Embedded Programmierung

Methoden und Verfahren

Jürgen Plate, 26. Dezember 2016

Inhaltsverzeichnis

1

Softwareentwicklung

Je nach Größe des Projektes kommen während der Entwicklungsphase immer noch Änderungswünsche der unterschiedlichsten Art vom Kunden und auch vom Entwickler. Der Kunde entwickelt sich mit seinen Aufgaben weiter, und der Entwickler versteht seine Aufgabe immer besser, je länger er daran arbeitet. Damit das alles nicht aus dem Ruder läuft, gibt es noch die Rolle des Projekt- oder Entwicklungsleiters. Dessen Verantwortung ist es, mit den Hilfsmitteln des Projektmanagements die Softwareentwicklung so zu steuern, dass die Software in der geplanten Zeit, im geplanten Umfang und dem festen Budget fertig wird; das ist keine leichte Aufgabe.

Das Wichtigste dürfen wir nicht vergessen: die Kundenanforderungen! Es ist erstaunlich, wie viel Software am eigentlichen Nutzer vorbei entwickelt wird, nur weil der Entwickler denkt, er wisse, was der Kunde braucht, und sich nicht erkundigt, was der Kunde wirklich will. Oftmals verliert der Entwickler während des Entwurfs den Kunden aus den Augen und baut technikverliebt nicht gewollte Komplexität ins Programm ein, die meistens viel Zeit beim Testen kostet.

Es ist wichtig, dass eine Problemlösung möglichst einfach ist, weil sie damit an Klarheit und Übersichtlichkeit gewinnt. Eine erste Faustformel könnte daher lauten:

KISS – Keep It Simple, Stupid!

1.1 Problemanalyse

Bevor der Prozess der Softwareerstellung beginnt, ist das Wichtigste, das Ziel festzulegen: Wofür und warum muss Software erstellt werden und was muss sie können, um diese Aufgabe zu erledigen? Zunächst gibt es große Unterschiede bezüglich der Anforderungen an die Flexibilität eines Systems. Folgende Klassen lassen sich hier unterscheiden:

- Feste Anwendung: Typisch sind hier die Einsatzbereiche bei Geräten mit fest umrissenen Aufgaben, wo immer die gleichen Programme ausgeführt werden. Der Code ist vollständig in einem ROM untergebracht.
- Parametrierbare Anwendung: Hier ist der Code ebenfalls fest, jedoch kann eine Anpassung an den besonderen Einsatzfall über Parameter erfolgen, die dann in einem EEPROM oder einem RAM gespeichert werden. Bei vielen Fahrzeuganwendungen gibt es diese minimale Flexibilität.
- Programmierbare Anwendung: Vom eingebetteten System wird entweder eine Laufzeitumgebung bereitgestellt, auf der dann eine individuelle Programmierung erfolgen kann, oder ein Interpreter, der Kommandos oder Programme in einer Sprache ausführen kann. Typisch sind hier Maschinensteuerungen. Oft wird auch eine Kernanwendung vorgegeben, die durch individuelle Programme ergänzt oder modifiziert werden kann.
- Freie Programmierbarkeit: Hier gibt es keinerlei Einschränkungen der Anwendung. Der Entwicklungsaufwand und die Stückkosten für jedes individuelle Gerät sind aber sehr hoch.

Es gibt mindestens zwei Beteiligte am Softwareerstellungsprozess: den Kunden und den Entwickler. Der erste Schritt ist das schriftliche Festhalten der Kundenanforderungen. Der Entwickler schlüpft hier in die Rolle des Systemanalytikers. Er versucht, den Kunden und dessen Anwendungswelt zu verstehen und aus seiner Aufgabenbeschreibung (Lastenheft) die für die Softwareerstellung nötigen Zusammenhänge zu erfassen um daraus eine realisierbare, in sich konsistente – für die Realisierung verbindliche – Aufgabenbeschreibung (Pflichtenheft) zu erstellen. Er muss vor allem das Wesentliche vom Unwesentlichen unterscheiden, abstrahieren und auf den Kunden eingehen können, um zu erfassen, was zu tun ist.

Hilfreich können dabei die „Regeln zur Leitung des Geistes“ des französische Philosophen und Mathematikers Descartes sein:

- Übereilung und Vorurteil sind sorgfältig zu meiden.
- Jede der zu untersuchenden Schwierigkeiten ist in so viele Teile zu zerlegen, wie es möglich und zur besseren Lösbarkeit wünschenswert ist.
- Mit den einfachsten und fasslichsten Objekten ist zu beginnen und von da aus schrittweise zur Erkenntnis der kompliziertesten fortzuschreiten.
- Hinreichend vollständige Aufzählungen und allgemeine Übersichten sind anzufertigen, um sicherzugehen, dass nichts ausgelassen wurde.

Die Problemanalyse ist in einzelne Phasen unterteilt, die im Folgenden kurz umrissen werden. Zuerst wird die gegebene Situation genau untersucht, wobei die Wünsche für die EDV und die Einsatzmöglichkeiten der EDV festgehalten werden (Ist-Analyse). Ausgehend von der Beschreibung der Probleme wird dann eine Beschreibung der Lösungen für den Anwender und eine Aufgabendefinition für den Entwickler der Software erstellt (Sollkonzept). Nun wird festgestellt, ob es möglich ist, eine derartige Aufgabe überhaupt zu lösen (technische Durchführbarkeit) und ob die gefundenen Lösungen wirtschaftlich sind (ökonomische Durchführbarkeit). Die optimale Lösungsmethode wird vervollständigt und die Projektplanung ausgearbeitet.

Der Aufwand der Vorarbeiten zu einem Programm ist natürlich von der Größe des Projekts abhängig, das Planungsprinzip ist aber auch auf kleine Programme anwendbar. Auf jeden Fall sollten die Aufgabenstellung, die Lösungsmethoden, die Benutzerwünsche, die Qualifikation der Benutzer und die im Rahmen der Problemanalyse zutage getretenen Argumente schriftlich fixiert werden.

Ein verantwortungsbewusster Entwickler sollte seine Tätigkeit als Erfüllung eines Vertrags auffassen. Zu einem vertraglich fixierten Problem wird ein Programm geliefert, welches das Problem effizient löst. Voraussetzung der genauen Erfüllung eines solchen Vertrags ist die genaue Spezifikation des Problems, damit Auftraggeber und Auftragnehmer wissen, was sie vereinbaren. Das Programm als Vertragsgegenstand muss in einer Weise geliefert werden, dass es durch den Auftraggeber angewandt und gewartet werden kann. Unentbehrliches Hilfsmittel hierfür ist die Dokumentation. Man unterscheidet zwei Arten von Dokumenten:

- Gebrauchsanweisung (Benutzerhandbuch): Dem Programmbenutzer, der mit der Programmentwicklung nichts zu tun hatte, braucht über die Art der Problemlösung, den Algorithmusentwurf und andere Interna nichts mitgeteilt zu werden. Folgendes sollte die Gebrauchsanweisung jedoch enthalten:

  - Zweck und Anwendungsbereich des Programms
  - Beschreibung der Steueranweisungen zum Starten und Anhalten des Programms
  - Beschreibung der Dialogführung mit Hinweis auf vorprogrammierte Hilfen
  - Dialogbeispiele
  - Form und Bedeutung der Eingabedaten
  - Form und Bedeutung der Ausgabedaten
  - Erläuterung der einzuhaltenden Wertebereiche
  - Erläuterung der Fehlermeldungen

  Zusätzlich zum Benutzerhandbuch sollte eine Kurzanleitung vorliegen, die nur die wichtigsten, für den Umgang mit dem Programm notwendigen Schritte und Anweisungen bereithält.

- Programmbeschreibung: Für spätere Änderungen und Ergänzungen (die sogenannte Wartung) des Programms wird zusätzliche Dokumentation benötigt:
  - Name des Programmautors, Datum der Erstellung und Titel
  - Beschreibung des Problems
  - Modellbildung und Spezifikation
  - Entwicklung des Lösungsalgorithmus
  - Formulierung der Algorithmen in normierter Entwurfssprache
  - Programmquelle
  - Programmlauf
  - Diskussion
  - Überlegungen zur Korrektheit und zur Effizienz
  - benutzte Literatur

Der Softwareerstellungsprozess beginnt mit der Festlegung des Ziels. Anhand der Aufgabenbeschreibung des Kunden (Lastenheft) erfolgt die Problemanalyse, die der Entwickler zunächst in eine verbindliche und realisierbare Aufgabenbeschreibung, das Pflichtenheft, umsetzt. Weitere Dokumente, die für den Auftraggeber die Erfüllung der Aufgabe nachvollziehbar machen, sind die Gebrauchsanweisung und die Programmbeschreibung. Der Entwickler sollte sich während der Problemlösung immer bewusst sein, dass eine gute Lösung einfach sein sollte. Heutzutage legt man großen Wert auf einen transparenten Programmierstil: Ein gutes Programm ist so geschrieben, dass man es auch nach Jahren noch verstehen, ausbessern und erweitern kann.

1.2 Programmentwurf

Zentraler Teil der Programmentwicklung ist der Programmentwurf und nicht – wie es manchem Anfänger scheinen mag – die Programmierung bzw. Codierung in einer Programmiersprache. Der Entwickler schlüpft hier in die Rolle des Systemdesigners. Ähnlich wie ein Architekt konstruiert er, geleitet von seinen Lösungsideen und der Machbarkeit, die Softwarestruktur (Softwaredesign). Er legt fest, wie die Aufgabe in Software umgesetzt wird. Eine umfassende Einführung in die Techniken des Programmentwurfs wäre für dieses Skript zu umfangreich, weshalb hier nur einige die wichtigsten Kriterien, denen ein Programm genügen sollte, aufgeführt werden.

- **Zuverlässigkeit und Korrektheit:** Für jede korrekte Eingabe wird das nach dem gewählten Algorithmus richtige Ergebnis abgeliefert. Falsche Eingaben werden, soweit möglich, vom Programm erkannt, abgefangen und gemeldet. Ausfälle der Hardware können abgefangen oder in ihren Auswirkungen stark gemildert werden (Datensicherung, Wiederaufsetzen von Aufträgen). Durch die Zerlegung des Problems in kleine Häppchen ist schon ein hohes Maß an Korrektheit gesichert, da die Teilprobleme übersichtlich darzustellen sind. Da für jeden Programmteil Eingabegrößen, Ausgabegrößen und der Lösungsweg bereits feststehen, da sie schon im Entwurf festgelegt worden, wird die „trickreiche“ Programmierung weitgehend verhindert. Durch strukturierte Programmierung allein kann man noch keine Zuverlässigkeit erreichen. Es ist schließlich möglich, die dümmsten Programme sauber zu strukturieren. Aber aufgrund der Strukturierung ist es einfach, Prüf- und Testalgorithmen einzufügen. Weiterhin kann die Datensicherung und der Neustart von Programmen durch definierte Übergänge zwischen den Programmteilen (Modulschnittstellen) erleichtert werden.
- **Benutzerfreundlichkeit:** Für den gegebenen Benutzerkreis stellt das Programm keine zu hohen Anforderungen. Fragen sind dem Benutzer verständlich und unvollständige Eingaben resultieren in Nachforderungen. Fehlermeldungen werden im Klartext ausgegeben und enthalten Hinweise auf die Behebung des Fehlers. Beispielsweise sollte eine Fehlermeldung nicht „ERROR EXIT 05 AT 54767“, sondern „Fehler in Funktion Nullstelle: Division durch 0“ lauten.
- **Flexibilität:** Wenn sich ein neuer Benutzerwunsch ergibt, so muss im Allgemeinen das Programm geändert werden. Bei einem gut strukturierten Programm beschränkt sich die Änderungsarbeit

meist auf wenige Teilprobleme, die Anpassbarkeit oder Adaptibilität ist hoch. Bei nicht strukturierten Programmen treten bei Änderungen oft Fehler an ganz anderen Stellen im Programm auf, weil dort beispielsweise Variablen für einen völlig anderen Zweck verwendet wurden. Andererseits soll es möglich sein, ohne große Schwierigkeiten die Hardware zu wechseln. In diesem Fall spricht man von Übertragbarkeit oder Portabilität.

- **Lesbarkeit:** Für die Lesbarkeit von Programmen ist entscheidend, dass
  - viele Kommentare im Text stehen,
  - der Programmtext sauber strukturiert ist (z. B. durch Einrückungen),
  - Hauptprogrammteile und Unterprogramme kurz sind,
  - aussagekräftige Namen für Unterprogramme verwendet werden,
  - im Hauptprogramm die Lösung jedes Teilproblems kurz und übersichtlich beschrieben ist und
  - für Details auf Unterprogramme verwiesen wird, welche die Teilprobleme in entsprechender Weise behandeln.

  Lesbarkeit ist Voraussetzung für Wartbarkeit!

