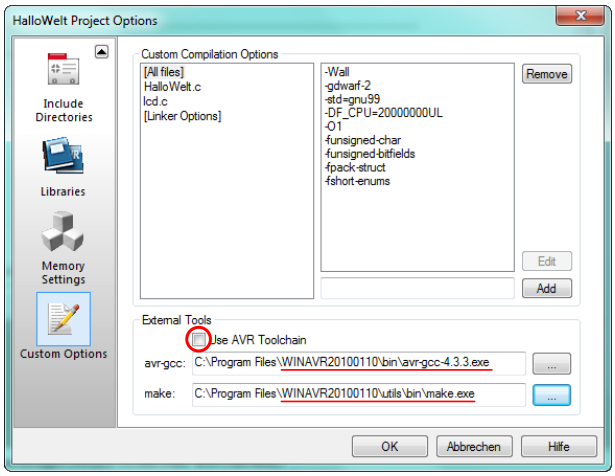


Abbildung 3.2: Custom Options in den Projekteinstellungen

Unter dem Reiter *Custom Options* finden sich Einstellungen, welche es ermöglichen WINAVR, statt der AVR-Toolchain von Atmel, zu verwenden. Abb. 3.2 zeigt das Fenster mit richtig konfigurierten Pfaden. Das Häkchen an der Einstellung *Use AVR Toolchain* muss entfernt werden, sowie die Pfade abhängig von dem Installationspfad von WINAVR gesetzt werden. Die genauen Pfade können Abb. 3.2 entnommen werden.

3.3.5 Debugging

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

- Das Setzen von Breakpoints - siehe Kapitel 5.2.1.
- Das Überwachen von Variablen - siehe Kapitel 5.2.3.
- Das Anzeigen von Registereinstellungen - siehe Kapitel 5.2.3.

In Kapitel 5, „Hinweise zum Debuggen“, wird ausführlicher auf diese und andere Methoden eingegangen. Es empfiehlt sich diese Kapitel zu lesen um den Prozess des Debuggen schneller und effizienter durchführen zu können.

ACHTUNG

Durch Codeoptimierungen seitens des Compilers kann das Debuggen erschwert werden! Beachten Sie dazu Kapitel 3.5. Beachten Sie außerdem, dass die Aktivierung des Debug-Modus einige Einstellungen des Mikrocontrollers (z. B. die LockBits) zurücksetzen.

3.4 Benutzung des Testpool

Der Testpool des Praktikum Systemprogrammierung besteht aus 15 per Windows Remotedesktopverbindung aus dem RWTH-Netz erreichbaren Computern. Diese sind mit der gleichen Peripherie ausgestattet, welche auch während der Präsenzzeit des Praktikums Verwendung findet. Das Verhalten der Mikrocontroller kann sowohl durch eine Kamera beobachtet werden, als auch durch Tasterbestätigung beeinflusst werden. Im Folgenden wird erläutert, wie eine Verbindung hergestellt werden kann und was bei der Benutzung zu beachten ist.

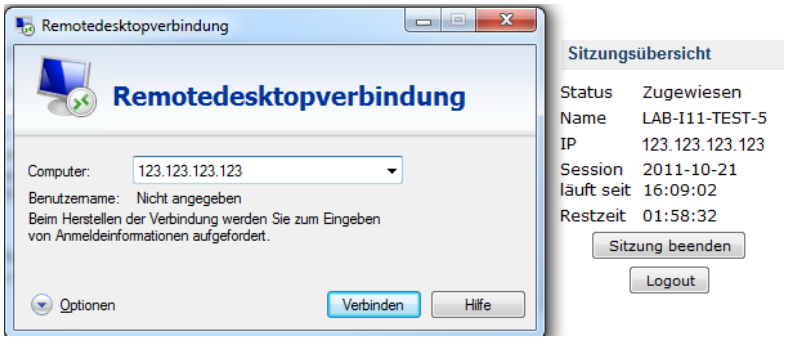


Abbildung 3.3: Remotedesktopverbindung unter Windows

Verbindung herstellen Um eine Verbindung herzustellen muss man sich zunächst im RWTH-Netz befinden. Von außerhalb kann man mithilfe einer VPN Verbindung einwählen. Informationen dazu stellt das Rechenzentrum bereit: Installation VPN.

Danach muss auf der Lehrstuhlseite https://psp.embedded.rwth-aachen.de/pool eine Sitzung für einen Rechner des Testpools beantragt werden. Dazu wird der ausgegebene Benutzeraccount des Praktikums mit vorangestelltem EMBEDDED\ verwendet. Bei Benutzername PSP-1-1, also EMBEDDED\ PSP-1-1. Sitzungen werden nach 2 Stunden automatisch getrennt, um das Blockieren von Rechnern durch einzelne Gruppen zu vermeiden. Da die Benutzungsdauer in die Priorität in der Warteschlange eingeht, sollten die Sitzungen am Ende immer manuell beendet werden, statt sie einfach auslaufen zu lassen.

Sobald eine Sitzung beantragt wurde, wird die Position in der Warteschleife angezeigt. Position 1 bedeutet dabei, dass gerade ein Rechner vorbereitet wird. Sobald die Vorbereitung abgeschlossen ist, wird die IP des vorbereiteten Rechners angezeigt. Zur Verbindung kann unter Windows das Programm Remotedesktopverbindung, wie in 3.3 dargestellt, verwendet werden. Unter Linux gibt es unter anderem das Programm "'Terminal-Server-Client"' das ebenfalls funktioniert. Die beim verbinden eventuell erscheinende Warnung sollte gelesen und bestätigt werden.

Auf Remotedesktop arbeiten Nach erfolgter Anmeldung wird der Remotedesktop angezeigt. Nachdem einloggen mit Namen und Passwort kann auf diesem wie auf einem Rechner im Praktikumsraum gearbeitet werden. Um den Mikrocontroller sehen und mit ihm interagieren zu können, ist es nötig, die `Atmega Remote` Verknüpfung auf dem Desktop zu starten. In 3.4 ist das Programmfenster dargestellt. Die Taster, die das Video Bild überlagern, können per Maus bedient werden. Hierbei lässt ein Linksklick ein kurzes Drücken des Tasters aus und ein Rechtsklick dient dazu, den Taster gedrückt zu halten. Im oberen Teil des Programmfensters befindet sich ein Button mit welchem sich der JTAG neustarten lässt, dies ist nötig falls er von AVR Studio nicht korrekt erkannt wird.

Ein AVR-Projekt kann per Copy/Paste von dem lokalen PC auf den Testpool PC kopiert werden. Zu beachten ist, dass nur auf dem Netzlaufwerk gearbeitet werden sollte, um Datenverlusten vorzubeugen. Alle nicht dort befindlichen Dateien sind nach dem Ausloggen gelöscht.

3.5 Compiler Optimierungen

Der AVR-GCC-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 werden ein Teil der vom Compiler angewandten Optimierungen genauer dargestellt. Zum besseren Verständnis sind die hier dargestellten Techniken teilweise vereinfacht.

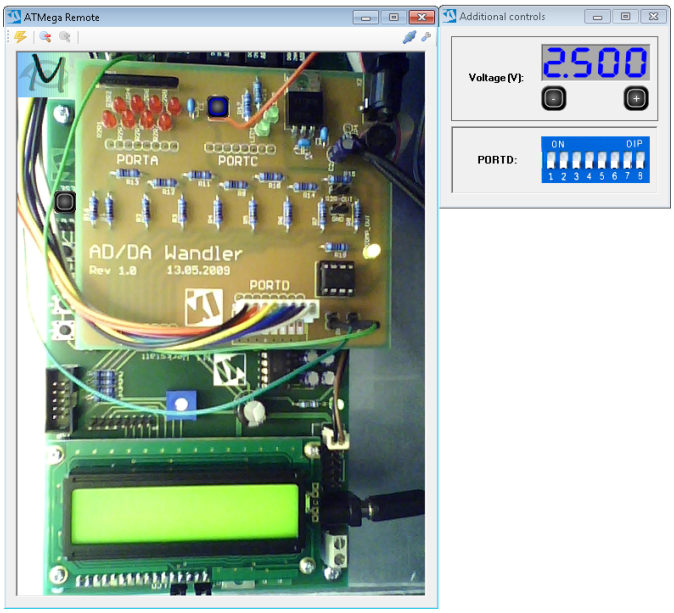


Abbildung 3.4: Das Programm Atmega Remote

Sprungvorhersage Gibt es im Programm bedingte Sprungbefehle (etwa bei einer if-Anweisung oder einer Schleife), 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

```
1 //Anweisungen vorher
2 if (1==0)
3 {
4   // nie erreichbarer Code
5 }
6 //Anweisungen nachher
```

Nach der Optimierung

```
1 //Anweisungen vorher
2 //Anweisungen nachher
```

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 be-

nötigter Sprungbefehle im Maschinencode. So gibt es unter anderem für den Compiler folgende Möglichkeiten:

- (Teilweises) Auflösen von Schleifen (engl. „Loop unrolling“), wenn kein inhaltlicher Zusammenhang zwischen dem Schleifenkörper und der Abbruchbedingung vorliegt (verringert u. a. die Anzahl nötiger Sprünge)
- Zusammenfassen von mehreren voneinander unabhängigen Schleifen mit identischen Abbruchbedingungen (Verringert Verwaltungsaufwand für Schleifen)
- 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 (Ermöglicht Verwendung des effizienten "Jump-Not-Zero"-Befehls)

Beispiel: Loop unrolling

Vor der Optimierung

```
1 int i;
2 for (i=0;i<5;i++)
3 {
4   array[i]=array[i]*2;
5 }
```

Nach der Optimierung

```
1 array[0]=array[0]*2;
2 array[1]=array[1]*2;
3 array[2]=array[2]*2;
4 array[3]=array[3]*2;
5 array[4]=array[4]*2;
```

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

Beispiel: Zusammenfassen von Schleifen

Vor der Optimierung

```
1 int i;
2 for (i=0;i<5;i++)
3 {
4   array1[i]=0;
5 }
6 for(i=0;i<5;i++)
7 {
8   array2[i]=1;
9 }
```

Nach der Optimierung

```
1 int i;
2 for (i=0;i<5;i++)
3 {
4   array1[i]=0;
5   array2[i]=1;
6 }
```

Hierbei wurde der Verwaltungsaufwand für die zweite Schleife eingespart.