- **Effizienz:** Die Effizienz eines Programms kann an der benötigten Laufzeit und an den Speicheranforderungen gemessen werden. Und kürzere Programme sind nicht nur schneller, sondern haben (rein statistisch betrachtet) auch weniger Fehler. Bei der Echtzeitverarbeitung von Daten spielt die Effizienz eine dominierende Rolle. Sie kann darunter leiden, dass C-Programme redundanter sind als z. B. in Assembler geschriebene und strukturierte Programme redundanter sind als nicht strukturierte. Gegenüber den Vorteilen, die sich mit der Strukturierung bezüglich der Flexibilität und der Sicherheit des Ablaufs bei der Ausführung der Programme ergeben, spielen die Effizienzverluste aber meist kaum eine Rolle. Auf das Kriterium der Effizienz darf erst dann geachtet werden, wenn eine fertige Version des Programms vorliegt, die allen anderen Anforderungen genügt. Erst dann sollten zusätzliche Effizienzverbesserungen, ausführlich erläutert, eingeführt werden.

Es ist klar, dass solche Anforderungen besondere Methoden der Programmentwicklung notwendig machen. Ausgehend von der Aufgabenstellung entsteht durch schrittweise Verfeinerung des Problems schließlich das fertige Programm. Die Vorgehensweise ist immer dieselbe, egal wie groß das Programm wird: Das Problem wird in einzelne Teilprobleme zerlegt. Die Kunst besteht darin, die Zerlegung derart zu gestalten, dass die Teilprobleme möglichst nicht voneinander abhängen (Prinzip der starken lokalen Bindung) und das Teilproblem in sich alleine – ohne Zuhilfenahme der Lösung anderer Teilprobleme der gleichen Zerlegungsstufe – zu lösen ist (Prinzip der möglichst losen Koppelung). Diese so gefundenen Teilprobleme werden weiter zerlegt und verfeinert, bis sie so leicht zu überschauen sind, dass sie problemlos programmiert werden können. Diese Methode des Top-Down-Design sichert sauber strukturierte und verifizierbare Programme oder Objekte. Beachten Sie dabei die folgenden Grundregeln:

- Jeder Zerlegungsschritt sollte auf einer, höchstens zwei DIN-A4-Seiten beschrieben werden.
- Alle Teilprobleme sollen etwa den gleichen Umfang besitzen.
- Eine klare Gliederung muss den Überblick garantieren.

So zeichnet sich eine Struktur von einzelnen Verfeinerungsstufen oder -schichten ab. Eine solche Schicht können Sie als Beschreibung des Problems in einem bestimmten Abstraktionsgrad verstehen. Sie können sich auch vorstellen, dass auf jeder Schicht ganz bestimmte, abstrakte Grundoperationen existieren. Die Grundoperationen werden von Schicht zu Schicht nach unten einfacher. Sie bewegen sich so von sehr komplexen Konstruktionen zu konkreten Operationen.

In der Informatik wird eine Schicht als abstrakte Maschine aufgefasst, also als ein gedachter Computer, der alle Operationen der entsprechenden Detaillierungsstufe beherrscht. Aus diesem Gedankenkonzept folgt sofort, dass auch die Verfeinerung schichtweise geschehen soll. Eine Schicht sollte daher erst vollständig aufgebaut sein, bevor die nächste Verfeinerung in Angriff genommen wird. Diese Vorgehensweise hat den wesentlichen Vorzug, dass alle Teilprobleme etwa den gleichen Abstraktionsgrad und ungefähr den gleichen Umfang haben. Es kann also nicht vorkommen, dass Sie sich einerseits um die Optimierung eines Algorithmus bemühen, während Sie in einem anderen Teilproblem noch nach dem Lösungsweg suchen.

Beim Programmentwurf wird festgelegt, wie die Aufgabe in Software umgesetzt werden soll. Die wichtigste Phase beim Erstellen eines Algorithmus ist die Analyse der Problemstellung und das Nachdenken über die Lösung. Dieser Teil ist wesentlich wichtiger und zeitaufwendiger als die Codierung in einer Programmiersprache. Die Methoden der strukturierten Programmierung müssen jedem Entwickler bekannt sein. Sie sind der Schlüssel, um auf effiziente Weise Software zu erstellen, die den Anforderungen genügt. Dazu gehören der Top-down-Entwurf mit schrittweiser Verfeinerung, die Unterprogrammtechnik und andere. Wichtige Kriterien für gute Programme sind Zuverlässigkeit und Korrektheit, Benutzerfreundlichkeit, Flexibilität, Lesbarkeit und Effizienz. Die Darstellung von Algorithmen erfolgt in der verbalen Beschreibung, dem Programmablaufplan oder dem Struktogramm.

1.3 Programmierung

Die Programmierung ist eine der letzten Stufen der Programmentwicklung. In diesem Schritt übernimmt der Entwickler die Rolle des Programmierers, er erstellt den Programmcode und die Dokumentation.

Programmieren heißt, den zeichnerisch, verbal oder sonstwie dargestellten Algorithmus in eine Programmiersprache umzusetzen und zu testen. Dabei werden die Schritte Codierung, Eingabe, Übersetzung und Testen zumeist wiederholt durchlaufen. Der Übersetzungslauf als gesonderter Schritt ist bei Sprachen mit Compiler, nicht aber bei solchen mit Interpreter erforderlich.

Zum Programmieren gehören essenziell die Programming Standards. Das sind Vorschriften und Regeln, an die sich alle an einem Projekt beteiligten Programmierer zu halten haben. Diese Regeln können Länge und Komplexität von Unterprogrammen betreffen, Bezeichnungskonventionen oder Schreibweisen (Einrückung, Leerzeilen, Kommentare, Dateinamen etc.), aber auch den Inhalt von Kommentaren (z. B. Verfasser, Erstellungsdatum, Änderungsdaten und eine genaue Beschreibung der Parameter) und anderes. Solche Regeln dienen der übersichtlichen Gestaltung von Programmen und damit auch wieder der Lesbarkeit.

Bedenken Sie immer, dass ein Programm zwar mehrfach geändert oder erweitert, jedoch wesentlich häufiger gelesen wird. Dann sollte der Leser – schlimmstenfalls ist man das selbst – auch nach Jahren schnell verstehen, worum es geht.

Wünschenswert ist die Aufteilung von Programmen in einzelne Module. In allen höheren Programmiersprachen und auch im Befehlsumfang nahezu aller Prozessoren ist die Möglichkeit der Modularisierung in Form von Unterprogrammen realisiert. Für die Verwendung von Unterprogrammen/Methoden (UP) sprechen weitere Gründe:

- **Übersichtlichkeit**
  Durch die Verwendung von UP steigen Lesbarkeit und Verständlichkeit des Programms, da sich UP als direkte Abbildung eines Funktionsblocks unter einem aussagekräftigen Namen ins Programm einfügen lassen. Zudem erlauben sie den vom Rest des Programms unabhängigen Test von einzelnen Funktionsblöcken und machen so auch große Programme überschaubar und (relativ) fehlerfrei.

- **Wirtschaftlichkeit**
  Häufig werden die gleichen Anweisungsfolgen für unterschiedliche Daten und an verschiedenen Stellen im Programm benötigt. Durch ein UP kann eine Anweisungsfolge definiert werden, die dann mehrfach und mit unterschiedlichen Daten aufgerufen werden kann. Dadurch wird der Code kürzer und der Speicherbedarf wird entsprechend geringer.

- **Änderungsfreundlichkeit**
  Änderungen (etwa aufgrund neuer Hardware) und Optimierungen betreffen immer nur einige wenige Unterprogramme. Da die Verbindung zum Rest des Programms über eine fest definierte Schnittstelle erfolgt, wirken sich Änderungen in einem UP normalerweise nicht auf den übrigen Programmcode aus. Voraussetzung dafür sind möglichst voneinander unabhängige Unterprogramme.

- **Lokalität**
  Die lokale Begrenztheit von Variablen kann durch die Vereinbarung einzelner Variablen im UP hervorgehoben und unterstützt werden. Viele Hochsprachen erlauben es, solche Variablen in ihrer

Gültigkeit auf das UP zu beschränken. Wird der gleiche Variablenname in verschiedenen Unterprogrammen verwendet, ergeben sich keine Konflikte der lokalen Variablen untereinander.

- **Allgemeingültigkeit**
  Durch die Erarbeitung möglichst allgemein gültiger Unterprogramme kann eine Unterprogrammbibliothek erstellt werden, deren Inhalt immer wieder bei Programmierproblemen herangezogen werden kann. Die Zeit zum Erstellen neuer Programme wird so verkürzt. Sind die Bibliotheksroutinen sorgfältig getestet, sinkt auch die Fehlerrate bei neu erstellten Programmen. Vielfach kann ein Entwickler auch Bibliotheken zur Lösung spezieller Probleme kaufen.

- **Programmentwicklung im Team**
  Von mehreren Mitarbeitern entwickelt jeder für sich Unterprogramme mit fest definierten Schnittstellen für die Ein- und Ausgabe von Daten. Die Unterprogramme werden später zum Gesamtprogramm zusammengeführt.

Etliche der oben aufgeführten Kriterien werden durch die Objektorientierung noch verbessert und erweitert.

Sowohl beim Entwickeln des Algorithmus als auch beim Programmieren schleichen sich Fehler ein. Der Softwarebenutzer erwartet aber ein fehlerfreies Produkt. Streckenweise ist dies sogar überlebenswichtig. Beispielsweise dürfen eine Avioniksoftware oder eine Software zur Steuerung von Geräten der Intensivmedizin keine Fehler aufweisen, denn deren Auswirkungen könnten Menschenleben gefährden. Ziel einer jeden Softwareentwicklung muss es demnach sein, ein möglichst fehlerfreies Produkt zu erstellen. Die Fehlerfreiheit gehört zu den Produkteigenschaften, die dem Nutzer zugesichert wurden. Fehlerfreiheit ist somit ein ein Qualitätsaspekt.

Als die Top 10 im Bereich „Internet of Things“ listete 2014 das Open Web Application Security Project (OWASP) die folgenden Fehler auf:

1. Insecure Web Interface
2. Insufficient Authentication/Athorization
3. Insecure Network Services
4. Lack of Transport Encryption
5. Privacy Concerns
6. Insecure Cloud Interface
7. Insecure Mobile Interface
8. Insufficient Security Configurability
9. Insecure Software/Firmware
10. Poor Physical Security

Diese Top 10 sind eigentlich alle leicht vermeidbare „Anfängerfehler“ und fast alle schon durch den Schreibtischtest (s. u.) aufzuspüren.

Softwarequalität ist somit die Gesamtheit der Merkmale und Merkmalswerte eines Softwareproduktes, die sich auf dessen Eignung beziehen, festgelegte und vorausgesetzte Erfordernisse zu erfüllen (nach DIN ISO 9126). Jede Abweichung vom gewünschten Ergebnis ist ein Fehler.

Ideal wäre es, Fehler zu vermeiden, bevor sie entstehen. Prävention ist eine Strategie, die sich auszahlt, denn Fehler entstehen an den verschiedensten Stellen bei der Softwareentwicklung. Es beginnt bei der Anforderungsdefinition, die schon fehlerhaft sein kann. Es ist also wichtig, mit dem Kunden und dem Softwarenutzer zu reden und die Anforderungen möglichst genau und widerspruchsfrei zu erfassen. Weitere Fehler entstehen in der Spezifikation, beim Entwickeln des Algorithmus und bei der Programmierung.

Wie findet man nun die Fehler, die man vermeiden möchte? Dazu gibt es verschiedene Strategien: Zum einen gibt es das konstruktive Qualitätsmanagement. Hierbei legt man im Vorfeld den Rahmen für den einzelnen Softwareentwickler fest. Dies geschieht mittels Vorgaben, wie z. B. Programmierrichtlinien, Checklisten und den Einsatz von entsprechenden Werkzeugen, Programmiersprachen

und Methoden. Gewisse Tricks in der Programmierung werden untersagt sowie Art und Umfang der Kommentierung des Quellcodes vorgegeben. Außerdem wird eine Programmiersprache gewählt, die zur Aufgabe passt und dem Ausbildungsstand der beteiligten Programmierer entspricht.

Zum anderen prüft man beim analytischem Qualitätsmanagement mittels Tests, ob das Geschaffene mit dem übereinstimmt, was vorgegeben war. Was vorgegeben ist, richtet sich nach der Softwareentwicklungsphase, in der man sich gerade befindet. Wie getestet werden kann, richtet sich nach den Möglichkeiten und dem Aufwand, den man investieren möchte. Bleiben wir doch gleich beim Testen.

Schon während der Programmentwurfsphase wechselt der Entwickler in die Rolle des Testplaners und überlegt sich die notwendigen Vorgehensweisen. Der Qualität förderlich ist es natürlich, wenn eine andere Person die Software-Quellen im Rahmen einer Peer- Review begutachtet und die Testprotokolle erstellt. Das Testen erfolgt nicht nur als Computertest, sondern auch als Schreibtischtest. Der Compiler hilft uns zwar, syntaktische Fehler zu finden; logische Fehler im Programm lassen sich aber nur durch Nachdenken und Schreibtischtests herausfinden.

Der **Schreibtischtest** simuliert den Einsatz des Algorithmus im Kopf des Entwicklers. Anhand einer sinnvollen Auswahl von Eingabedaten wird händisch das richtige Funktionieren des Algorithmus Schritt für Schritt nachvollzogen. „Richtig“ heißt hier, dass in jedem Schritt der Algorithmus das tun muss, was der Erfinder gewollt hat und natürlich auch die bekannte richtige Ausgabe liefert. Der Schreibtischtest ist heute obligatorisch für jeden Softwareentwickler. Jedes Stück neue Software wird in dieser Form getestet, denn jede Teilsoftware hat definierte Eingabedaten und muss in irgendeiner Form bekannte Ausgabedaten erzeugen. Viele Denkfehler werden dabei aufgedeckt und können aufwandsarm beseitigt werden. Aber: Es ist auf gar keinen Fall der einzige Test!