Einsparung von Unterfunktionsaufrufen Vor allem bei schnell zu berechnenden Unterfunktionen ist der Rechenaufwand für den Sprung in die Unterfunktion, die Übergabe der Parameter und den Rücksprung in die Hauptfunktion größer als der Aufwand zur Bearbeitung der Unterfunktion. Der Compiler kann Unterfunktionsaufrufe einsparen, indem er ihren Code modifiziert und direkt in die aufrufende Funktion integriert.

Beispiel

Vor der Optimierung

```
1 int doubleValue(int x)
2 {
3   return x*2;
4 }
5
6 ...
7   a=doubleValue(b);
8 ...
```

Nach der Optimierung

```
1 ...
2   a=b*2;
3 ...
```

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 anderen 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 5 „Hinweise zum Debuggen“.

Beispiel: Vorberechnung von Konstanten und Einsparung von Hilfsvariablen

Vor der Optimierung

```
1 ...
2   float pi = 3.14159;
3   float radius = 2*pi*r;
4 ...
```

Nach der Optimierung

```
1 ...
2   float radius = 6.28318*r;
3 ...
```

Hier wurde die Berechnung von `2*pi` 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 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.

4.1 Syntax und Semantik von C

4.1.1 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 4.1 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 4.1: Ein typisches C-Programm, `foo.c`

4.1.2 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 4.2: `#define` Beispiel

Beispiel 4.2 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` ist eine Möglichkeiten 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 4.3 verwen-

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

Listing 4.3: `#ifndef` Beispiel

det die „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` fortgefahren. 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 4.1.13). 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
       Compiler übergeben! Denn 0 ist niemals erfüllt.
5 #else
6   //anderer Code
7 #endif
```

Listing 4.4: `#if` Beispiel

In der Sprache C gibt es analog zu dem Präprozessor-Befehl `#if` einen Befehl `if` (Einführung in Abschnitt 4.1.15). Der maßgebliche Unterschied zwischen den beiden Formen der bedingten Anweisung ist, dass die Bedingung des Präprozessorbefehls schon zum Zeitpunkt des Kompilierens fest stehen 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 4.4 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 4.5 demonstriert das Einbinden eines Bibliotheksmoduls avr/io.h und einer Projektdatei `myTypes.h`.

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

Listing 4.5: `#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?

4.1.3 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 Kommentar-Blöcke können mit `/*` eingeleitet und mit `*/` beendet werden. Kommentare sind syntaktisch irrelevant. Listing 4.6 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 4.45.

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

```
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 4.6: 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?

4.1.4 Datentypen

Die Größe von Datentypen ist abhängig vom verwendeten Compiler und der Zielplattform. Im *Praktikum Systemprogrammierung* wird der AVR-GCC Compiler benutzt, eine Variante des GCC. Die Tabelle 4.1 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,

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 4.1: Datentypen in 16-Bit Architekturen

mit welcher es möglich ist Datentypen fester Größe zu deklarieren unabhängig vom konkreten System. Im Praktikum Systemprogrammierung wird diese Art der Deklaration verwendet.

Der Typ `void` kann nicht instanziiert werden, sondern wird für Zeiger unbekannten oder beliebigen Typs verwendet. Zeiger werden in Abschnitt 4.1.7 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?

4.1.5 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 4.7 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 4.8 illustriert dies. In Abschnitt 4.1.5 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 4.1.12) geschrieben werden.

3. *Projektweite Variablen*
   Werden Variablen im Interface (.h-Datei) deklariert, können diese projektweit, durch Einbinden der .h - Datei, sichtbar gemacht werden.

Storage Specifiers

Ein *Storage Specifier* ist ein Schlüsselwort, welches Eigenschaften einer Variable festlegt. Es ist wichtig zwischen diesen und den in Abschnitt 4.1.12 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 wiederzuspiegeln. Variablen, wie bisher vorgestellt, 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,

[1] Dies 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 4.7: Variablendeklaration mit und ohne Initialisierung

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. 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
       angelegt
2  int myFunction(void) {
3    int var3 = 47; // nur innerhalb der Funktion sichtbar
4    static int var4 = 0; // behält ihren Wert auch nach
       Verlassen der Funktion
5    var3 = var3 + 1; // hat immer den Wert 48 an dieser Stelle
6    var4 = var4 + 1; // bei jedem Aufruf der Funktion wird
       var4 um 1 erhöht
7    fluxCapacitor = var3; // schreibe 48
8    return var3+var4;
9  }
10 // ...
11 myFunction(); // gibt (47+1)+(0+1) = 49 zurück
12 myFunction(); // gibt (47+1)+(1+1) = 50 zurück
13 myFunction(); // gibt (47+1)+(2+1) = 51 zurück
14 fluxCapacitor = 2; // legaler Befehl
15 var4 = 40; // illegaler Befehl, da var4 hier nicht sichtbar
       ist
```

Listing 4.8: Variablendeklaration 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?

4.1.6 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 Vorsicht geboten, weil man die Typprüfung des Compilers umgeht. Listing 4.9 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 4.9: 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?

4.1.7 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 Variablendeklaration 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 4.10 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 Variablendeklaration 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 4.10: Zeiger

Lernerfolgsfragen

- Was ist ein Zeiger (Pointer) auf eine Variable?
- Betrachten Sie den Code in Listing 4.11.
  - Wieso wird diese Funktion mit hoher Wahscheinlichkeit zu verherenden 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 4.1.5.
- Was versteht man im Zusammenhang mit Zeigern unter den Begriffen „Referenzierung“ und „Dereferenzierung“?

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

Listing 4.11: Fehlerhafter Code

4.1.8 Funktionszeiger

Funktionszeiger sind eine Teilklasse 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 4.12 dargestellt.

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

```
long funktion1(long m, long n) {
  if (m > 0) {
    if (n > 0)
      return funktion1(m-1,funktion1(m,n-1));
    else
      return funktion1(m-1,1);
  } else
    return n+1;
}

// Zeiger auf funktion1 anlegen
long (*zeiger)(long, long) = &funktion1;
long a; // Variable für das Ergebnis
a = (*zeiger)(7,6); // Funktionsaufruf (explizite Deref.)
a = zeiger(7,6);    // Funktionsaufruf (Kurzschreibweise)
```

Listing 4.12: 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?

4.1.9 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 4.13 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 in später näher erläutert.
23 unsigned int i;
24 for (i = 0; i < 4; i++) {
25   text4[i] = text1[i];
26 }
```

Listing 4.13: 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?

4.1.10 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 4.14: 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 4.15 veranschaulicht.

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

Listing 4.15: 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 AVR-GCC Schlüsselwort `prog_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 4.16 veranschaulicht das Erzeugen von Zeichenketten.

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

Listing 4.16: 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 `prog_char`?
  - Welchen Unterschied gibt es hier im Vergleich zum Typ `char`?
  - Angenommen Sie haben eine Variable `prog_char str[10]`. Ist es sinnvoll auf `str[3]` zuzugreifen? Wieso (nicht)?

4.1.11 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 4.17 dargestellt. Die Deklaration von Variablen benötigt ebenfalls das Schlüsselwort. Der Zugriff auf die Attribute des Datentypen erfolgt über den Punktoperator, wie in Listing 4.18 gezeigt.

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

Listing 4.17: 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 4.19 zu sehen ist, kann auf diese Weise eine in einer Struktur hinterlegte Funktion aufgerufen werden.

Aufzählungstypen

Die C-Syntax stellt das Schlüsselwort `enum` bereit, um eigene Aufzählungstypen zu definieren. Solche Aufzählungen vermeiden Verwechslungen ihrer Elemente und besitzen

```
// Deklarieren
struct Student arthur;
// Zugriff
arthur.immatriculation = 1979;
arthur.matrNumber = 123456;

// Weitere variable deklarieren
struct Student tricia;
// Zugriff
tricia.immatriculation = 1979;
tricia.matrNumber = 123457;
```

Listing 4.18: Beispiel für die Verwendung zusammengesetzter Datentypen

```
struct MyClassStruct {
  int attr1;
  void (*attrFunc)(void);
};
// Erzeugung einer Instanz
struct MyClassStruct obj;
// Zeiger auf diese Instanz
struct MyClassStruct* p = &obj;
// Zugriff auf ein Attribut (hier: attr1)
(*p).attr1 = 5;
// Kurzschreibweise für den Zugriff auf attr1
p->attr1 = 5;
// Aufruf der attrFunc Methode
(*((*p).attrFunc))();
// Kurzschreibweise des Aufrufs der attrFunc Methode
p->attrFunc();
```

Listing 4.19: Verwendung des Strukturoperators

einen eigenen Wertebereich. Dadurch wird der Quelltext meist deutlich besser lesbar. Eine einfache Definition und die Verwendung des Typs sind in Listing 4.20 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 4.20: 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 4.21 veranschaulicht.

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

Listing 4.21: Verwendung von `typedef`

Da `struct` Typen einen eigenen Namensraum besitzen, ist sogar der in Listing 4.23 gezeigte `typedef` legal und äquivalent zu 4.22.

```
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 4.22: Verwendung von `typedef` bei `structs`

```
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 4.23: 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 der selben Speicheradresse beginnen, können diese nicht unabhängig geändert werden. Listing 4.24 zeigt einen Verbundtypen für eine IPv4-Adresse.

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

```
union IpAddr {
  unsigned char block[4];
  unsigned long ip;
  unsigned char first_block;
};
IpAddr localhost = {{127,0,0,1}};
IpAddr lh2;
lh2.ip = localhost.ip; // == 0x0100007F
// lh2.block == {127,0,0,1}
// lh2.first_block == 127
```

Listing 4.24: `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 `enum`s 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 `struct`s wie auch `union`s können mehrere Attribute besitzen. Was ist der signifiante 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 4.24 mit einem Union Attribut `second_block` direkt auf den zweiten Eintrag von `block` in `IpAddr` zuzugreifen?

4.1.12 Type qualifiers

Type Qualifiers sind Schlüsselwörter, mit denen sich Eigenschaften von Typen ausdrücken lassen. Diese lassen sich auch in zusammengesetzen 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 4.25 erläutert die Wirkung dieses Schlüsselworts.

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

Listing 4.25: 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 4.26 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
       möglich
13 *px = 89; // illegal!
14 // Das Ändern von py ist NICHT legal, da py konstant ist
15 py = px; // illegal!
16 // Ein konstanter Zeiger auf eine veränderliche Variable
17 int* const pz = &z;
18 *pz = 89; // legal!
19 pz = 0; // illegal!
20 /* 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. */
21 short* const* (*volatile func)(char const* c, int i);
```

Listing 4.26: Zusammengesetzte Typen mit Type Qualifiers

Die Variablendeklaration und damit auch die korrekte Aussprache komplizierterer Konstrukte orientiert sich an der Hierachie der C-Operatoren. Eine hervorragende Erklärung hat T.Birnthaler 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, indem das `const` Schlüsselwort (als Schlüsselwort) vorkommt, der aber bei einer Variablendeklaration 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`

4.1.13 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: `var3 = 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 rundet es nach unten.
Beispiel: `var = 103 / 4; //var ist 25, da 103/4 gleich 25.75 ist`

Mithilfe von `/` lässt sich auch eine aufrundende und kaufmännisch rundende 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. -dekrement, steht er hinter dem Operand, bezeichnet man ihn als Postinkrement bzw. -dekrement. 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 0b11111 (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 4.3.7.

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: `var25 += 17; //äquivalent zu var25 = var25 + 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 4.2: Operatoren

Zusammenfassung

Die Tabelle 4.2 zeigt eine Kurzzusammenfassung der zuvor vorgestellten Operatoren in absteigender Präzendenz (erstgenannte werden vor letztgenannten ausgeführt - falls ungeklammert).

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 ^ b) == (a | b)` nachweisen?

4.1.14 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 4.27 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 4.27 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 gearbeitet wird und zyklische Funktionsaufrufe vorkommen. Für das obige Beispiel wür-

```
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 4.27: Beispiel einer Funktion und ihres Aufrufs

de also die in Listing 4.28 genannten Zeilen am Anfang von Listing 4.27 genügen, um der Funktion `main` die Funktion `istGerade` bekannt zu machen.

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

Listing 4.28: 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*?

4.1.15 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 4.29 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 4.30.

Verzweigung bei aufzählbaren Datentypen

Aufzählbare Datentypen sind grundsätzlich alle ganzzahligen Zahlen-Typen sowie selbst definierte Aufzählungstypen. Grundsätzlich ist es möglich, derartiges mit `if` Anweisungen, wie in Listing 4.30 gezeigt zu realisieren. Bei mehreren Bedingungen ist es der Übersichtlichkeit halber sinnvoller mit den Schlüsselwörtern `switch` und `case` zu arbeiten. Listing 4.31 zeigt ein Beispiel.

```
// Bei mehren Befehlen mit Klammern
if (var == 5) {
  // Nur wenn var fünf ist
  Befehl1();
  Befehl2();
}

// Bei einem Befehl sind Klammern nicht notwendig - dieses
   Beispiel zeigt aber, dass dadurch sehr leicht Fehler
   entstehen, da nachstehende Befehle eventuell so aussehen
   als gehörten sie zu dem if-Block
if (var == 5)
  Befehl1();
  Befehl2(); // Dieser Befehl wird immer ausgeführt, also
     unabhängig von var

if (var == 5) {
  // Nur wenn var fünf ist
  Befehl1();
  Befehl2();
} else {
  // Nur wenn var NICHT fünf ist
  Befehl3();
  Befehl4();
}
```

Listing 4.29: Verzweigung mit if

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

Listing 4.30: 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
          ausgeführt - Dieser Effekt wird als Fall-Through
          bezeichnet und sollte der Übersichtlichkeit halber
          immer explizit mit einem Kommentar hervorgehoben
          werden
10
11   case 3:
12     Befehl3(); // Befehle für wert == 3 und wert == 2
13     break;
14
15   default:
16     Befehl4(); // Befehle für alle anderen Fälle
17 }
```

Listing 4.31: 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 Initialisierungsteil 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 4.32 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.

```
int i;
//(Initialisierung; Abbruchbedingung;
// Inkrementierungsvorschrift)
for (i=0; i<10; i++) {
  if (i == 7) {
    //beim 7. Durchlauf wird keine Meldung ausgegeben
    continue;
  }
  // Das hier wird 9 mal ausgeführt
  printString("Das ist Iteration ");
  printNumber(i);
}
```

Listing 4.32: 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 4.33 zeigt ein Beispiel.

```
int i = 0;
while (generateRandomNumber() > 42) {
  i++;
}
```

Listing 4.33: 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 4.34 zeigt ein Beispiel.

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

Listing 4.34: 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?

4.1.16 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 (teilweise ist auch eine hexadezimale oder oktale Darstellung hilfreich) zu notieren. Beispiel 4.35 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 4.36 verwendet, um die Maske durch den Compiler ausrechnen zu lassen - dies geschieht zur Compilezeit, verlangsamt also die Ausführungsgeschwindigkeit des Programms nicht.

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

Listing 4.35: 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 4.36: Bitmasken

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 4.37 zeigt verschiedene Varianten von Registermanipulationen.

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

Listing 4.37: 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?

4.1.17 Inline Assembler

In manchen Fällen kann es erforderlich sein, Assemblercode in eine C Datei einzubetten. Dazu werden die Schlüsselwörter `asm volatile` verwendet. Das Listing 4.38 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"
3 "out DDRB, r16"
4 "JMP 0x0000"
5 );
```

Listing 4.38: Inline Assembler

Lernerfolgsfragen

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

4.1.18 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 4.39 zeigt die Verwendung.

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

Listing 4.39: Zufallszahlen

Lernerfolgsfragen

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

4.2 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 untereinan-

der 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.

4.2.1 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 4.40. In diesem Beispiel finden Sie einige spezielle Tags. Diese Tags werden für das Dokumentationstool Doxygen, siehe Kapitel 6, 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 dann unverständlich bleibt oder sein könnte sollte kommentiert werden.

- 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 6,„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 4.40: Dokumentation von Funktionsbestandteilen

4.2.2 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 Extraktion von Funktionen. Sinngemäß zusammenhängende Codestücke oder solche, die oft wiederkehren, sollten herausgenommen und in eine eigene Funktion ausgelagert werden. Dies gibt den entsprechenden Codestücken erstens einen verständlichen Bezeichner, zweitens können Redundanzen reduziert und der Code verkleinert werden. In Listing 4.41 ist eine unverständliche Funktion gezeigt. Diese ist in Listing 4.42 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 4.43 dargestellt, vermieden werden.

```
int value;
int *pointer;
char highBits;

int main(void){
  int i;
  for(i=1; i<10; i++){
    if(!highBits){
      value = *pointer & 0b00001111;
    } else {
      value = *pointer & 0b11110000;
    }
    highBits++;
    // ...
    if(highBits){
      pointer = pointer+1;
      highBits = 0;
    } else {
      highBits = 1;
    }
  }
}
```

Listing 4.41: Überfüllte Funktion

4.2.3 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`) schreiben. Ein 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 4.42: 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 4.43: Doppelter Negation

⇓

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

Listing 4.44: Vermeidung doppelter Negation

Gute Bezeichner

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

- schlecht: `lstadcnv`
- besser: `lastADConversion` oder `lastADC`

Überflüssige Variablen

Temporäre Variablen sind eventuell überflüssig, wie in Listing 4.45 illustiert.