Zusätzlich zum Schreibtischtest lassen sich Softwaretests auch automatisiert durchführen. Hierbei kommt je nach Art der Software und Testziel eine Vielzahl verschiedener Werkzeuge zum Einsatz.

In jedem Fall müssen sinnvolle Testfälle bekannt sein. Das heißt, für einen Satz Eingabedaten muss das Ergebnis, die Ausgabedaten, bekannt sein. Nur dann kann ein Test auf richtiges oder falsches Funktionieren des Algorithmus und seiner Implementierung als Computerprogramm überhaupt durchgeführt werden. In der Regel können aufgrund der Komplexität nicht alle möglichen Variationen der Eingabedaten für Tests verwendet werden. Die sinnvolle Auswahl der Testfälle ist entscheidend. Darüberhinaus gibt es unterschiedliche Sichtweisen, was getestet werden soll. Hier zwei Beispiele:

Der **Black-Box-Test** ist ein sogenannter funktionaler Test. Er betrachtet das Softwaresystem (Algorithmus) als schwarzen Kasten, in den man nicht hineinschauen kann. Der Anwender einer Software ist ein klassischer Black-Box-Prüfer. Er ist nur an den versprochenen Funktionen interessiert, aber wie die Software intern funktioniert, ist ihm egal. Also werden im Black-Box-Test alle vereinbarten Funktionen durch systematische Benutzung mit sinnvollen Testfällen aus Anwendersicht durchgespielt. Dies ist in der entsprechenden Spezifikation im Vorhinein niedergelegt. Der Black-Box-Test ist der klassische Abnahmetest. Der Kunde, der Software kauft, testet sie nach der Lieferung, oftmals im Beisein des Entwicklers. Auch in der Entwicklung selbst wird dieses Verfahren häufig eingesetzt. Jeder Algorithmus und jede Programmfunktion ist in der Regel schriftlich spezifiziert. Somit testet der Entwickler sein Programm gegen die Spezifikation mit daraus abgeleiteten Testfällen.

Problematisch beim Black-Box-Testverfahren ist die sogenannte Testüberdeckung. Die Interna des Programms werden nicht berücksichtigt. So kann es vorkommen, dass gewisse Programmteile, wie Schleifen oder Abfragen, bei solchen Tests nie oder immer mit „harmlosen“ Daten durchlaufen werden. Sinnvolle Testfälle, die sich nicht aus der Verwendung, sondern einzig aus dem Aufbau und der Formulierung des Programms ergeben, werden beim Black-Box-Test nicht systematisch durchgeführt. Die Konsequenz: unentdeckte Fehler. Um dies zu vermeiden, gibt es das Prinzip des wesentlich aufwendigeren White-Box- Tests.

Ausgehend vom Programmquelltext wird im **White-Box-Test** der Kontrollfluss des Programms untersucht und daraus werden Testfälle abgeleitet. Der Kontrollfluss ist der Weg durch das Programm, die Abfolge der Befehle bei der Ausführung durch den Rechner. Alle möglichen Wege mit allen möglichen Datenkombinationen zu testen, ist in der Regel zu aufwendig. Eine Variante des White-Box-Tests sieht vor, dass die Testfälle so generiert werden, dass zumindest alle Programmanweisungen mindestens einmal durchlaufen werden. Dieser wird Anweisungsüberdeckungstest (C0-Test) genannt. Durch Verzweigungen im Programm kann eine Anweisung auf mehreren Wegen erreicht werden. Der Anweisungsüberdeckungstest nimmt hier keine Rücksicht. Hauptsache, jede Anweisung wurde einmal durchlaufen.

Beim sogenannten Zweigüberdeckungstest (C1-Test), werden Testfälle so konstruiert, dass alle Zweige durchlaufen werden. Dieser ist aufwendiger als der Anweisungsüberdeckungstest. Aber selbst dieser reicht nicht aus, um alle Fälle abzudecken. Schleifen und datenabhängige Verzweigungen führen zu unterschiedlichen Wegen durch das Programm, diese werden Pfade genannt. Die Kombination der beschrittenen Zweige hat unterschiedliche Auswirkungen und kann somit Fehlerquelle sein. Dieses berücksichtigen die sehr aufwendigen Pfadüberdeckungstests (C2-Tests).

Selbst hier ist noch nicht alles getan. Die datenabhängigen Verzweigungen haben meist komplexe Entscheidungsbedingungen, beispielsweise `if (A && B || C) { ... } else { ... }`. Das Ergebnis der Bedingung `(A && B || C)` hängt von den Ergebnissen der Bewertung der Teilausdrücke A, B und C ab. Eigentlich müssten für alle möglichen Wertekombinationen dieser Teilausdrücke die Verzweigung getestet werden. Im Pfadüberdeckungstest wird hierauf keine Rücksicht genommen. Erst der Bedingungsüberdeckungstest (C3-Test) kümmert sich darum.

Betrachtet man nun noch die möglichen Daten, die ein Programm erzeugt und die es eingespeist bekommt, so finden noch die datenflussorientierten Testverfahren Anwendung. Statt nur den Kontrollfluss zu betrachten, wird hier der Datenfluss mit den möglichen Werten und Wertebereichen der einzelnen Variablen im Programm untersucht. In der Praxis wird man sich für eine sinnvolle und realisierbare Mischung aus den vorgestellten White-Box-Testverfahren und dem Black-Box-Testverfahren entscheiden.

Bei der Auswahl der Testdaten kann man sich von folgenden Gesichtspunkten leiten lassen:

- Wählen Sie die Testdaten so, dass das Programm für alle Aufgaben, die in der Spezifikation gefordert werden, getestet wird.
- Wählen Sie die Testdaten so, dass jeder Zweig des Programms mindestens einmal durchlaufen wird.
- Wählen Sie spezielle Testdaten derart, dass alle Sonderfälle des Programms erfasst werden.
- Wählen Sie zufällige Testdaten, auch wenn sie nicht sinnvoll erscheinen, um Lücken im Test aufgrund scheinbarer Selbstverständlichkeiten zu vermeiden.

2

Besonderheiten der Embedded-Programmierung

Mikrocontrollersysteme bzw. eingebettete Systeme werden je nach Anwendungsfall und Ausstattung des Controllerboards mit oder ohne Betriebssystem konzipiert. Falls ein Betriebssystem vorhanden ist, kommen neben Linux und embedded-Varianten von Windows hauptsächlich spezielle Echtzeitbetriebssysteme zum Einsatz. Systemnahe Programme (z. B. Treiber) werden oft noch in maschinennahen Sprachen (Assembler), zunehmend aber in Hochsprachen entwickelt. Viele Assemblersprachen erlauben auch Datendefinitionen (Datentypen, Überprüfung von Bereichsgrenzen, Konstantendefinition) und bieten über Makros vordefinierte Kontrollstrukturen, wie z. B. Schleifen, bedingte Anweisungen etc., die in Grenzen eine strukturierte Programmierung erlauben.

Zu Anfang der Mikroprozessortechnik gab es einige prozessorabhängige Implementierungssprachen, z. B. PL/M, die heute durch Standardsprachen abgelöst wurden. Die wichtigste höhere Programmiersprache für embedded Systeme ist nach wie vor C. Diese Sprache ersetzt Assembler zunehmend und hat heute einen Nutzungsanteil von 90 %. Inzwischen gibt es so gute C-Compiler, dass man gegenüber einer Codierung in maschinennahen Sprachen nur noch mit anderthalbfachem Speicher und auch etwa anderthalbfacher Ausführungszeit rechnen muss. Die bessere Übersichtlichkeit, die Übertragbarkeit auf andere Mikroprozessoren, die verkürzte Entwicklungszeit sowie steigende Rechenleistung und Speicherkapazität machen C immer attraktiver. Selbst objektorientierte Sprachen wie C++ beginnen für den Controllersektor interessant zu werden.

Für bestimmte Aufgaben gibt es speziell angepasste Programmiersprachen, die sich an den Erfahrungen der Anwender orientieren. Beispiele dafür sind Steuerungssprachen für speicherprogrammierbare Steuerungen (SPS) und Blockdiagrammsprachen, z. B. zur Beschreibung von Reglern oder dynamischen Systemen sowie Systeme zur Messdatenerfassung.

Eine zunehmende Rolle spielen Sprachen, die zur funktionalen Spezifikation und Modellbildung dienen, aus denen aber zunehmend auch Code erzeugt werden kann. Beispiele hierfür sind Matlab/Simulink oder Rose/RealTime.

Im Vergleich zum „normalen“ PC hat ein Controller sehr wenige Ressourcen. Beispielsweise besitzt ein sehr beliebter Controller, der Atmel ATMega 128 keine Floating Point Unit und er läuft „nur“ mit 16 MHz. Die Wortbreite ist 8 Bit, ab Speicher hat er 4 KByte RAM und 128 KByte Flash-Speicher. Aber die Ressourcen gehören einem einzigen Programm (es gibt ja kein Betriebssystem).

Das hat natürlich Auswirkungen auf die Programmierung: Der Programmierer muss:

- abwägen zwischen Rechenkosten und Speichernutzung
- effizient Programmieren z. B. durch Ausnutzung mathematischer Eigenschaften:
  - Bitshifting,
  - boolsche Verknüpfungen auf Bitebene,
  - Transponieren von Gleitpunkt- auf Integerarithmetik,

  - Tabellenzugriff statt Reihenentwicklung
  - usw.
- so allgemein und erweiterbar wie nötig und so gradlinig wie möglich programmieren

Mikrocontroller zeichnen sich gegenüber dem PC auch durch eine Vielzahl hardwarenaher Schnittstellen aus. Insbesondere stehen Interrupts und Timer zur Verfügung, an die man beim PC nicht so einfach heran kommt, weil einem da das Betriebssystem „im Weg steht".

Über Interrupts können bestimmte Teile des Codes durch externe Ereignisse ausgeführt werden. Diese können dabei beispielsweise von den Timern ausgelöst werden. Der momentane Programmablauf wird unterbrochen und der Code der Interrupt-Serviceroutine wird ausgeführt. Anschließend wird der normale Programmablauf fortgesetzt. Der Programmablauf wird in gewisser Weise weniger vorhersehbar. Deshalb müssen Stellen im normalen Programmablauf, die nicht unterbrochen werden dürfen, durch Sperren einzelner oder aller Interrupts geschützt werden. Das Problem dabei ist, dass man im ungünstigen Fall einen Interrupt verpasst. Variablen, die im normalen Programmablauf und in der Interrupt-Serviceroutine genutzt werden, müssen als volatile gekennzeichnet sein (`volatile __u8 Foo;`). Auch sollten die Interrupt-Serviceroutinen möglichst kurz sein.

Ohne Interrupts müssen entsprechende Events durch „polling" (regelmäßiges Abfragen) detektiert werden. Dabei gilt:

- Besonderes Augenmerk ist auf zeitkritische Ereignisse zu richten.
- Ein mögliches Modell: Die Hauptschleife in ausreichend große Zeitscheiben einteilen, die kontinuierlich durchlaufen werden.
- Alternative: Einen Scheduler implementieren, der die verschiedenen Aufgaben organisiert (aufwendiger).
- Das Timing planen
  - Gibt es mehrere periodisch Abläufe, müssen sie strukturiert werden
  - Einfache Nebenläufigkeiten (Ereignisse) lassen sich oft durch Interrupts lösen

Das folgende Listing zeigt ein typisches Grundgerüst eines C-Programms auf einem Mikrocontroller:

```
#include ...

void init(void)
  {

  // hier alle Initialisierungen

  }

int main(void)
  {
  init();

  while (1)
    {

    // Programmcode

    }

  return 0;
  }
```

Bei mehreren periodischen Abläufen in der `while(1)`-Schleife können diese mit Hilfe des größten gemeinsamen Teilers der Perioden strukturiert werden, was wieder für Linux-PC und Controller gilt. Beim Linux-Rechner ist „Busy Wait" per Schleife (z. B. `for (i = 0; i < 500; i++);`) zu vermeiden. Solche Konstruktionen führen dazu, dass der Prozess extrem viel Rechenzeit aufnimmt. Viel besser sind da eingestreute Wartezeiten (`nanosleep`).

Weiterführende Informationen allgemeiner Art:
https://www.cs.arizona.edu/ mccann/cstyle.html
ftp://ftp.cs.toronto.edu/doc/programming/ihstyle.ps

2.1 Variablen

Die Kenntnis des Wertebereiches einer Variablen ist entscheidend für das Verständnis eines Algorithmus. Ohne explizite Angabe des Wertebereiches ist es meistens schwierig festzustellen, welche Art von Objekten eine Variable repräsentiert und die Ermittlung möglicher Fehler wird erheblich erschwert. Zweckmäßigkeit und Korrektheit eines Programms sind in den meisten Fällen abhängig von den Anfangswerten der Argumente, und diese sind nur für bestimmte Bereiche garantiert. Der Speicherbedarf für die Repräsentation einer Variablen im Speicher eines Computers hängt von deren Wertebereich ab. Damit ein Compiler die nötigen Speicherzuordnungen vornehmen kann, ist die Kenntnis der Wertebereiche unerlässlich. Operatoren, die in Ausdrücken vorkommen, sind nur für gewisse Wertebereiche ihrer Argumente wohldefiniert. Ein Compiler kann aufgrund der angegebenen Wertebereiche prüfen, ob die vorkommenden Kombinationen von Operatoren und Operanden zulässig sind.