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

Listing 4.45: Überflüssige Variable

⇓

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

Listing 4.46: 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 4.47: 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 4.48: 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) >
       voltageTreshold) || (readSensorA() > currentTreshold)))
        {
3     alert();
4   }
5 }
```

Listing 4.49: Großes if-Konstrukt

⇓

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

Listing 4.50: Einführung beschreibender Variable(n)

Es ist allerdings anzumerken, dass in Listing 4.50 alle Funktionen ausgeführt werden, egal wie der Wert von `runlevel` ist. In Listing 4.49 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 verwendet werden. Vgl. Listing 4.51

Datenstrukturen Auflistungen, deren Bedeutung nur dem Programmierer selbst verständlich sind, sollten in kommentierte `structs` ausgelagert werden. Auch für das Merken von Zuständen sollten selbsterklärende Bezeichner gewählt werden, hier bieten sich `enums` an.

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 4.51: 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 4.52: Konstantenverwendung

Ein weiterer Vorteil ist, dass eine Änderung des technischen Typs mehrerer Variablen des gleichen logischen Typs bei Verwendung eines `typedef`s eine einzige Änderung erfordert. Listing 4.56 zeigt die modifizierte Version des in Listing 4.55 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.

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

Listing 4.53: Seltsames Array

⇓

```
//statt unbekannter Zahlenkodierung, verwende enum
enum Studiengaenge {
  DIPLOM, BACHELOR, MASTER
};
enum StudentIndex {
  STUDIENGANG = 0,
  MATRNUMMER = 1
};

unsigned int marvin[2];
marvin[STUDIENGANG]=MASTER;
marvin[MATRNUMMER]=245915;
```

Listing 4.54: Array mit Bezeichnern

⇓

```
enum Studiengaenge {
  DIPLOM, BACHELOR, MASTER
};
//statt Array konstruiere aussagekräftiges struct
struct Student {
  enum Studiengaenge studienGang;
  unsigned int matrNummer;
};

struct Student marvin;
marvin.studienGang = MASTER;
marvin.matrNummer = 245915;
```

Listing 4.55: Structverwendung

```
// Definiere "Studiengang" als anonymes enum
typedef enum {
  DIPLOM, BACHELOR, MASTER // ...
} Studiengang;
// lege extra Typnamen für Matrikelnummer und Jahr an
typedef unsigned short Matnum;
typedef unsigned short Jahr;
typedef struct {
  Studiengang studienGang;
  Matnum matrNummer;
  Jahr immatrikulation;
  Jahr geboren;
} Student;

// Definition ohne Initialisierung
Student marvin;
marvin.studienGang = MASTER;
marvin.matrNummer = 123456;
marvin.immatrikulation = 2009;
marvin.geboren = 1989;

// Initialisierung eines konstanten structs
Student const marvin = {
  .studienGang = MASTER,
  .matrNummer = 123456,
  .immatrikulation = 2009,
  .geboren = 1989,
};
```

Listing 4.56: Structverwendung

Lesbares Bitshifting

In Abschnitt 4.37 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 man die einzelnen Bits nicht um ihre Stelligkeit verschiebt, sondern den Namen des entsprechenden Bits verwendet. 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 4.57.

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

Listing 4.57: Schlecht lesbares Bitshifting

⇓

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

Listing 4.58: Bereits lesbar

⇓

```
//sollte global verfügbar gemacht werden
//"Set Bit"
#define sbi(bitset,bit) bitset |= (1 << (bit))
//"Clear Bit"
#define cbi(bitset,bit) bitset &= ~(1 << (bit))

//--------------------------------------
//Dann kann verwendet werden:
sbi(ADCSRA,ADEN); //ADC Enable Bit
cbi(ADCSRA,ADIE);  //ADC Interrupt Enable Bit
```

Listing 4.59: 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 4.58 zeigt.

Die in Listing 4.59 gezeigten `cbi` und `sbi` Makros waren ursprünglich Teil der AVR-GCC-Toolchain. Sie wurden in neueren Versionen herausgenommen, um dem Compiler Optimierungen zu ermöglichen, die auf Hardware-sbi Funktionen basieren, sofern diese zur Verfügung stehen.

4.3 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.

4.3.1 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.
- Der vorherige Punkt trifft nicht zu, wenn unabhängige Module verwendet werden. In diesem Fall empfiehlt es sich Konstanten, Strukturen, etc. dem jeweiligen Modul zuzuordnen. Eine Ausnahme bilden Projektweite Defines und Typedefs.

4.3.2 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 4.60 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 4.60: Doxygen-Kommentare

4.3.3 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. Gleiches 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.

4.3.4 Definitionen und Konstanten

- Feststehende Werte sollten mit der `#define`-Direktive definiert werden.
- Definition von Zeichenketten mit `prog_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 4.61 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 4.61: Definition von Konstanten und Symbolen über Präprozessordirektiven

4.3.5 Klammern

- Der Einsatz von Klammern erhöht die Lesbarkeit des Quelltextes.
- Schließende geschwungene Klammern stehen in einer eigenen Zeile, öffnende nicht bzw. auch[3].

4.3.6 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 4.62: 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.

4.3.7 Casten von Variablentypen

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 }
```

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

```
11
12 // Test B
13 lcd_line2();
14 if (x+1 > 200) {
15   lcd_writeString("groesser"); // <---
16 } else {
17   lcd_writeString("kleiner");
18 }
```

Listing 4.63: 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 4.64: 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.*“.

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.

5 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 5.2 führt in das Debuggen mit den Bedienelementen von AVR Studio ein. In Kapitel 5.3 werden Probleme beim Debuggen diskutiert, die meistens mit der Optimierung des Compilers zusammenhängen. Schließlich werden in Kapitel 5.4 fehlerhafte Programme vorgestellt, die mit Hilfe der in den vorherigen Kapiteln diskutierten Verfahren korrigiert werden.

5.1 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 und unübersichtlicher.

5.1.1 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 relevanten Codesegmente immer kleiner werden, 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 AVR Debugger aufgespürt und behoben werden.

5.1.2 Debuggen mit dem AVR 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 JTAG ICE mkII Interface verwendet, um eine solche Verbindung herzustellen.

AVR Studio enthält einen integrierten Debugger, der auch für den ATmega 644 Mikrocontroller verwendet werden kann. Der AVR Debugger wird über die Debug-Symbolleisten, welche in Abbildungen 5.1 dargestellt wird, und über die Menuelemente im Debug-Dropdownmenu gesteuert.

Abbildung 5.1: Die Debug-Symbolleisten von AVR Studio

5.2 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.

5.2.1 Überwachen der Programmausführung

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

Zustand markiert der AVR Debugger die aktuelle Position der Programmausführung im Quellcode (Abbildung 5.2).

Hinweis

Die Unterbrechung der Programmausführung kann unter Umständen bekannte Fehler verändern oder neue produzieren. Beispielsweise bei zeitkritischen Anwendungen oder der Kommunikation mit anderen Geräten, die nicht angehalten wurden und deren Ausführung fortsetzen.

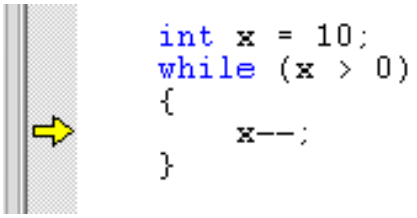


Abbildung 5.2: Markierung der aktuellen Position der Programmausführung

Pausieren des Programms

Während das Programm ausgeführt wird, kann es mit Hilfe des Pause-Knopfs (Abbildung 5.3) angehalten werden, um zu überprüfen, an welcher Stelle im Quelltext sich die Programmausführung zur Zeit befindet.

Abbildung 5.3: 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.

Darüberhinaus 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. Abbildung 5.4 zeigt die Bedienelemente von AVR Studio, mit denen Breakpoints gesetzt und entfernt werden können. Um einen Breakpoint zu setzen, klickt man zunächst in die entsprechende Codezeile und dann auf den Toggle-Breakpoint-Knopf. 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. Der Knopf Remove-all-Breakpoints entfernt alle gesetzten Breakpoints.

Hinweis

Es kann vorkommen, dass ein Breakpoint nicht in einer Zeile platziert werden kann, oder beim Kompilieren entfernt wird. Die liegt daran, dass durch die Compileroptimierungen nicht jede Zeile C-Code im Kompilat vorhanden ist.

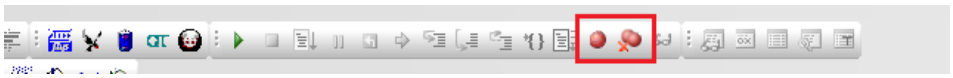


Abbildung 5.4: Die Bedienelemente für Breakpoints

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 5.2.1) 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 5.5) 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 wir nur dann, wenn diese Zeile auch erreichbar ist.

Abbildung 5.5: Der Run-To-Cursor-Knopf

Schritt-Für-Schritt-Ausführung

Die Schritt-Für-Schritt-Ausführung (auch: schrittweise Abarbeitung) des Programms ermöglicht eine feinkörnige Kontrolle des Programmflusses. Die Bedienelemente zur schrittweisen Abarbeitung (Abbildung 5.6) bieten dem Benutzer eine Auswahl zur unterschiedlichen Behandlung der Unterfunktionsaufrufe.

- Der *Step-Into-Knopf* weist den Debugger an, einem Sprung in eine Unterfunktion zu folgen, so dass das Verhalten dieser Funktion ebenfalls schrittweise überprüft werden kann. Am Ende der Unterfunktion folgt der Debugger automatisch dem Rücksprung in den ursprünglichen Kontrollfluss.
- Mit dem *Step-Over-Knopf* wird die Unterfunktion ohne Schritt-für-Schritt Ausführung bearbeitet, danach springt der Debugger in die nächste Codezeile des aktuellen Kontrollflusses.
- Der *Step-Out-Knopf* ermöglicht es schließlich, das Debuggen eines Unterfunktionsaufrufs abzubrechen und in den aufrufenden Kontext zurückzukehren.

Abbildung 5.6: Die Bedienelemente zur schrittweisen Abarbeitung

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.

5.2.2 Disassembler

AVR Studio bietet die Möglichkeit den vom Compiler erzeugten Assemblercode, mitsamt der zugehörigen mnemonischen Anweisungen, zu betrachten. Je höher die Optimierungsstufe ist, desto stärker kann dieser Code vom ursprünglichen C-Code abweichen, obwohl die Funktionalität erhalten bleibt (weitere Hinweise zur Codeoptimierung können in Kapitel 5.3 gefunden werden). Bei Betrachtung des Assemblercodes kann ganz genau nachvollzogen werden, in welcher Reihenfolge der Prozessor welche Befehle ausführt und wann er an welche Stelle springt. Die Assembleransicht kann auf die gleiche Weise wie die C-Ansicht durchlaufen werden. Den Assemblercode zu debuggen ist aufgrund seiner Komplexität oft die letzte Option, gleichzeitig aber auch die beste, um ein mögliches Problem im Detail zu verstehen.