Bei Mikrocontroller gibt es in der Regel keine Floating Point Unit, was bedeutet, dass Gleitpunktoperationen per Software emuliert werden müssen → Verbrauch von Programmspeicher und Rechenzeit. Aber auch bei Ganzzahlwerten kann man etwas optimieren. Dort gelten zwei einfache Faustregeln:

- wo immer es möglich ist, positive ganze Zahlen verwenden (`unsigned int`)
- Variablentypen so klein wie möglich halten, also z. B. `unsigned char` statt `unsigned int` verwenden.

Gerade bei Zugriff auf 8-Bit-Peripherie macht `int` keinen Sinn – außer, dass der Compiler 16-Bit-Operationen erzeugt und so nutzlosen Binärcode erzeugt. Dazu ein Beispiel, das eine 8-Bit-Zufallszahl mit der oben erwähnten Formel erzeugt. Die Ermittlung des Divisionsrestes aus der Division durch *M* wird implizit durch einen Datentyp-Überlauf erreicht.

```
#define RAND_MULTIPLIER  19
#define RAND_CONSTANT    37

/* Globale Variable fuer die "Seed"*/
static unsigned char LastRandom = 1;

/* Setzen Anfangswert, falls gewuenscht */
void RandSeed(unsigned char Seed)
  {
  LastRandom = Seed;
  }

/* Zufallsfunktion */
unsigned char Rand()
  {
  /* kopieren Makros in lokale Variablen,
     um 16-Bit-Expansion zu vermeiden     */
  unsigned char Multiplier = RAND_MULTIPLIER;
  unsigned char Constant = RAND_CONSTANT;

  /* implizite mod-256-Operation durch 8-Bit-Datentyp-Ueberlauf */
  LastRandom = (LastRandom * Multiplier + Constant);
  return LastRandom;
  }
```

Die wichtigste Regel für Gleitkommazahlen (float, double) lautet:

- Gleitkommazahlen vermeiden, wo immer es geht

Oft lassen sich Gleitkomma-Operationen in den Ganzzahlbereich transponieren. Nehmen wir als Beispiel einen A/D-Wandler, der einen Wert zwischen 0 und 4095 liefert (12-Bit-Wandler). Aufgrund des Sensors entspricht bei einer maximalen Eingangsspannung von 5 V ein Schritt des Wandlers einem Wert von 0,00122 V. Der Gleitpunkt-Ansatz würde nun lauten:

```
Volt = (float) ADC_Wert*0.00122;
```

Es soll eine ganze Zahl berechnet werden, die auf Vielfachen einer Float-Zahl beruht. Zuerst wird der Faktor für 1 V ermittelt: $4096/5V = X/1V$, was 819.2 für X liefert. Gerundet auf die nächste ganze Zahl ergibt sich 819 (der Fehler liegt dabei unter einem Promille).

Wird nun das Ergebnis des A/D-Wandlers durch 819 geteilt, erhält man den ganzzahligen Anteil der gemessenen Spannung. Im Rest der Division verbirgt sich der gebrochene Anteil. Dieser Rest wird nun mit 10 multipliziert und anschließend wieder durch 819 dividiert. Das Resultat sind die Zehntelvolt der Messung. Der nun vorhandene Divisionsrest wird wieder mit 10 multipliziert und durch 819 dividiert und so fort, bis die gewünschte Anzahl Nachkommastellen erreicht ist.

2.2 Typecast

Wenn man den Datentyp einer Variablen verändern will, kann man das mit einer expliziten Typumwandlung (cast-Operator) erreichen. Wenn man beispielsweise eine Variable vom Typ `int` in eine vom Typ `float` umwandeln will, würde man `floatvar = (float)intvar;` schreiben. Die Typwandlung mittels cast-Operator kommt in der Praxis recht häufig vor. Sie kann in beide Richtungen angewendet werden:

- von kleinerem Typ zum größeren Typ ohne Datenverluste
- von größeren zum kleineren Typ mit Datenverlust.

So werden beim cast von `int` nach `char` immer die höherwertigen Bytes abgeschnitten. Bei `float` nach `integer` wird der Dezimalbruch abgeschnitten. Speziell bei der Umwandlung von `unsigned int` in `signed int` und umgekehrt ist der Typecast erforderlich, wenn man nicht seltsame Effekte erhalten will, die auf der Komplementdarstellung negativer Zahlen beruhen.

2.3 Speicherklassen

Bei der Programmentwicklung für Mikrocontroller können auch die Speicherklassen der Sprache C eine wichtige Rolle spielen – insbesondere die Klassen „volatile“ und „static“.

static definiert, wie der Name schon sagt, „statische“ Variable, deren Inhalt in Funktionen zwischen zwei Aufrufen erhalten bleibt (initialisiert mit 0). Genauer gesagt, wird der Speicher solcher Variablen so allokiert, dass Ihre Lebensdauer nicht durch die Sichtbarkeit (Scope) der Variablen beschränkt wird.

- static-Objekte können sowohl intern als auch extern sein
- static-Objekte innerhalb von Funktionen sind nur lokal bekannt, behalten im Gegensatz zu auto-Objekten aber ihre Werte zwischen den Funktionsaufrufen bei.
- Bezüglich der Initialisierung gilt dasselbe wie für externe Objekte, static-Vektoren sind daher initialisierbar.
- Zeichenketten innerhalb von Funktionen sind immer von der Speicherklasse static.
- static-Objekte außerhalb von Funktionen sind externe Objekte, deren Namen aber nur in dieser Quellencodedatei bekannt ist.

Dazu ein Beispiel. Man kann die Random-Funktion des Beispiels oben wie im folgenden Programm umschreiben. `LastRandom` ist nun als statische Variable in der Funktion`Rand()` definiert und trägt die „Saat“ von Aufruf zu Aufruf weiter:

```
unsigned char Rand()
  {
  static unsigned char LastRandom;

  /* kopieren Makros in lokale Variablen,
     um 16-Bit-Expansion zu vermeiden     */
  unsigned char Multiplier = RAND_MULTIPLIER;
  unsigned char Constant = RAND_CONSTANT;

  /* implizite mod-256-Operation durch 8-Bit-Datentyp-Ueberlauf */
```

```
    LastRandom = (LastRandom * Multiplier + Constant);
    return LastRandom;
    }
```

volatile weist den Compiler darauf hin, dass der Wert dieser Variable von außerhalb verändert werden kann. Ein externer Prozess kann z. B. eine Interrupt-Serviceroutine sein, aber auch ein E/A-Port ändert seinen Wert durch äußere Gegebenheiten. Die Speicherklasse volatile garantiert, dass bei jedem Lesezugriff auf diese Variable der Wert immer erneut ausgelesen wird. Der Compiler wird ja versuchen, den erzeugten Code zu optimieren. Wenn nun aus Sicht des Compilers die Variable eine Konstante zu sein scheint (sie wird ja nirgendwo durch den C-Code verändert), ersetzt er die Speicherzugriffe durch Konstanten – mit entsprechend fatalen Folgen. Um dies zu verhindern, verwendet man die Speicherklasse volatile.

Ports werden beispielsweise als volatile Variablen mit konstanter Adresse definiert (vorausgesetzt, die Ports liegen im Adressraum und nicht in einem separaten E/A-Bereich):

```
/* Macros fuer einfache Portdefinitionen als Pointer
   Beispiel: #define foo PORT_u32(1234) */

#define PORT_u8(x)  (*(volatile u8*)(x))
#define PORT_u16(x) (*(volatile u16*)(x))
#define PORT_u32(x) (*(volatile u32*)(x))
#define PORT_u64(x) (*(volatile u64*)(x))
```

register -Variablen werden möglichst in den Registern der CPU angelegt. Dadurch kann auf solche Variablen besonders schnell zugegriffen werden. Dieses Schlüsselwort ist bei modernen Compilern weitgehend überflüssig, da die entsprechenden Optimierungen schon vom Compiler vorgenommen werden.

auto wird in der Regel implizit verwendet (nie hingeschrieben). Lokale Variablen von Funktionen sind in aller Regel sogenannte automatische Variablen. Sie werden automatisch allokiert, wenn ein Block bzw. eine Funktion betreten wird, und danach wieder entfernt.

extern bezeichnet Symbole, die im ganzen Programm bekannt sind (genauer: in dem Block, in der die Deklaration steht). In unterschiedlichen Blöcken stehende Deklarationen beziehen sich auf das gleiche Symbol! Obgleich das Datum global zugreifbar ist, ist der Gültigkeitsbereich auf den deklarierenden Block begrenzt.

const -Typen können nicht mehr verändert werden und gelten als konstant. Bei vielen Compilern für Embedded-Systeme sorgt das Schlüsselwort „const“ dafür, dass Konstanten (wenn möglich) im billigen Programmspeicher (FLASH, ROM) anstatt im teuren (knapperen) Arbeitsspeicher (RAM) des Systems abgelegt werden. Das funktioniert bei Konstanten, die feste Werte haben, beispielsweise bei Timerwerten oder Schleifendurchläufen. Bekommt jedoch eine Konstante beim Anlegen derselben einen Wert aus z. B. einer Funktion zurückgeliefert, so kann der Compiler diese Konstante natürlich nicht im ROM ablegen.

3

Bitmanipulation und -verknüpfung

3.1 Bitmanipulation

Bitmanipulation und Bitoperatoren dienen u. a. zur Konfiguration der Peripheriekomponenten, zum Setzen, Rücksetzen und Testen einzelner Bits usw. Der „normale“ C-Programmierer benötigt sie fast nie, weshalb hier etwas näher darauf eingegangen wird.[1] Im Gegensatz zu den logischen Operatoren in C werden hier jeweils die einzelnen Bits stellenweise miteinander verknüpft – fast so wie in entsprechenden Assemblerbefehlen. Die Sprache C kennt für Bitmanipulation die folgenden Operatoren:

- `|` binäre Oder-Verknüpfung
- `&` binäre Und-Verknüpfung
- `^` binäre Exor-Verknüpfung
- `~` binäre Negation
- >> Rechts schieben
- << Links schieben

Als **Bitmaske** bezeichnet man eine Folge von einzelnen Bits, die den Zustand Null (0) oder Eins (1) darstellen können. Bitmasken werden im allgemeinen dazu verwendet, um unter Anwendung eines Operators eine Eingabe zu manipulieren. Das Ergebnis ist dann die Anwendung des Operators auf die Eingabe und der Bitmaske. Die Bitmaske ist häufig eine Konstante. Die Bitmaske muss die gleiche Länge wie die zu manipulierende Variable haben. Das Erzeugen einer Bitmaske kann auf mehrere Arten erfolgen:

```
// Variable für Bitmaske
uint8 Bitmaske;

// Definition in Binärschreibweise
Bitmaske = 0b00010100;

// Definition mit Schiebeoperationen
Bitmaske = ((1<<2) | (1<<4));
```

Die Schiebeoperation (`1<<n`) verschiebt 1 um n Binärstellen nach links (siehe auch weiter unten: Schiebebefehle). (`1<<2`) ergibt somit 0b000000100 und (`1<<4`) 0b00010000. Durch die Oder-Verknüpfung erhält man die Bitmaske 0b00010100.

[1] In allen Beispielen wird der Datentyp „uint8“ verwendet, also ein 8-Bit-Wort. In C kann er beispielsweise mit *unsigned char* oder *__u8* definiert werden.

3.2 Bitverknüpfungen

Wie das obige Beispiel zeigt, lassen sich einzelne Bits durch Oder-Verknüpfung mit einer Bitmaske **setzen** (Var enthalte den Wert 0b10000001):

```
// Festlegung der Bitmaske
Bitmaske = 0b01010100;

// Ausgeschrieben
Var = Var | Bitmaske;

// Kurzschreibweise
Var |= Bitmaske;

//-> Ergebnis Var = 0b11010101
```

Mit der Und-Verknüpfung lassen sich einzelne Bits auf 0 setzen. Überall dort, wo die Bitmaske eine 0 enthält, wird das Verknüpfungsergebnis auf jeden Fall 0. An den Stellen, wo in der Bitmaske eine 1 steht, bleiben die ursprünglichen Werte erhalten (Var enthalte den Wert 0b11110000):

```
// Festlegung der Bitmaske
Bitmaske = 0b01010100;

// Ausgeschrieben
Var = Var & Bitmaske;

// Kurzschreibweise
Var &= Bitmaske;

//-> Ergebnis Var = 0b01010000
```

Oft wird zum Setzen und Rücksetzen von Bit dieselbe Bitmaske verwendet. In diesem Fall muss man beim Rücksetzen (Und-Verknüpfung) die Werte der Bitmaske invertieren.

```
// Festlegung der Bitmaske
Bitmaske = 0b00000100;

// Setzen
Var = Var | Bitmaske;

// Rücksetzen
Var = Var & ~Bitmaske;

// die negierte Bitmaske hat den Wert 0b11111011
```

Will man ein Bit unabhängig von seinem aktuellen Wert umkehren ($0 \rightarrow 1$, $1 \rightarrow 0$), verwendet man das Exklusiv-Oder. Steht eine 0 in der Bitmaske, bleibt das entsprechende Bit unverändert, bei einer 1 wird es invertiert (Var enthalte den Wert 0b10101010):

```
// Festlegung der Bitmaske
Bitmaske = 0b00001111;

// Toggeln
Var = Var ^ Bitmaske;

//-> Ergebnis Var = 0b10100101
```

Will man prüfen ob ein oder mehrere Bits in einer Variable gesetzt oder gelöscht sind, muss man sie mit einer Bitmaske Und-verknüpfen. Die Bitmaske muss an den Stellen der zu prüfenden Bits eine 1 haben, an allen anderen eine 0. Ist das Ergebnis gleich Null, sind alle geprüften Bits gelöscht, ist das Ergebnis ungleich Null, ist mindestens ein geprüftes Bit gesetzt und ist das Ergebnis gleich der Bitmaske, sind alle geprüften Bits gesetzt.