Beispiel

In Abbildung 5.7 ist ein sehr kleines Programm dargestellt, welches in einer Endlosschleife von Port A einen Wert einliest, dessen Fakultät berechnet und das Ergebnis auf den Ports C und B ausgibt. Wie man aus den Registern auf der rechten Seite erkennt, wurde

die Fakultät von 5 bereits als 0x78 = 120 korrekt berechnet und ausgegeben. Das Programm befindet sich aktuell im zweiten Durchlauf der Schleife. Die Optimierungsstufe ist auf -O1 gesetzt.

Wählt man den Punkt „*Goto Disassembly*“ aus dem Kontextmenü in der Zeile `PORTB = f`, gelangt man zur Assembleransicht (Abbildung 5.8). Die Breakpoints werden hier automatisch übernommen. Außerdem wird analog zu der C-Ansicht der Programcounter[1] durch einen gelben Pfeil angezeigt. Darüberhinaus markiert ein grüner Pfeil die Codestelle im C-Code, die vor dem Wechsel in die Assembleransicht ausgewählt war.

Zur besseren Orientierung sind über den eigentlichen Assemblerbefehlen die ursprünglichen C Befehle notiert, aus denen der Compiler den Assemblercode generiert hat (falls eine solche Zuordnung nach der Optimierung möglich ist). Mit Hilfe des Handbuchs kann man die Korrektheit der Assemblerbefehle überprüfen. Beispielsweise wurde die Codezeile `PORTB = f` durch vier Assembleranweisungen ersetzt, welche im Folgendem erklärt werden.

```
-- Lade die Adresse von PORTB (0x0025) in die Register (R31,R30)
LDI     R30, 0x25
LDI     R31, 0x00
-- Lade das zweite Byte des Ergebisses (f) welches an der
-- durch Y referenzierten Adresse steht in das Register 24
LDD     R24, Y+1
-- Speichere den Inhalt von Register 24 an die erste von Z
-- referenzierte Stelle (Z referenziert R30,R31).
-- Schreibe also das Ergebnis f an die Adresse 0x25 (PORTB)
STD     Z+0, R24
```

5.2.3 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 5.9 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 defininierten Wert anzeigen, wenn die Programmausführung pausiert ist. Eine Änderung des jeweils betrachteten Speicherteils wird in roter Farbe dargestellt.

[1] Der Programcounter gibt die Stelle der aktuellen Ausführung des Programms an.

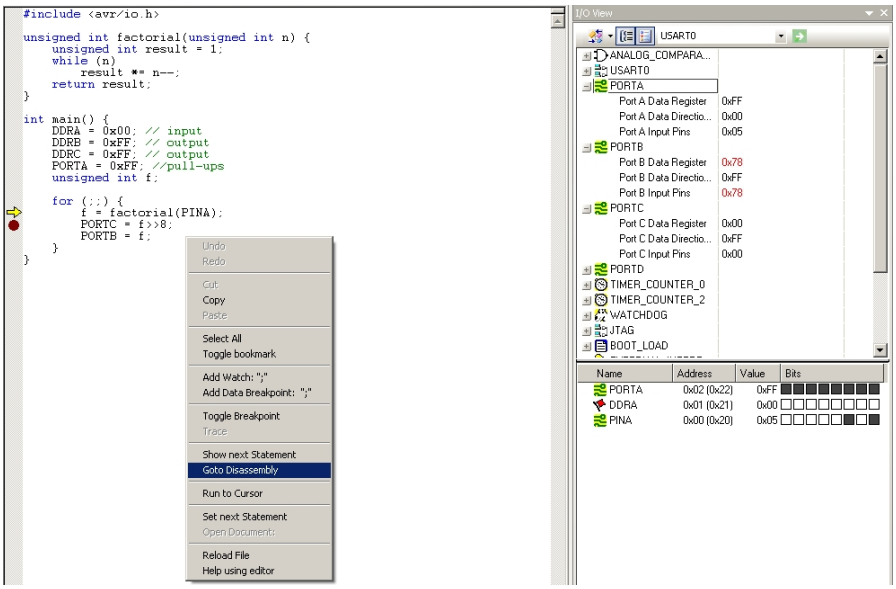


Abbildung 5.7: Disassembler-Beispiel: C-Code

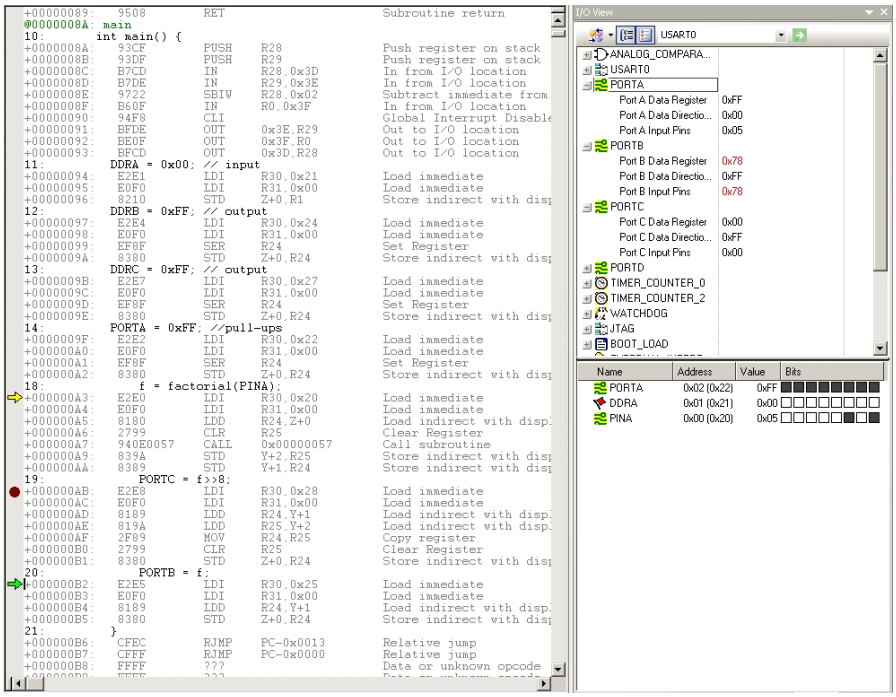


Abbildung 5.8: Disassembler-Beispiel: Assembler Code

Abbildung 5.9: Die Bedienelemente zur Speicheüberwachung

Variablenüberwachung

Sobald das Programm (z. B. durch einen Breakpoint) angehalten wurde, lassen sich die Variablen im Sichtbereich auslesen. Dazu muss die Variable mit der rechten Maustaste angeklickt, und dann „Add Watch VariablenName“ aus dem Kontextmenü gewählt werden. Es öffnet sich ein neues Fenster namens „Watch list“, in dem die Werte sowie Typ und Adresse, alle 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 5.3.2 gefunden werden.

Beispiel Ein Beispiel für eine solche Überwachung ist in Abbildung 5.10 dargestellt. Das Watch-Fenster zeigt an, dass Variablen `endian` und `length` aktuell überwacht werden. Die Variable `endian` mit dem Typ `uint16_t` (ohne Vorzeichen, 16 Bit lang) hat derzeit den Wert 0. Außerdem ist erkennbar, dass sie sich an der Adresse 256 (0x0100) im SRAM befindet.

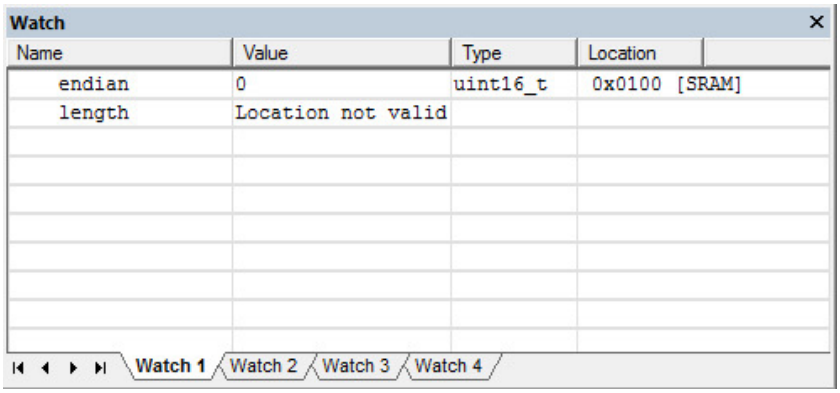


Abbildung 5.10: Das Watch-Fenster

In diesem Beispiel kann die Variable `length` 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 5.3.2