3.3 Schiebeoperationen

Die Schiebeoperationen erlauben bitweises Schieben nach links oder rechts. Die Anzahl der geschobenen Bit-Positionen stehen rechts vom Operanden. Von rechts rückt ein 0-Bit nach und von links rückt bei ganzen Zahlen (int) das Vorzeichen nach, bei vorzeichenlosen Zahlen (unsigned int) die 0. Beispiele:

```
1 << 1 ergibt 2
1 << 2 ergibt 4
1 << n ergibt 2 hoch n
x << 1 ergibt 2*x
x >> 1 ergibt x/2
```

Achtung: Die Shift-Operatoren haben eine niedrigere Priorität als * und +! Es sind also sinnvollerweise Klammern zu setzen. So hat `1<<10 - 1` als Ergebnis 2^9, denn der Compiler sieht implizit den Ausdruck `1<<(10 - 1)` und nicht etwa $2^{10} - 1$. Es sind also Klammern notwendig: `(1<<10) - 1`.

C-Ausdrücke mit der Definitionen von Bitwerten durch einen Schieboperator (z. B. `1<<4` für den Wert 0x10) sehen auf den ersten Blick ein Wenig abschreckend aus, funktionieren aber universell und sind manchmal deutlicher und nachvollziehbarer als Konstanten. Andererseits kann man durch Makrodefinitionen den Werten „sprechende“ Namen geben. Bei eingeschalteter Optimierung lösen die meisten Compiler solche konstanten Ausdrücke bereits zur Compilierungszeit auf und es entsteht kein zusätzlicher Maschinencode.

Das folgende Beispiel implementiert einen (Pseudo-)Zufallszahlengenerator auf Basis der Formel $X_{i+1} = (a \cdot X_i + c) \ mod \ p$. Hierfür werden ein (konstanter) Multiplikator a, ein (konstanter) Summand c und ein Anfangswert X_0 benötigt. Damit man keine Integer-Arithmetik braucht, wird die Formel realisiert, indem man sich die 16-Bit-Zahl X als Polynom von Zweierpotenzen vorstellt. Dadurch werden nur Schiebebefehle und logische Verknüpfungen benötigt:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Definitionen fuer den Zufallszahlengenerator
// POLY erzeugt GF(2^15) = GF(2)[x]
// POLY = 1 + x^2 + x^5 + x^8 + x^13 + x^14 + x^15
#define POLY 0xe125
#define DEGREE 15
// ROOT ist ein zyklischer Erzeuger von GF(2^15)^*
// ROOT = x^9 + x^13
#define ROOT 0x2200
#define MASK(b) (((unsigned short) 1) << (b))


// ************************************************
// Ein Algorithmus zur Errechnung einer Zufallszahl.
// ************************************************


// Arithmetik in GF(2)[x] / p(x)*GF(2)[x]
// Berechnet c = a*b mod p

unsigned short pprod (unsigned short a, unsigned short b)
  {
  const unsigned short mask = MASK (DEGREE);
  unsigned short c = 0;
  do
    {
    // Ist Bit i in b gesetzt?
    if (b & 1) c ^= a;           // dann c = c + a * x^i
    a <<= 1;                     // a = a*x
    if (a & mask) a ^= POLY;     // a = a mod p
    b >>= 1;
    } while (b);

  return c;
  }

// Liefert eine Pseudo-Zufallszahl x
// mit 1 <= x <= 2^DEGREE-1 = 32767
unsigned short zufall (void)
  {
  // seed speichert den letzten Wert von Aufruf zu Aufruf
  static unsigned short seed = ROOT;

  return (unsigned short) (seed = pprod (seed, ROOT));
  }

int main(void)  // Testprogramm
```

```
{
int r, i;
// erstmal einige Werte erzeugen
r = time(0) % 512;
for (i=0; i<r; i++) zufall();
for (i=1; i<10; i++)
  {
  r = zufall();
  printf("%5d %4x\n",r,r);
  }
return(0);
}
```

3.4 Nützliche Makrodefinitionen

Wenn Ihnen die mit Schiebeoperationen gebildeten Bitwerte nicht gefallen, können Sie das Manko mit einem einfachen Makro beheben:

```
#define BIT(x) (1 << (x))
```

Die Klammerung im obigen Makro ist wichtig, damit auch Makroaufrufe wie z. B. `BIT(Var + 1)` richtig interpretiert werden. Für den Wert 0x10 kann man nun auch `BIT(4)` schreiben.

Auch das Setzen oder Rücksetzen von einzelnen Bits lässt sich per Makro lesbarer gestalten:

```
/* Bit setzen */
#define SETBIT(VAR,B) ((VAR) |= (1 << (B)))

/* Bit löschen */
#define CLRBIT(VAR,B) ((VAR) &= (unsigned)~(1 << (B)))

/* Bit togglen */
#define TOGBIT(VAR,B) ((VAR) ^= (1 << (B)))

/* Bit abfragen */
#define TSTBIT(VAR, B) (((VAR) & BIT(B))?1:0)
```

4

Programmierprinzipien

4.1 Mapping-Funktion

Vielfach findet man bei Programmen eine `switch - case`-Anweisung, wenn es darum geht, Zahlenwerte irgendwie umzucodieren. Der folgende Programmschnipsel steuert eine Siebensegment-Anzeige an:

```
switch(ziffer)
  {
  case 0: PORTC= 63; break;
  case 1: PORTC=  6; break;
  case 2: PORTC= 91; break;
  case 3: PORTC= 79; break;
  case 4: PORTC=102; break;
  case 5: PORTC=109; break;
  case 6: PORTC=125; break;
  case 7: PORTC=  7; break;
  case 8: PORTC=127; break;
  case 9: PORTC=111; break;
  }
```

Will man auch noch die Hexziffern A bis F, kommen nochmal fünf Zeilen hinzu. Dabei ginge es auch viel einfacher. Man definiert ein Array für die iebensegment-Codes und ordnet die Codes so, dass ihre Reihenfolge dem dezimalen oder hexadezimalen Eingabewert entspricht:

```
//              0  1   2   3    4    5    6  7
byte sseg[] = {63, 6, 91, 79, 102, 109, 125, 7,
//             8    9    A    B   C   D    E    F
             127, 111, 119, 124, 57, 94, 121, 113};
```

Für die Decodierung genügt dann anstelle der `switch - case`-Konstruktion ein einfacher Array-Zugriff:

```
PORTC = sseg[ziffer];
```

Im Variablen-Speicher werden jetzt zwar 15 Bytes zusätzlich belegt, aber der Code ist mit Sicherheit wesentlich kürzer. Das Verfahren läßt sich auf alle Zuordnungs-Probleme anwenden. Mit einem 256 Byte großen Array könnte man beispielsweise ASCII-Zeichen beliebig umcodieren.

4.2 Magic Numbers

Wenn Sie im Programm Berechnungen durchführen, Register setzen oder E/A-Ports ansprechen, sollten Sie vermeiden, dort direkt mit Zahlen zu arbeiten. Nutzen Sie immer die Möglichkeiten von `#define` oder Konstanten mit sinnvollen Namen. Schon nach ein paar Wochen können selbst Sie nicht mehr sagen, was die Zahl in Ihrer Formel sollte. Hierzu ein kleines Beispiel:

```
i = (int) log(2.718281828) + 1
```

Wenn man zufällig weiss, dass 2.718281828 etwa gleich der mathematischen Konstante e ist, bekommt man heraus, dass oben `i = 2` steht. Noch schlimmer kommt es hier:

```
for (i = 0; i < 13; i++)
  { putchar(i + 32); putchar(13); putchar(10); }
```

Wenn man lange genug nachdenkt, kommt man darauf, dass die ersten 13 druckbaren ASCII-Zeichen (beginnend beim Leerzeichen) mit anschließendem Zeilenwechsel (CR+LF) ausgegeben werden. Wehe, es kommt jemand auf die Idee, nun 15 Zeichen ausgeben zu wollen und die 13 global durch 15 ersetzt.

Auch wenn Sie Register haben, die mit ihren Bits irgendwelche Hardware steuern, sollten Sie statt der Magic Numbers einfach einen Header schreiben, welcher über `#define` den einzelnen Bits eine Bedeutung gibt, und dann über das binäre ODER eine Maske schaffen die ihre Ansteuerung enthält, hierzu ein Beispiel:

```
CNTR = 0xA6;
CNTR = BIN | CNT_DOWN | RATE_GEN;
```

Beide Zeilen machen auf einem fiktiven Mikrocontroller das gleiche, aber für den Code in der ersten Zeile müsste ein Programmierer erstmal die Dokumentation des Mikrocontroller lesen, um die Zählrichtung zu ändern. In der zweiten Zeile weiß man sofort, dass das CNT_DOWN geändert werden muss. Wenn der Entwickler des Headers schlau war, ist auch ein CNT_UP bereits definiert. Im folgenden Positiv-Beispiel werden einzelne Bits gesetzt, deren Bedeutung sich dem Programmierer aus den Namen der Konstanten sofort erschließt:

```
TCCR0 |= ( 1<<CS02 )|( 1<<CS00 );  // counter0, Prescaler auf 1024
TIMSK |= ( 1<<TOIE0 );             // enable counter0 overflow interrupt
TCNT0 = 0x00;                      // Counter0 auf Null setzen
```

In einigen Fällen sind numerische Konstante akzeptiert und zwar dort, wo sie eben nicht „magisch“ sind. Unter anderem ist dies in folgenden Situationen der Fall:

- Verwendung von 0 und 1 als Initial- oder Incrementalwerte in Schleifen, wie beispielsweise in: `for (i = 0; i < max; i = i + 1)`.
- Verwendung von 2, um zu prüfen, ob eine Zahl gerade oder ungerade ist (z. B. `even = (x % 2 == 0)`) oder bei Zweierpotenzen.
- Verwendung von einfachen arithmetischen Konstanten wie beispielsweise in der Formel $U = 2 \cdot \pi \cdot R$ für den Kreisumfang.

Jedoch ist es sinnvoll, zwischen dem Zahlenwert 0, dem Buchstaben mit dem Code 0 (`'\0'`) und dem Nullpointer (`null`) zu unterscheiden, auch wenn der C-Compiler das nicht unbedingt braucht.

4.3 Enumeration

Erstaunlicherweise lieben immer noch viele Programmierer das Definieren zahlreicher Konstanten mittels `#define` und schreiben beispielsweise

```
#define Mon 1
#define Tue 2
#define Wed 3
#define Thu 4
#define Fri 5
#define Sat 6
#define Sun 7
```

um „sprechende“ Konstantennamen verwenden zu können. Dabei liefert uns die Sprache C eine oft sehr viel bessere Möglichkeit (was nichts daran ändert, dass oft auch solche Konstanten-Ungetüme notwending sind).

Das `enum`-Schlüsselwort ist ein oft sehr stiefmütterlich behandeltes Konstrukt der Sprache C. Es wird zum Deklarieren eines Aufzählungstyps (Enumeration) verwendet. Dies ist ein eigener Typ, der aus einer Gruppe benannter Konstanten, der so genannten Enumeratorliste, besteht. Jeder Enumerationstyp besitzt einen zugrunde liegenden Typ, bei dem es sich um jeden ganzzahligen Typ außer char handeln kann. Meist handelt es sich um `int`, aber manche Mikrocontroller-Compiler verwenden auch

nur 8-Bit-Typen. Der erste *Enumerator* hat per Default den Wert 0, und der Wert jedes nachfolgenden Enumerators wird jeweils um 1 erhöht. Ein Enumeratorname darf keine Leerzeichen enthalten. Beispiel:

```
enum Tage {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
   ...
enum Tage Day = Mon;
```

In dieser Enumeration entspricht `Mon` dem Wert 0, `Tue` dem Wert 1, `Wed` dem Wert 2 usw. Enumeratoren können über *Initialisierer* verfügen, die die Standardwerte überschreiben, z. B. :

```
enum Tage {Mon=1, Tue, Wed, Thu, Fri, Sat, Sun};
```

In dieser Enumeration wird erzwungen, dass die Abfolge von Elementen mit 1 und nicht mit 0 beginnt. Es kann auch jedem Element ein bestimmter Wert zugewiesen werden.

Nachteilig ist, dass einer Variablen vom Aufzählungstyp jeder Wert im Bereich des zugrunde liegenden Typs zugewiesen werden. Die Werte sind nicht auf die benannten Konstanten eingeschränkt. Der zugrunde liegende Typ legt zwar fest, wie viel Speicher für jeden Enumerator reserviert wird. Es ist jedoch eine explizite Typumwandlung (Typecast) erforderlich, um einen enum-Typ in einen ganzzahligen Typ zu konvertieren.

Wird ein Aufzählungstyp öfter im Programm verwendet, ist es sinnvoll, mittels `typedef` einen regulären Datentyp zu erzeugen. Es braucht dann nicht immer das Schlüsselwort `enum` jedesmal angegeben werden. Auch das Tag-Feld vor der geschweiften Klammer kann dann weggelassen werden.