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 5.11 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 }
```

Abbildung 5.11: 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, der in einem Ringpuffer zwischengespeichert wird, durch einen `head` und einen `tail` Zeiger lokalisiert werden, die den Anfang und das Ende der Daten angeben. Um Fehler in einem solchen Datenstrom zu finden, gibt es die Möglichkeit sich den entsprechenden SRAM-Bereich anzeigen zu lassen.

Diese Vorgehensweise ist keinesfalls nur auf den SRAM-Speicher beschränkt. AVR Studio bietet zusätzlich die Möglichkeit sich den Programmspeicher (auch Flash genannt) anzeigen zu lassen. Wie man in Abbildung 5.12 sieht, können hier leicht Muster erkannt und Abweichungen von den erwarteten Werten festgestellt werden.

Die Speicheranzeige ist wie folgt zu lesen: Jede Zeile listet mehrere Byte des jeweiligen Speichers auf. In diesem Beispiel ist oben links der Speicher „Program“ ausgewählt, sodass die angezeigten Werte dem Inhalt des Programmspeichers entsprechen. In der blau hinterlegten 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).

Beispiel In Abbildung 5.12 steht an Adresse `0x007E48` des Programmspeichers der Wert D9F7 und an Adresse `0x007E49` der Wert 11E0.

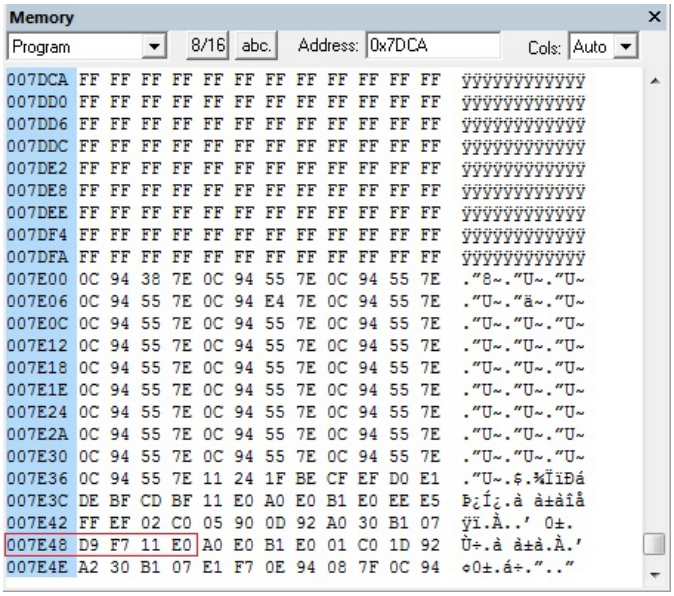


Abbildung 5.12: Die Anzeige des Mikrocontrollerspeichers

Dass die Werte in diesem Beispiel über zwei Spalten gehen liegt daran, dass der Programmspeicher 16-Bit-Wörter enthält, wohingegen beispielsweise der SRAM nach 8-Bit-Bytes adressiert wird.

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 5.13 gezeigte Fenster wird von AVR Studio automatisch auf der linken Seite der Umgebung geöffnet, sobald die Ausführung pausiert wird (Standardeinstellung). Hier kann abgelesen werden, dass derzeit das Interruptflag (Bit 7) des SREG auf 0 und das Zeroflag (Bit 1) auf 1 stehen. Durch Klicken auf das Kästchen mit dem `I` in der Zeile SREG kann man die Interrupts zur Laufzeit global einschalten.

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 werden[2].

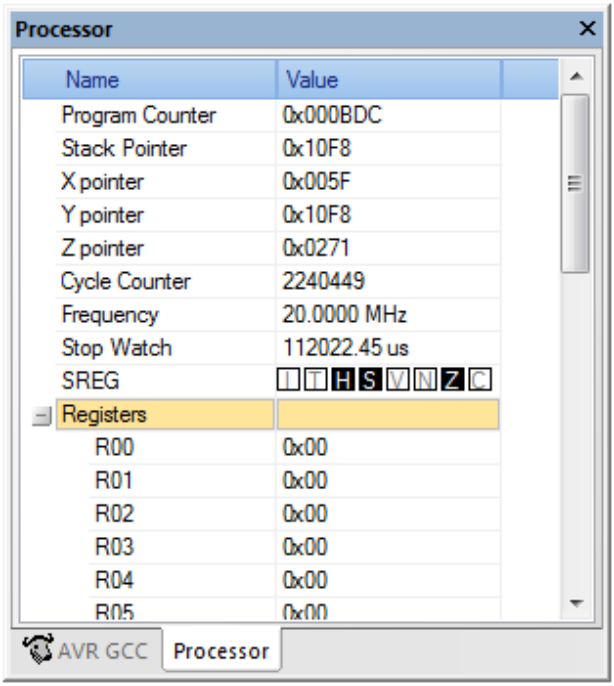


Abbildung 5.13: Die Anzeige des Prozessorstatus und der Register

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.

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.

[2]Der ATmega644 hat einen SRAM von 0x1000 Byte (4KB). Dieser beginnt bei Adresse 0x100, damit ist die höchste Adresse 0x10FF. Detailiertere Informationen zu diesem Thema können dem Datenblatt des Mikrocontrollers ATmega 644 entnommen 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 festgestellt werden ob ein Pin als Ein- oder Ausgang gesetzt, ob ein Taster gedrückt ist oder auch welcher Wert sich gerade im Timer-Register befindet. Dort werden alle Register des Mikrocontrollers aufgelistet. Geöffnet werden kann die IO-View über „View“ → „Toolbars“ → „I/O“. 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 AVR Studio) nur aktualisiert, wenn das Programm angehalten wird.

Beispiel In Abbildung 5.14 kann man die genaue Konfiguration der zu Port B gehörenden Register sehen. Außerdem werden Ports C und D, sowie die SPI Konfiguration und einige Register von Timer 0 angezeigt. Bei Port B ist die Datenrichtung aller Bits 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 Port B derzeit auf Input geschaltet ist. Durch Anklicken eines der Kästchen kann die Belegung eines bestimmten Bits in einem der Register negiert werden.

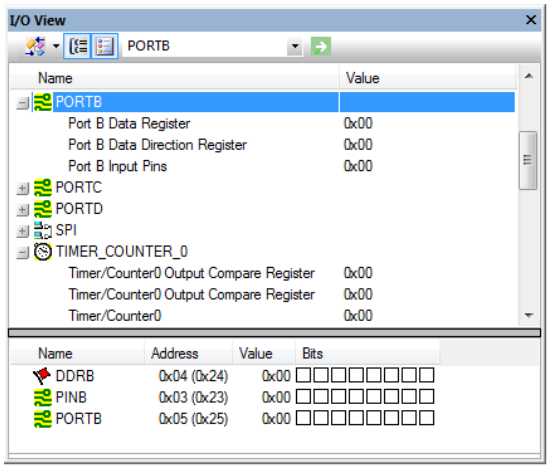


Abbildung 5.14: Die I/O-Anzeige

Verwendung Die Verwendung der I/O-Anzeige ist vielfältig, und immer dann besonders hilfreich, wenn man ein bestimmtes Gerät (etwa einen Timer) zur Laufzeit manuell umkonfigurieren oder Signale von der Außenwelt überwachen möchte. Damit lassen sich mögliche Ursachen für ein Fehlverhalten eingrenzen. Zum Beispiel kann während des Debuggens bequem getestet werden, ob ein Interrupt korrekt ausgeführt wird, wenn sich ein Pin an Port B ändert.

5.2.4 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. Als zusätzliches Werkzeug kann das LC-Display verwendet werden, um Debugnachrichten oder den Inhalt von Variablen wiederzugeben. Dies garantiert, dass das Programm weiterläuft und die Belegung von Variablen und das Eintreten von Ereignissen dennoch überwacht werden kann. Eine kritische zeitliche Verzögerung kann aber auch hier auftreten. Im Fall einer asynchronen Übertragung würde das Programm auf der anderen Seite die Übertragungsgeschwindigkeit beibehalten, wodurch Daten verloren gehen können.

Eine Ausgabe aller wichtigen Daten auf dem Display ist oft sehr zeitaufwändig und nicht sehr präzise, da die angezeigten Variablen gleich wieder veraltet sind, sobald sie ausgegeben wurden.

5.3 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 von Optimierungen des Compilers verursacht, desshalb sollte als erste Maßnahme die Optimierungsstufe für das aktuelle Projekt so weit wie möglich heruntergesetzt werden[3].

[3] Weitere Informationen zur Compileroptimierung finden Sie in Kapitel 3.5

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.

5.3.1 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

Es kann vorkommen, dass beim Klicken auf den Toggle-Breakpoint-Knopf der Breakpoint nicht in die aktuell ausgewählte Zeile gesetzt wird, weil 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 davon, zu denen keine Maschinenbefehle im Speicher des Mikrocontrollers gefunden werden, werden deaktiviert. Abbildung 5.15 zeigt einen deaktivierten Breakpoint.

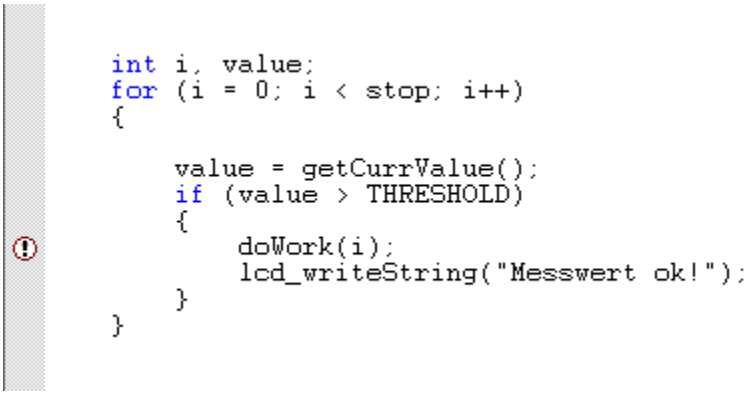


Abbildung 5.15: Ein deaktivierter Breakpoint

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
```

Die zweite mögliche Ursache ist die Compileroptimierung. Es kann vorkommen, dass der Compiler Teile des Codes entfernt, mit anderen Teilen vereinigt oder tauscht. In diese 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 }
```

Eine mögliche Lösung des Problems wäre die Variable `i` als `volatile` zu deklarieren. Diese Vorgehensweise wird im Abschnitt 5.3.2 genauer erläutert.

Lösungsvorschläge

Wenn Teile des Codes wegen Präprozessoranweisungen nicht mitkompilieren 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 Asseblercode Schritt für Schritt überprüft wird.

5.3.2 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.

Location not valid

Wenn im Watch-Fenster als Wert einer Variablen „*Location not valid*“ (Abbildung 5.16) erscheint, so konnte der Debugger keine gültige Speicheradresse für die überwachte Variable feststellen.

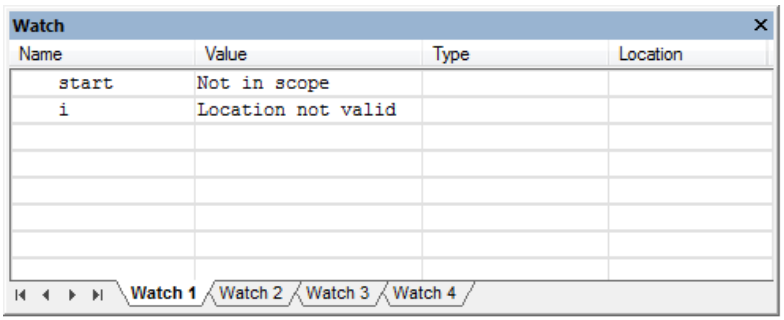


Abbildung 5.16: Variablen, deren Wert nicht überwacht werden kann

Not in scope

Eine Variable ist dann „*Not in scope*“ (Abbildung 5.16), wenn sie im aktuellen Laufzeitkontext nicht bekannt ist. D.h. der Teil des Quellcodes, der vom Debugger gerade überwacht wird, ist nicht der Sichtbarkeitsbereich (scope) der betreffenden Variable. Man betrachte Beispiel 5.17:

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 „*Not in scope*“ markiert.

```
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 }
```

Abbildung 5.17: Sichtbarkeitsbereiche der lokalen Variablen

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 auf sie (im Vergleich zu den im SRAM gespeicherten Variablen) relativ schnell zugegriffen werden. 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 „*Location not valid*“ 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  }
```

Lösungsvorschläge

Wenn Codeteile, 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 4.1.12 „Einführung in die C Programmierung“ gefunden werden, der im L^{2}P-Lernraum zur Verfügung steht.

Das Beispiel 2 aus dem Abschnitt 5.3.1 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 werden
5 }
```

5.4 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 wider.

5.4.1 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  unsigned char i;
4  unsigned char array[LENGTH];
5
6  for (i = 0; i < LENGTH; i++) {
7      array[i] = receive_char();
8  }
9  lcd_writeInt(array[0]);
10
11 ...
```

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. Die Fehlerursache liegt also nicht in der Ausgabe; Nur die erste der aufgestellten Hypothesen muss 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 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`.

5.4.2 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.

Am folgenden Beispiel mit zwei parallel laufenden Tasks wird dies verdeutlicht:

```
1  // defines & globale variablen
2  #define nop asm volatile ("nop")
3  unsigned char* pb = NULL;
4
5  void task1(void) {
6      unsigned char b = 0xBB;
7      unsigned char a = 0xAA;
8      pb = &b;            // Adresse von b an task2 übergeben
9      while (a == 0xAA) nop;      // warten bis a sich ändert
10     lcd_writeString("a changed!");
11 }
12
13 void task2(void) {
14     while (!pb) nop; // warten auf task1
15     pb--;       // zeigt auf die Adresse VOR b (wo a liegt)
16     *pb = 0;    // in a wird eine 0 geschrieben
17 }
```

Was gemeint ist

Die Variable `pb` ist global und dient der Kommunkation 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 5.18). 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 5.19). Sobald `task1` wieder ausgeführt wird, müsste die `while` Schleife verlassen und der Text „a changed“ auf dem LCD ausgegeben werden.