```
typedef enum {Mon, Tue, Wed, Thu, Fri, Sat, Sun} tage_t;
typedef enum {FALSE, TRUE} bool_t;
typedef enum {BELL = '\a', BACKSPACE = '\b', HTAB = '\t',
            RETURN = '\r', NEWLINE = '\n', VTAB = '\v' } special_t;

   ...
tage_t Day = Mon;
bool_t Error = FALSE;
special_t c = BELL;
```

Ein Vorteil von `enum` gegenüber `#define` besteht darin, dass Enumeratoren einen Gültigkeitbereich besitzen. Werte, die daraus abgeleitet werden, sind wie alle anderen Variablen auch, nur in einem bestimmten Bereich gültig sind während `#define`-Festlegungen im ganzen Programm gelten. Da in C keine Namensräume existieren, werden Enumerations leider grundsätzlich im globalen Namensraum angelegt.

Der Vorteil eines Aufzählungstyps zeigt sich am folgenden Beispiel einer Ampel. Ohne Enumeration kann man den aktuellen Status einer Ampel wie hier speichern:

```
int Ampel(int light)
  {
  if     (light == 0) printf("Ampel ist aus\n");
  else if(light == 1) printf("Ampel ist grün\n");
  else if(light == 2) printf("Ampel ist gelb\n");
  else if(light == 3) printf("Ampel ist rot\n");
  else if(light == 4) printf("Ampel ist rot/gelb\n");
  else                  printf("Ampel ist kaputt\n");
  }
```

Oder auch mit einer `switch -- case`-Konstruktion:

```
int Ampel(int light)
  {
  switch(light)
    {
    case 0: printf("Ampel ist aus\n"); break;
    case 1: printf("Ampel ist grün\n"); break;
    case 2: printf("Ampel ist gelb\n"); break;
    case 3: printf("Ampel ist rot\n"); break;
    case 4: printf("Ampel ist rot/gelb\n"); break;
    default: printf("Ampel ist kaputt\n");
    }
  }
```

Bei Programmieren muss man also immer im Kopf behalten, welche Zahl welche Bedeutung hat, womit wir wieder einmal bei den oben erwähnten „Magic Numbers“ angelangt wären. Mit einem Aufzählungstyp werden die oben verwendeten Zahlenwerte vermieden. Die fünf Zustände der Ampel kann man aufzählen und die Funktion entsprechend anpassen:

```
enum AmpelStatus {Off, Gruen, Gelb, Rot, RotGelb};

int Ampel(enum AmpelStatus light)
  {
  switch(light)
    {
    case Off:     printf("Ampel ist aus\n"); break;
    case Gruen:   printf("Ampel ist grün\n"); break;
    case Gelb:    printf("Ampel ist gelb\n"); break;
    case Rot:     printf("Ampel ist rot\n"); break;
    case RotGelb: printf("Ampel ist rot/gelb\n"); break;
    default: printf("Ampel ist kaputt\n");
    }
  }
```

Statt kryptischer Zahlen sieht man nun, was passiert. Und die Funktion erwartet als Eingabe einen Wert vom Typ (enum AmpelStatus), statt einer nichtssagenden Integerzahl.

4.4 Variante Strukturen (union)

Während bei einer normalen Struktur alle Komponenten im Speicher hintereinander liegen, gestattet der Datentyp `union` mehrere Komponenten übereinander zu legen, sodass ein- und derselbe Platz für verschiedenartige Daten genutzt werden kann. Genauer gesagt: alle Komponenten beginnen an der selben Stelle im Speicher. Sind die Komponenten unterschiedlich groß, enden sie entsprechend an verschiedenen Stellen. Verwendet werden solche varianten Strukturen, wenn ein- und dieselben Daten über unterschiedliche Datentypen angesprochen werden sollen.

Bei der Intel-Prozessorarchitektur können Unions für die Arbeit mit den Registern eingesetzt werden, die entweder byte- oder wortweise angesprochen werden. Zum Beispiel:

```
struct WORDREGS /* wortweise adressierte Register */
  {
  unsigned int ax, bx, cx, dx, si, di, cflag, flags;
  };

struct BYTEREGS /* byteweise adressierte Halbregister */
  {
  unsigned char al, ah, bl, bh, cl, ch, dl, dh;
  };

union REGS /* die variable Verbindung dieser beiden */
  {
  struct WORDREGS x;
  struct BYTEREGS h;
  };
```

Wird nun eine Variable mittels `union REGS reg` deklariert, kann mit `reg.x.ax = 0xFF00` dafür gesorgt werden, dass `reg.h.ah` den Wert `0xFF` und `reg.h.al` den Wert `0x00` erhält.

Ein Spezialfall einer Struktur sind die sogenannten Bit-Felder (bit fields). Hier wird in einer Strukturvereinbarung durch einen Doppelpunkt nach dem Bezeichner vorgeschrieben, wieviele Bits diese Komponente umfassen soll. Bedenken Sie aber, dass fast alle Aspekte von Bit-Feldern implementierungsabhängig sind. Deshalb gibt es auch keine Arrays von Bit-Feldern und Bit-Felder haben keine Adressen, auf die der Adressoperator angewendet werden könnte.

```
#include <stdio.h>
#include <stdlib.h>

/* Definition von 8 einzelnen Bits  (Laenge 1)*/
typedef struct
  {
  unsigned int a7:1;
  unsigned int a6:1;
  unsigned int a5:1;
  unsigned int a4:1;
  unsigned int a3:1;
  unsigned int a2:1;
  unsigned int a1:1;
  unsigned int a0:1;
  } BIT;
```

```
/* Hier werden ein Byte (unsigned char) und die
   Bits uebereinander gelegt. Die Komponenten
   'byte' und die Struktur BIT belegen denselben
   8-Bit-Wert */
typedef union
  {
  unsigned char byte;
  BIT bit;
  } BITS;

/* Eine Ausgaberoutine fuer die einzelen Bits */
void aus (BITS wert)
  {
  printf("%2d %2d %2d %2d %2d %2d %2d %2d\n",
         wert.bit.a7,wert.bit.a6,wert.bit.a5,wert.bit.a4,
         wert.bit.a3,wert.bit.a2,wert.bit.a1,wert.bit.a0);
  }

/* Hauptprogramm zeigt den Effekt */
int main()
  {
  BITS wert;
  wert.byte=0x55;
  aus(wert);

/* Ausgabe:  1  0  1  0  1  0  1  0 */

  wert.byte=0xAA;
  aus(wert);

/* Ausgabe:  0  1  0  1  0  1  0  1 */

  wert.bit.a7 = 1;
  aus(wert);

/* Ausgabe:  1  1  0  1  0  1  0  1 */

  return 0;
  }
```

Unions erlauben effiziente Speicherung einzelner Bits, z. B. auch Flags oder I/O-Konfigurationsbits. Es gilt jedoch:

- Komponenten können nur per Namen angesprochen werden, nicht über die Adresse
- es geht nur innerhalb von struct und union
- es gibt keine Arrays
- die Bitorder ist nicht portabel (denken Sie an Big- und Little-Endian
- Eine union ist immer so groß wie ihr größtes Element.

Noch ein Beispiel, bei dem eine Gleitpunktzahl im IEEE-Format in ihre Komponenten zerlegt wird. Die `union` würde sich beispielsweise eignen, um ohne viel Rechenaufwand den Exponenten der Zahl zu ermitteln – einfach 127 von der Charakteristik subtrahieren:

```
typedef struct
  {
  unsigned int sign:1;  // ein Bit Vorzeichen
  unsigned int char:8;  // acht Bit Charakteristik
  unsigned int frac:23; // 23 Bit Mantisse
  } ieee_float;         // insgesamt 32 Bit

typedef union
  {
  ieee_float a;
  float b;
  } my_float;
```

Ein weiters Beispiel für die PC-Architektur und Linux. Es dient zum Verarbeiten des Statusbytes, das unter der Basisadresse+1 vom Duckerport gelesen werden kann. Betrieb nur als root-User, Compilieren mit -O oder -O2.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/io.h>
#include <unistd.h>

#define BASEPORT 0x378

union statusport
  {
  unsigned char byte;
  struct status
     {
     unsigned int unused:3; // Bit 0-2 nicht verwendet
     unsigned int fehler:1; // 0 = Error
     unsigned int online:1; // 1 = Drucker online
     unsigned int papier:1; // 1 = kein Papier
     unsigned int ack:1;    // 1 = Ack
     unsigned int busy:1;   // 1 = Busy
     } bit;
  } statusport;

int main(void)
   {
   /* Port freischalten */
   if (ioperm(BASEPORT, 3, 1))
      { perror("Error: cannot access ioport"); exit(255); }

   /* Get status port */
   statusport.byte = inb(BASEPORT + 1);

   if(statusport.bit.busy && statusport.bit.online)
     {
     printf("Drucker ist bereit!\n");
     ioperm(BASEPORT, 3, 0);
     return 0;
     }
   else if(!statusport.bit.online)
      printf("Drucker nicht online!\n");
   else if(statusport.bit.papier)
      printf("Kein Papier vorhanden!\n");
   else
      printf("Drucker ist nicht bereit!\n");
   ioperm(BASEPORT, 3, 0);
   return 1;
   }
```

5

Typische Algorithmen

5.1 Statemaschine

Ein Zustandsautomat nimmt sich eine abstrakte Maschine zum Vorbild, die über *interne Zustände* verfügt. Die Maschine arbeitet, indem sie von einem Zustand in einen anderen Zustand wechselt und dabei weitere Aktionen ausführt. Der Folgezustand wird dabei durch den aktuellen Zustand und gegebenenfalls externe Ereignisse, z. B. einem Tastendruck, festgelegt.

Hier wollen wir uns mit der Implementierung von Zustandsautomaten beschäftigen. Grundsätzlich können nur vollständige, deterministische Automaten direkt implementiert werden (nicht deterministische Automaten können nur mit Hilfe von Backtracking realisiert werden). Bei unvollständigen Automaten kann man vielfach mit einem Fehlerzustand reagieren, wenn ein nicht gültiges Eingabesymbol vorkommt. Im Prinzip gibt es zwei Möglichkeiten, einen Automaten zu implemeniteren:

- Implementierung mit einer `switch -- case`-Anweisung.
- Implementierung mit einer Zustandstabelle.

5.1.1 Implementierung mit einer switch – case-Anweisung

Hier wird in einer Schleife ein Eingabesymbol nach dem anderen abgearbeitet. Der aktuelle Zustand wird in einer Variablen (z. B. state) gespeichert. Die Schleife muss verlassen werden, wenn ein Endzustand erreicht oder ein ungültiges Eingabesymbol gelesen wurde.

Jedem Zustand ist innerhalb der `switch`-Anweisung ein `case`-Label zugeordnet. Für jeden Zustand wird dann für das gelesene Symbol entschieden, welcher Folgezustand angenommen werden muss. Der Variablen `state` wird der neue Zustand zugewiesen. Soll eine Funktion beim Zustandsübergang ausgeführt werden, so braucht diese nur eingefügt werden. Man kann dabei folgendermaßen vorgehen:

- Es gibt eine globale Zustands-Variable. Die einzelnen Zustände werden über einen Aufzählungstyp definiert.
- Die Statemachine wird als Funktion implementiert, die für jeden einzelnen Takt (Schleifendurchlauf der Hauptschleife des Programms) aufgerufen wird.
- Jeder Zustand wird innerhalb der Funktion durch einen C-`case` innerhalb einer `switch`-Anweisung dargestellt.
- Der Folgezustand wird durch Ereignisse und den aktuellen Zustand festgelegt. Die Zustandsvariable wird als `static`-Variable definiert.
- Jegliche Form von Warteschleifen innerhalb der Statemachine ist verboten. Wenn die Statemachine auf ein Ereignis warten muss, ist dafür ein eigener Zustand vorzusehen, der auf das Eintreten des Ereignisses prüft und nur dann den nächsten Zustand auswählt, wenn das Ereignis eingetreten ist.