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

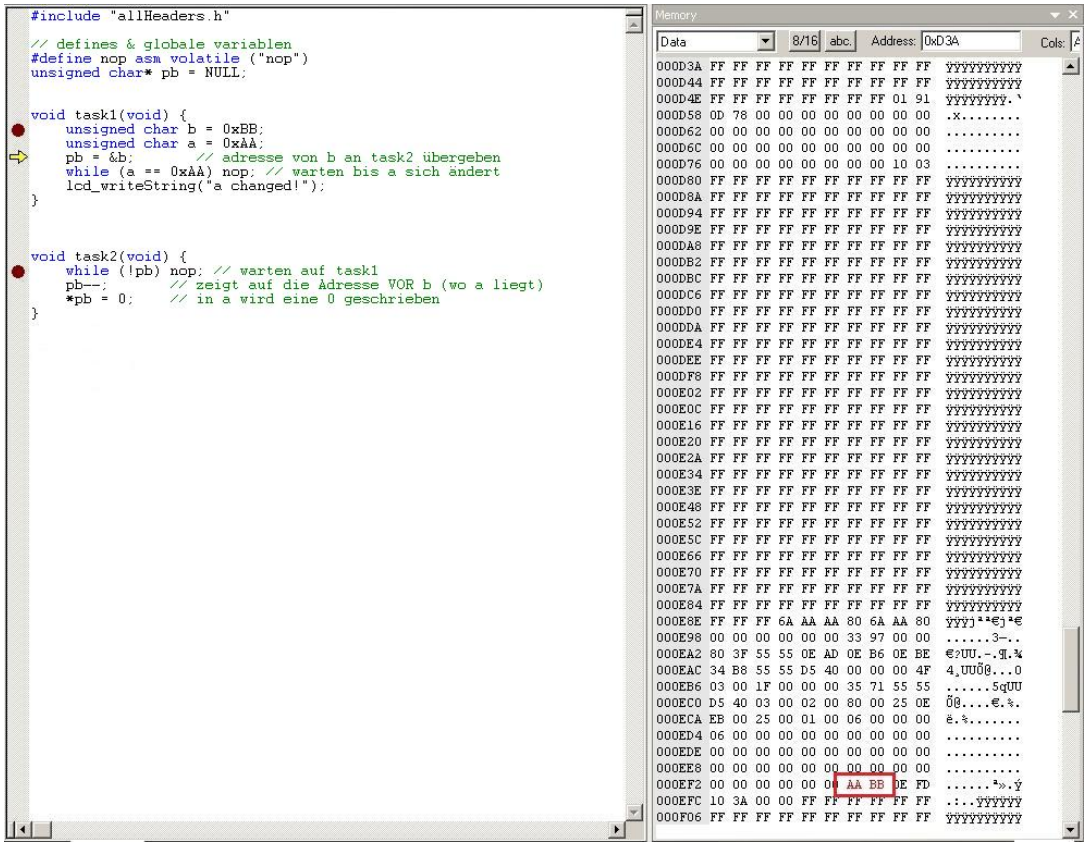


Abbildung 5.18: task1: SRAM ohne Codeoptimierung

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 }
```

Fehlersuche

Zunächst wird der Benutzer feststellen, dass die erwartete Zeichenkette nicht ausgegeben wird. Ein Versuch einen Breakpoint in Zeile 10 zu setzen wird scheitern, weil zu diesem Befehl kein Assemblercode existiert. Der nächste Schritt könnte sein, einen Breakpoint

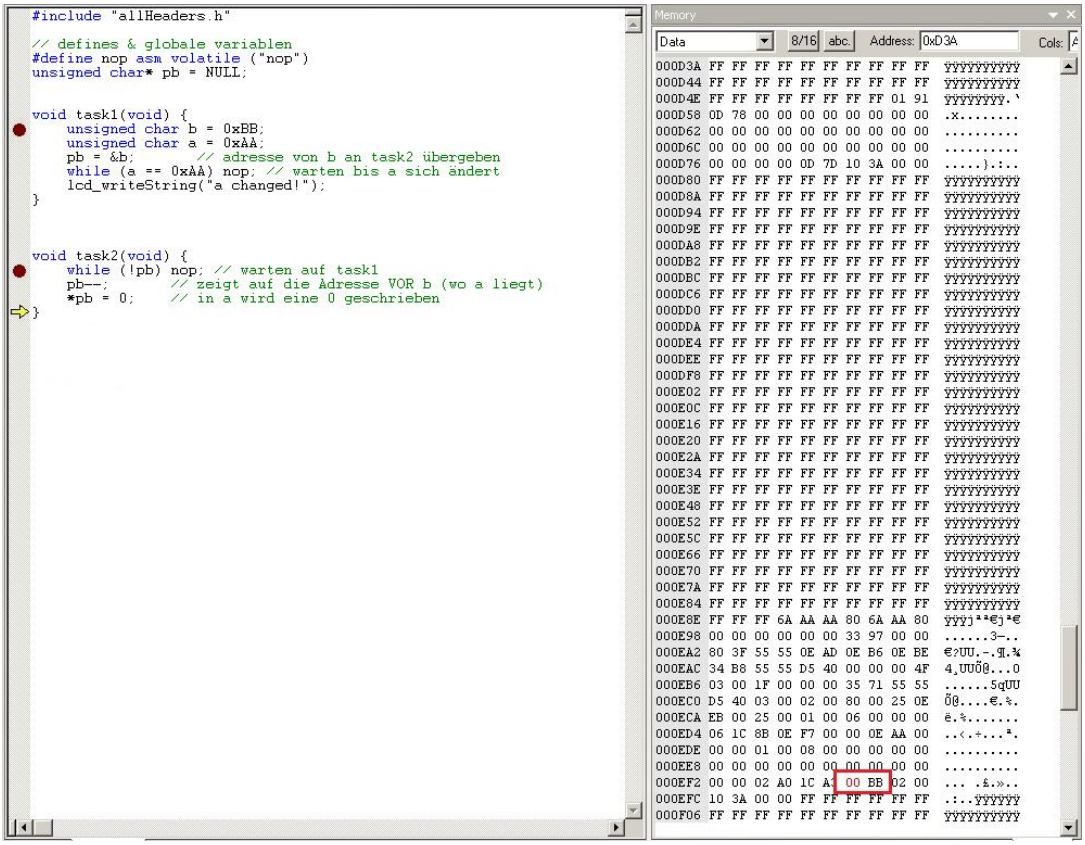


Abbildung 5.19: task2: SRAM mit Codeoptimierung

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.

6 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-, LaTeX- oder RTF-Dokument ausgegeben.

6.1 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 L^{2}P-Lernraum 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 und es ist nicht möglich, jede einzelne Methode explizit in den Unterlagen zu erklären. Deswegen 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-Kommentarem **im eigenen Code nicht verpflichtend**. Es wird jedoch dringend empfohlen, eine strukturierte 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.

6.2 Ausgabe von Doxygen

Abbildung 6.1 zeigt die Startseite der Doxygen-Doku (Datei `index.html`) zu einem fiktiven Versuch X.

Wird in diesem Dokument beispielsweise die Datei `os_Main.h` ausgewählt, so wird ihr Inhalt detaillierter dargestellt (Abb. 6.2). Hier werden alle implementierten Methoden 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 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.

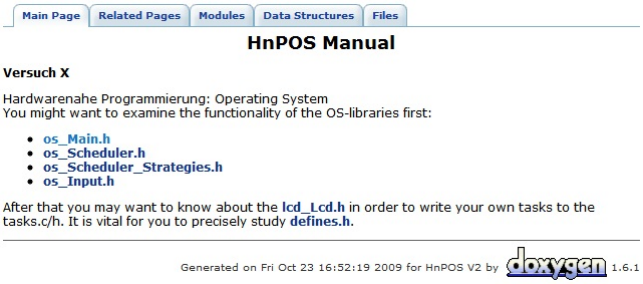


Abbildung 6.1: Die Startseite einer beispielhaften Doxygen-Dokumentation

6.3 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.

6.3.1 Konfiguration

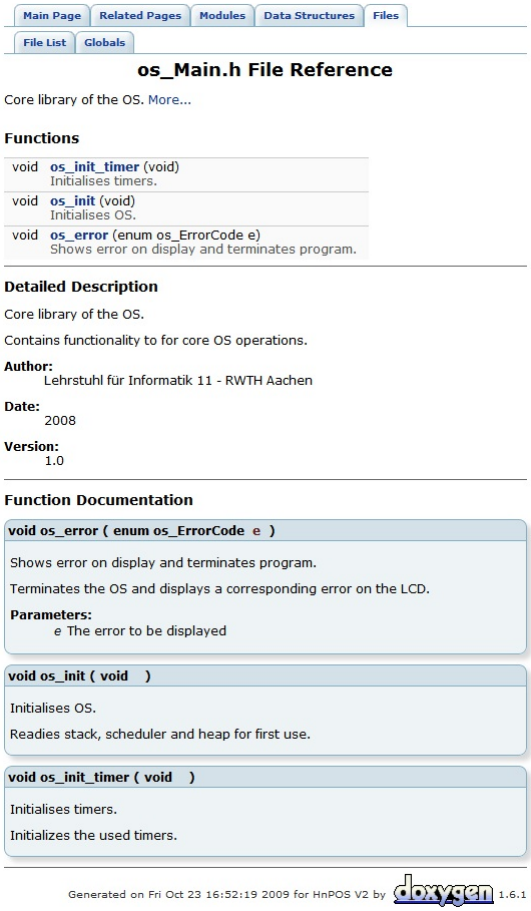
Die grafische Benutzerschnittstelle bietet einen Assistenten an, der einen Großteil der Einstellungen eigenständig vornehmen kann: den *Doxywizard* zur Erzeugung von *Doxyfiles* (in Anlehnung an den Begriff *Makefiles*). In einem Doxyfile werden alle Einstellungen gespeichert, die für die Erzeugung einer Dokumentation berücksichtigt werden sollen. Der Assistent (Abb. 6.3) 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.

6.3.2 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



Main Page | Related Pages | Modules | Data Structures | Files

File List | Globals

os_Main.h File Reference

Core library of the OS. More...

Functions

void **os_init_timer** (void)
Initialises timers.

void **os_init** (void)
Initialises OS.

void **os_error** (enum os_ErrorCode e)
Shows error on display and terminates program.

Detailed Description

Core library of the OS.

Contains functionality to for core OS operations.

Author:
Lehrstuhl für Informatik 11 - RWTH Aachen

Date:
2008

Version:
1.0

Function Documentation

void os_error (enum os_ErrorCode e)

Shows error on display and terminates program.

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

Parameters:
e The error to be displayed

void os_init (void)

Initialises OS.

Readies stack, scheduler and heap for first use.

void os_init_timer (void)

Initialises timers.

Initializes the used timers.

Generated on Fri Oct 23 16:52:19 2009 for HnPOS V2 by doxygen 1.6.1

Abbildung 6.2: Detailansicht der `os_Main.h`

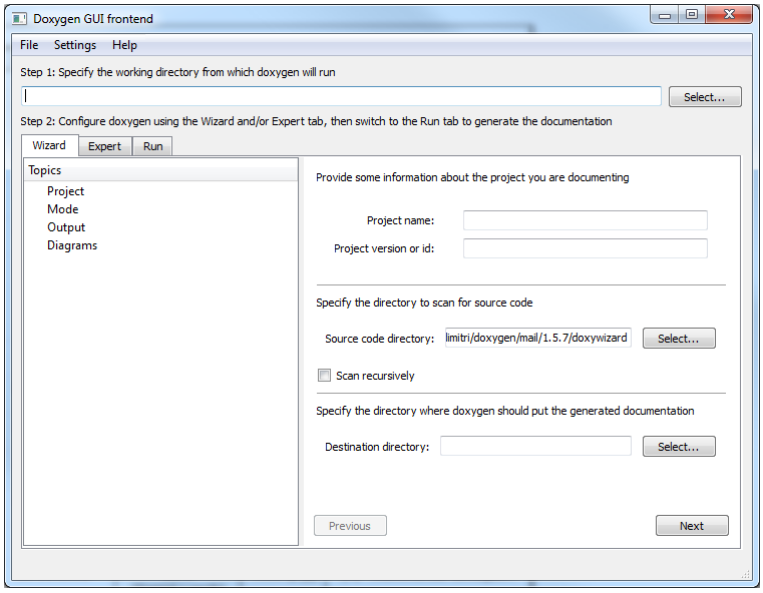


Abbildung 6.3: Der Doxygen-Assistent

Fall, dass die Ausgabe im HTML-Format erfolgt, kann die erstellte Datei *index.html* in einem beliebigen Browser geöffnet werden (Abb. 6.4).

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.

6.4 Doxygen-Kommentare

6.4.1 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.

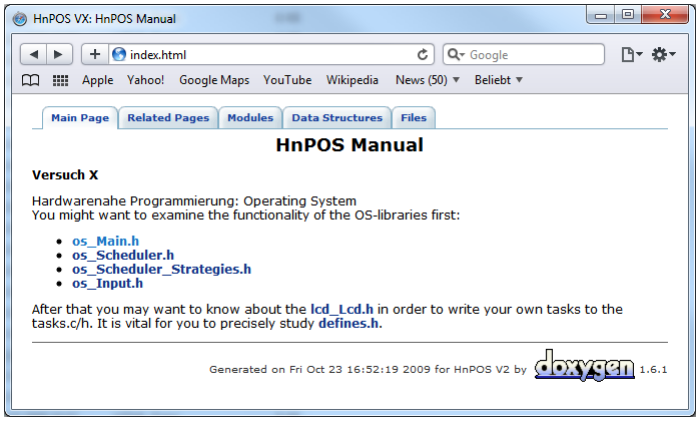


Abbildung 6.4: Die Startseite der neu erstellten Dokumentation

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

Das Dokumentationswerkzeug Doxygen erstellt Beschreibungen zu den Programmier-

schnittstellen 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 zeigt den Unterschied zwischen einem Kurzkommentar und einem ausführlichen Kommentar. Die aus diesen Codebeispielen erzeugte Doxygen-Dokumentation ist in Abb. 6.5 dargestellt.

Code in der Header-Datei:

```
1  //! Allocates some bytes (max 65535) of memory.
2  MemAddr os_malloc(MemDriver* driver, unsigned int size);
```

Code in der C-Datei:

```
/*!
 *  Allocates a chunk of memory on the medium given by the
 *  driver and reserves it for the current task.
 *  Throws an error, if the allocation fails.
 *
 *  \param driver The Memory driver to be used
 *  \param size   The amount of memory to be allocated in
 *                Bytes. Must be able to handle a single byte
 *                and values greater than 255 bytes.
 *  \returns      A pointer to the first Byte of the
 *                allocated chunk.\n NULL pointer if
 *                allocation fails.
 */
MemAddr os_malloc(MemDriver* driver, unsigned int size ){
  ...
}
```

⇓

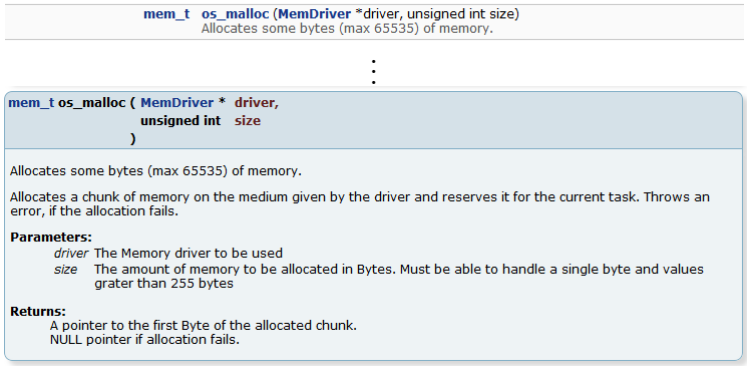


Abbildung 6.5: Die aus dem Beispielcode erzeugte Doxygen-Dokumentation

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.

6.4.2 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.

6.4.3 Kommentieren von Dateien

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

```
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  */
```

Die 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.

6.4.4 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 (Dieses 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] - (**s**ee **a**lso) 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.

7 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 Verständniss hilfreich sein können.

7.1 AVR-Mikrocontroller

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

7.2 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_bei_der_C-Programmierung
- http://www.cppreference.com/wiki/c/start

7.3 Doxygen

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