Als Beispiel soll hier ein Automat dienen, der Binärzahlen akzeptiert, die durch 3 teilbar sind. Das Zustandsdiagramm ist in Bild 5.1 dargestellt. Der Automat soll also Zahlen wie beispielsweise 11, 110, 1001, 1100, 1111, 10010 usw. akzeptieren:

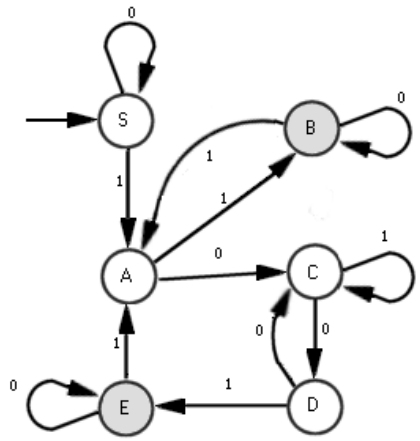

Bild 5.1: Zustandsfolgen des Beispielautomaten (B und E sind Endzustände)

```
#include <stdio.h>
#include <stdlib.h>

/*Definintion des Automaten */
#define N_ZUST 6  /* Anzahl Zustaende */
typedef enum States
  {
  NIL  = -1,
  S    =  0,
  A    =  1,
  B    =  2,
  C    =  3,
  D    =  4,
  E    =  5
  } states_t;

#define IST_TEILBAR(s) ((s == B) || (s == E))

/* Statemachine bearbeitet Input-String */

states_t statemachine(char *input)
  {
  states_t state = S;
  int in;

  while(*input != '\0')
    {
    in = *input - '0';
    if ((in == 0) || (in == 1))
      {
      switch( state )
        {
        case S: /* führende Nullen erlauben */
          if (in == 0) state = S;
          else         state = A;
          break;

        case A:
          if (in == 0) state = C;
          else         state = B;
          break;

        case B: /* Endzustand */
          if (in == 0) state = B;
          else         state = A;
          break;

        case C:
```

```
            if (in == 0) state = D;
            else         state = C;
            break;

          case D:
            if (in == 0) state = C;
            else         state = E;
            break;

          case E: /* Endzustand */
            if (in == 0) state = E;
            else         state = A;
            break;
          }
        }
      else /* Abbruch */
        return(NIL);
      input++;
      }
    return(state);
    }

  int main(int argc, char *argv[])
    {
    states_t EndState;

    argv++;
    while(argc > 1)
      {
      printf("Eingabewert: %s ", *argv);
      EndState = statemachine(*argv);
      if (IST_TEILBAR(EndState))
         printf( "ist durch 3 teilbar\n" );
      else
         printf("ist nicht teilbar\n");
      argv++; argc--;
      }
    return 0;
    }
```

Die Statemaschine hat noch eine interne Schleife zur Bearbeitung der kompletten Eingabestrings. Überlegen Sie, wie man die Funktion umschreiben müsste, um die Schleife ins Hauptprogramm zu verlagern.

5.1.2 Implementierung mit einer Zustandstabelle

Auch hier wird in einer Schleife ein Eingabesymbol nach dem anderen abgearbeitet. Jedoch wird die Zustandstabelle des Automaten direkt als zweidimensionales Array implementiert (Bild 5.2). Eine Dimension entspricht den Zuständen, die auf den Index abgebildet werden müssen. Die zweite Dimension entspricht den Eingabesymbolen, wobei auch hier die Symbole auf den Index abgebildet werden müssen (dies ist bei der vorigen Variante nicht nötig).

Bild 5.2: Schema der Zustandstabelle

Ist es notwendig, dass Funktionen bei einem Zustandsübergang ausgeführt werden, so könnte man z. B. ein Feld von Strukturen verwenden, wobei in den Strukturen nicht nur der Folgezustand gespeichert wird, sondern auch ein Zeiger auf die auszuführende Funktion.

Es handelt sich um den selben Automaten wie oben. Die Zustandstabelle sieht folgendermaßen aus:

State	Next		Ende?
	0	1	
S	S	A	
A	C	B	
B	B	A	ja
C	D	C	
D	C	E	
E	E	A	ja

Diese Tabelle wird als zweidimensionales Array im Programm definiert. Die Bearbeitung des Automaten reduziert sich dann auf eine einzige Zeile:

```
state = TransitionTable[state][in];
```

Als weiterer Vorteil kommt hinzu, dass alle Eigenschaften der Statemaschine durch die Zustandstabelle festgelegt sind. Bei einer Änderung muss nicht der Programmablauf (fehlerträchtig) geändert werden, sondern nur die Tabelle. Bei anderen Eingangs-„Events“ bietet es sich auch an, für diese eine weitere Enumeration vorzusehen. Damit ergibt sich folgendes Programm:

```
#include <stdio.h>
#include <stdlib.h>

/*Definintion des Automaten */
#define N_ZUST 6  /* Anzahl Zustaende */
#define N_INP 2   /* Anzahl Eingangswerte */

typedef enum States
  {
  NIL = -1,   S   =  0,
  A   =  1,   B   =  2,
  C   =  3,   D   =  4,
  E   =  5
  } states_t;

states_t TransitionTable[N_ZUST][N_INP] =
  {
  /*  State    0  1  */
     /* S */  {S, A},    /* führende Nullen erlauben */
     /* A */  {C, B},
     /* B */  {B, A},    /* Endzustand */
     /* C */  {D, C},
     /* D */  {C, E},
     /* E */  {E, A}     /* Endzustand */
  };

#define IST_TEILBAR(s) ((s == B) || (s == E))

/* Statemachine bearbeitet Input-String */

states_t statemachine(char *input)
  {
  states_t state = S;
  int in;

  while(*input != '\0')
    {
    in = *input - '0';
    if ((in == 0) || (in == 1))
      state = TransitionTable[state][in];
    else
       return(NIL);
    input++;
    }
  return(state);
  }

int main(int argc, char *argv[])
  {
  states_t EndState;

  argv++;
  while(argc > 1)
    {
    printf("Eingabewert: %s ", *argv);
    EndState = statemachine(*argv);
```

```
    if (IST_TEILBAR(EndState))
       printf( "ist durch 3 teilbar\n" );
    else
       printf("ist nicht teilbar\n");
    argv++; argc--;
    }
  return 0;
  }
```

Was nun noch fehlt, sind die Aktionen bei den einzelnen Zuständen. Bisher wurde nur der Zustandswechsel (abhängig vom aktuellen Zustand und dem „Event“) programmiert, aber noch keine zusätzlichen Aktionen ausgelöst. Auch dies ist ein Vorteil der Statemaschine als Softwarekonstrukt: man kann die beiden Aufgabenbereiche gut voneinander trennen und auch separat ändern.

Das obige Programm wird um zwei Teile erweitert. Zuerst werden die Aktionen für jeden Zustand als Funktionen definiert, wobei diese Funktionen auch in einer separaten Datei gehalten werden könnten und in der Statemaschine nur eine Header-Datei per `#include` eingelesen wird.

Als zweites wird analog zur Zustandstabelle ein Array von Funktionspointern auf die Aktionen im Programm eingetragen:

```
void (* ActionTable [N_ZUST])(void) =
  {
  action_S, action_A, action_B,
  action_C, action_D, action_E
  };
```

Achten Sie bei der Typdefinition der `States` und den beiden Tabellen darauf, dass die gleiche Reihenfolge eingehalten wird – sonst passen sie nicht zueinander. Nun werden in der Statemaschine noch zwei Aufrufe von Aktionen untergebracht. Am Anfang wird die Startaktion ausgeführt und in der Schleife nach jedem Zustandswechsel die entsprechende Aktion. Der Übersichtlichkeit halber sind im folgenden Beispiel die Aktions-Funktionen ohne Parameter definiert worden. Häufig werden solche Funktionen mit dem aktuellen Event als Parameter versorgt.

```
#include <stdio.h>
#include <stdlib.h>

/*Definintion des Automaten */
#define N_ZUST 6  /* Anzahl Zustaende */
#define N_INP 2   /* Anzahl EIngangswerte */

typedef enum States
  {
  NIL = -1,   S   =  0,
  A   =  1,   B   =  2,
  C   =  3,   D   =  4,
  E   =  5
  } states_t;

states_t TransitionTable[N_ZUST][N_INP] =
  {
  /*  State   0  1  */
     /* S */  {S, A},    /* führende Nullen erlauben */
     /* A */  {C, B},
     /* B */  {B, A},    /* Endzustand */
     /* C */  {D, C},
     /* D */  {C, E},
     /* E */  {E, A}     /* Endzustand */
  };

/* einige total sinnlose Aktionen */
void action_S(void) { printf("Action S\n"); }
void action_A(void) { printf("Action A\n"); }
void action_B(void) { printf("Action B\n"); }
void action_C(void) { printf("Action C\n"); }
void action_D(void) { printf("Action D\n"); }
void action_E(void) { printf("Action E\n"); }

/* Array von Funktionspointern */
void (* ActionTable [N_ZUST])(void) =
  {
  action_S, action_A, action_B,
  action_C, action_D, action_E
```

```
  };

#define IST_TEILBAR(s) ((s == B) || (s == E))

/* Statemachine bearbeitet Input-String */

states_t statemachine(char *input)
  {
  states_t state = S;
  int in;

  /* Aktion fuer Start-Zustand ausfuehren */
  ActionTable[state] ();

  while(*input != '\0')
    {
    in = *input - '0';
    if ((in == 0) || (in == 1))
      {
      /* Zustandswechsel */
      state = TransitionTable[state][in];
      /* Aktion fuer neuen Zustand ausloesen */
      ActionTable[state] ();
      }
    else
       return(NIL);
    input++;
    }
  return(state);
  }

int main(int argc, char *argv[])
  {
  states_t EndState;

  argv++;
  while(argc > 1)
    {
    printf("Eingabewert: %s\n", *argv);
    EndState = statemachine(*argv);
    if (IST_TEILBAR(EndState))
       printf( "ist durch 3 teilbar\n" );
    else
       printf("ist nicht teilbar\n");
    argv++; argc--;
    }
  return 0;
  }
```

Der Beispiel-Aufruf `./state 001111 11 1010` erzeugt die folgende Ausgabe:

```
Eingabewert: 001111
Action S
Action S
Action S
Action A
Action B
Action A
Action B
ist durch 3 teilbar
Eingabewert: 11
Action S
Action A
Action B
ist durch 3 teilbar
Eingabewert: 1010
Action S
Action A
Action C
Action C
Action D
ist nicht teilbar
```

5.1.3 Unterschiedliche Laufzeiten der Aktionen

Häufig kommt es vor, dass einzelne Aktionen/Tasks sehr unterschiedliche Laufzeiten haben. Beispielsweise sollte das Multiplexen einer Anzeige in der Regel immer gleich getaktet sein. Wenn nun ein Task zum Auslesen eines Sensors hinzu kommt, der sehr viel länger dauert, würde die Multiplex eventuell bei jeder Messung unregelmäßig flackern. Das will man natürlich nicht. Was also tun? Die Messdauer lässt sich ja nicht verkürzen und das Multiplexen kann eben sowenig beliebig gedehnt werden.

Der Trick besteht darin, die kürzeste benötigte Taktperiode als Maßstab für die Taktung des Zustandsautomaten verwenden. Dann werden aber die „langsamen“ Aktionen zu häufig aufgerufen. Dieses Manko behebt man, indem diese Tasks zusätzlich intern getaktet werden. Dies kann mit einer einfachen Zählvariablen erreicht werden. Bei jedem Durchlauf wird die Zählvariable inkrementiert und nur bei Erreichen des Schwellenwertes wirklich etwas getan und die Zählvariable zurückgesetzt. Das folgende Schema zeigt, was gemeint ist:

```
void Messung()
  {
static int count = 0;    /* Zaehlvariable - static, damit der Wert
                            von Aufruf zu Aufruf erhalten bleibt */
  count++;                 /* Inkrementieren */
if (count >= SCHWELLE)   /* Messung nur alle SCHWELLE Durchlaeufe starten */
   {
 Speichere_Messwert();
 Starte_Messung();
 count = 0;
 }
}
```

Im Beispiel oben wird nur bei jedem n-ten Durchlauf (`SCHWELLE` definiert den Wert von n) der Wert der aktuellen Messung abgespeichert und anschließend die nächste Messung gestartet. Deren Ergebnis wird dann wieder nach n Durchläufen übernommen und so weiter.

Damit werden länger dauernde Aktionen in den Takt des Zustandsautomaten eingebunden, obwohl dieser mit einem schnellen Takt läuft. Das folgende Beispiel realisiert auf recht komplizierte Weise eine Art Binärzähler als Zustandsautomat und zeigt, wie sich sogar recht einfach mehrere Tasks unterschiedlich takten lassen. Der schnellste Task ist hier `task0`, nach dem sich alle anderen richten. `task1` läuft mit halber Geschwindigkeit, `task2` und `task3` mit einem Viertel bzw. einen Achtel der Grundgeschwindigkeit. Das Beispiel steuert LEDs am GPIO-Port eines Raspberry Pi Computers an.

```
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <signal.h>
#include "GPIO.h"

/* Signalhandler fuer STRG-C */
void  finish(int sig)
  {
  int pin;

  printf("\nSignal empfangen. Programm wird beendet.\n");
  /* Switch off */
  for (pin = 22; pin <= 25; pin++)
    GPIO_Write(pin, LOW);
  /* Disable GPIO pins   */
  for (pin = 22; pin <= 25; pin++)
    GPIO_Unexport(pin);
  exit(0);
  }

void task0(int *LED)
  {
  GPIO_Write(LED[0], ! GPIO_Read(LED[0]));
  }

void task1(int *LED)
```

```
  {
  static int count = 0;

  count = (count + 1)%2;
  if (count == 0)
    GPIO_Write(LED[1], ! GPIO_Read(LED[1]));
  }

void task2(int *LED)
  {
  static int count = 0;

  count = (count + 1)%4;
  if (count == 0)
    GPIO_Write(LED[2], ! GPIO_Read(LED[2]));
  }

void task3(int *LED)
  {
  static int count = 0;

  count = (count + 1)%8;
  if (count == 0)
    GPIO_Write(LED[3], ! GPIO_Read(LED[3]));
  }

int main(void)
  {
  int pin;      /* Laufvariable */
  int state;    /* aktueller Zustand */

  /* LED-Pins */
  int LED[4] = {22, 23, 24, 25};

  struct sigaction sig_struct;

  /* Signalhandler fuer STRG-C einrichten */
  sig_struct.sa_handler = finish;
  sigemptyset(&sig_struct.sa_mask);
  sig_struct.sa_flags = 0;
  sigaction(SIGINT,&sig_struct,NULL);

  /* Enable GPIO Output Pins     */
  for (pin = LED[0]; pin <= LED[3]; pin++)
    { if (GPIO_Export(pin) < 0)
        return(1);
    }
  /* Set GPIO Output directions */
  for (pin = LED[0]; pin <= LED[3]; pin++)
    { if (GPIO_Direction(pin, OUT) < 0)
        return(2);
    }

  state = 0;
  while(1)
    {
    switch (state)
      {
      case 0: task0(LED); state = 1; break;
      case 1: task1(LED); state = 2; break;
      case 2: task2(LED); state = 3; break;
      case 3: task3(LED); state = 0; break;
      }
    usleep(100 * 1000);
    }

  return(0);
  }
```

5.2 FIFO-Speicher

„FIFO“ ist die Abkürzung für „First In First Out“. Es handelt sich dabei um einen Pufferspeicher nach dem Warteschlangen-Prinzip. Ein neu hinzukommendes Element wird hinten an die Warteschlange angehängt, wogegen beim Lesen das erste Element der Warteschlange entnommen wird. Die Software-Realisierung eines FIFO-Speichers erfolgt meist als Ringpuffer (auch „zirkulärer Puffer“ genannt). Sie kann entweder als einfach verkettete Liste mit Pointern (variable Puffergröße) oder mit einem Array (feste Puffergröße) erfolgen. Wesentlicher Vorteil des Arrays gegenüber der verketteten Liste ist die schnellere Ausführungszeit und der geringere Speicherbedarf. Deshalb erfolgt bei Controllern in der Regel eine Programmierung mit Arrays. Man stellt sich einfach vor, dass der Anfang des Arrays dicht auf dessen Ende folgt (Bild 5.3).

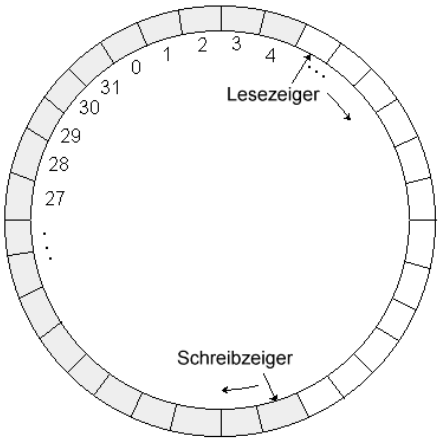

Bild 5.3: Schema des Ringpuffers

Die Inhalte des Puffers werden also in einem Array gespeichert und der Zugriff erfolgt über eine Integer-Variable als Index. Erreicht der Index die Array-Obergrenze wird er wieder auf Null gesetzt:

```
writeptr++;
if (writeptr >= BUFFERSIZE)
  writeptr = 0;
```

Der Ringpuffer braucht natürlich nicht nur einen Schreib-Zeiger (Schreib-Index), sondern auch einen Lese-Zeiger (Lese-Index). Haben Lese- und Schreib-Zeiger den gleichen Wert, ist der Puffer leer. Ist der Lese-Zeiger direkter Nachfolger des Schreib-Zeigers, ist der Puffer voll.

```
Puffer leer: (readptr == writeptr);
Puffer voll: ((writeptr + 1 == read) || ((readptr == 0) && (writeptr + 1 == BUFFERSIZE));
```

Erfolgt Schreiben/Lesen auf dem Puffer per Interrupt, muss sichergestellt werden, dass die Lese- oder Schreiboperation unterbrochen wird. Gegebenenfalls muss man atomare Abschnitte bilden, die durch keinen Interrupt unterbrochen werden. Die Radialkur deaktiviert alle Interrupts, besser ist die Deaktivierung eines einzelnen Interrupts. Normalerweise werden versäumte Interrupts nach deren erneuter Aktivierung nachgeholt.

Wie man oben schon sehen konnte ist das Inkrementieren eines der beiden Zeiger etwas unübersichtlich, weshalb gleich hier eine Funktion `succ()` für „Successor“ (Nachfolger) eingeführt wird, die den Nachfolgewert innerhalb des Ringpuffers liefert:

```
int succ(int ptr)
  {
  ptr++;
  if (ptr >= BUFFERSIZE) ptr = 0;
  return(ptr);
  }
```

Die Tatsache, dass der Pointer/Index nicht direkt inkrementiert wird, ermöglicht die Vereinfachung der „voll“-Abfrage:

```
Puffer leer: (readptr == writeptr);
Puffer voll: (succ(writeptr) == readptr);
```

Schon sind wir gerüstet für die Realisierung des Ganzen. Wenn alle Daten zum Puffer (Array und Zeiger) in einer Struktur gespeichert werden, bekommen wir sogar einen zarten Hauch von Objektorientierung:

```
#include <stdio.h>
#include <stdlib.h>

// Puffergroesse
#define BUFFERSIZE 42

// Fehler-Rueckgabewert
#define FAIL (-1)

//O.K.-Rueckgabewert
#define SUCCESS (0)

struct Puffer                // Puffer-Struktur
  {
  uint8_t data[BUFFERSIZE];  // FIFO-Puffer
  uint8_t readptr;           // Lese-Zeiger
  uint8_t writeptr;          // Schreibzeiger (zeigt auf's naechste freie Element)
  };
struct Puffer buffer = {{0}, 0, 0};

uint8_t succ(uint8_t ptr)
  {
  ptr++;
  if (ptr >= BUFFERSIZE) ptr = 0;
  return(ptr);
  }

// ein Byte in den Puffer schreiben
uint8_t Put(uint8_t byte)
  {
  // printf("DEBUG Put: %d -> %d\n",buffer.writeptr, byte);
  if (succ(buffer.writeptr) == buffer.readptr)  // Puffer voll
    return FAIL;
  buffer.data[buffer.writeptr] = byte;          // Daten ablegen
  buffer.writeptr = succ(buffer.writeptr);      // Schreibzeiger inkrementieren
  return SUCCESS;
  }

// ein Byte aus dem Puffer lesen
uint8_t Get(uint8_t *pByte)
  {
  // printf("DEBUG Get: %d -> %d\n",buffer.readptr, buffer.data[buffer.readptr]);
  if (buffer.readptr == buffer.writeptr)        // Puffer leer
    return FAIL;
  *pByte = buffer.data[buffer.readptr];         // Daten an Parameter uebergeben
  buffer.readptr = succ(buffer.readptr);        // Lesezeiger inkrementieren
  return SUCCESS;
  }
```

Wählt man für die Puffergröße eine Zweierpotenz, vereinfacht sich das Programm noch etwas. Die Definition der Obergrenze wird nicht durch einen Vergleich realisiert, sondern durch eine UND-Verknüpfung. die Funktion `succ()` degeneriert dann zu:

```
uint8_t succ(uint8_t ptr)
  {
  return ((ptr + 1) & MASK);
  }
```

`MASK` ist dabei genau `BUFFERSIZE - 1`.

Im Normalbetrieb sollte der Puffer nie voll ausgenutzt werden, denn wenn der Puffer bereits überläuft ist es meist schon zu spät. In diesem Fall kann ein Frühwarnsystem mit Meldeschwelle hilfreich sein, zum Beispiel:

```
if (abs(readptr - writeptr) < SCHWELLE)
   alert();
```

6

Ein- und Ausgabe

6.1 Software-Entprellung

Das Entprellen von Tasten ist vielfach und immer wieder Diskussionsthema. Ein anscheinend einfaches Problem erweist sich plötzlich als gar nicht so einfach, denn es spielt die Zeit eine Rolle. In der Regel dauert es 10 bis 20 ms, bis eine Taste einen stabilen Zustand angenommen hat. Nun ist es mitunter höchst ungünstig, innerhalb der üblichen `while(1)`-Schleife eine doch relativ lange Pause einzulegen, weshalb man oft als erste Idee auf eine Interruptsteuerung kommt. Das löst aber nicht das Problem an sich, denn der erste Kontakt der Taste löst den Interrupt aus und eine längeres Warten (bis es nicht mehr prellt) innerhalb der Interrupt-Serviceroutine ist noch ungünstiger.

Eine einfache Funktion zum Tasteneinlesen mit Warteschleife zeigt folgendes Listing. Wie Sie sehen, wird das gewünschte Eingaberegister als `volatile` definiert, damit der Port überhaupt gelesen wird. Im Programm wird die Funktion dann beispielsweise mittels `i = keypressed (&PINB, PB1)` aufgerufen.

```
uint8_t keypressed (volatile uint8_t *inreg, uint8_t inbit)
  {
  static uint8_t zustand = 0;

  if (zustand == (*inreg & (1<<inbit)))
     return 0; /* keine Änderung */

  /* Wenn doch, warten bis etwaiges Prellen vorbei ist: */
  _delay_ms(20);

  /* Zustand für naechsten Aufruf merken */
  zustand = *inreg & (1<<inbit);

  /* und den entprellten Tastendruck zurueckgeben */
  return *inreg & (1<<inbit);
  }
```

Die Zustandsvariable erlaubt es, zwei aufeinander folgende Tastendrücke zu erkennen, da die Routine sich den vorhergehenden Zustand des Ports merkt.

Ohne Zustandsvariable kommt das folgende Beispiel aus, das aber andere Nachteile besitzt: die Funktion detektiert erst das Loslassen der Taste, was minder ergonomisch ist, und außerdem braucht sie die doppelte Verzögerung. Die Gefahr, dass Tastenbetätigungen „untergehen“ ist also noch höher.

```
uint8_t keypressed(volatile uint8_t *inreg, uint8_t inbit)
  {
  if ( !(*inreg & (1 << inbit)) )
    {
    /* Taste gedrueckt */
    _delay_ms(20);
    if ( *inreg & (1 << inbit) )
      {
      /* Zeit zum Loslassen */
```

```
      _delay_ms(50);
      return 1;
      }
    }
  return 0;
  }
```

Die folgende Funktion verwendet keine feste Wartezeit, sondern versucht das Prellen zu detektieren. Dadurch kann eine kürzere Wartezeit (mindestens 8 * 1 ms = 8 ms) erreicht werden. Die Funktion testet den den gewünschten Pegel und nur, wenn dieser dem vorherigen entspricht, wird eine 1 in des Schieberegister `delayline` geschoben. Erst wenn der Pegel acht mal konstant war, wird die Schleife verlassen. Solange der Pegel des Tasters immer wieder wechselt, tauchen auch Nullen im Schieberegister auf und sorgen für eine Verlängerung der Wartezeit. Vorteil dieser Variante ist, dass nur so lange gewartet wird wie nötig (gegebenenfalls kann man den Delay-Wert auch noch herabsetzen):

```
void waitkey(volatile uint8_t *inreg, uint8_t inbit)
  {
  uint8_t prev, value;
  uint8_t delayline = 0;

  value = *inreg & (1 << inbit);   /* Portbit lesen */
  while(delayline != 0xff)         /* 11111111 erreicht? */
    {
    delayline <<= 1;               /* schieben */
    prev = value;                  /* alten Wert merken */
    _delay_ms(1);                  /* warten */
    value = *inreg & (1 << inbit); /* Port lesen */
    if(value == prev)              /* alter Wert == neuer Wert */
       delayline |= 0x01;
    }
  }
```

Will man mit noch weniger Wartezeit auskommen, muss man ein Verfahren wählen, das sich ganz allgemein innerhalb der Hauptschleife eignet. Man „verschmiert“ die Erkennung einer Taste auf mehrere Schleifendurchläufe. In gewisser Weise ist das ein winzig kleines kooperatives Multitasking: jede Aktion/Abfrage wird in der Schleife kurz bearbeitet, wobei oft mehrere Schleifendurchläufe für die vollständige Bearbeitung nötig sind. Es muss nur auf irgendeine Art und Weise der Zustand des gesamten Systems gespeichert und in jedem Durchlauf aktualisiert werden, womit schon ein Bogen zum Abschnitt über Statemaschinen (Seite 29) gespannt wurde.

Bleiben wir aber erst einmal bei der Tastenentprellung und modifizieren wir das letzte Beispiel, indem die `while`-Schleife durch eine Ergebnisausgabe ersetzt wird. Die Funktion liefert nun eine Information darüber zurück, ob das gewünschte Ergebnis (`delayline == 0xff`) schon erreicht wurde:

```
uint8_t keypressed(volatile uint8_t *inreg, uint8_t inbit)
  {
  uint8_t prev, value;
  static uint8_t delayline = 0;    /* Schieberegister */
  static uint8_t ff = 0;           /* Flip-Flop */

  value = *inreg & (1 << inbit);   /* Portbit lesen */
  if(delayline == 0xff && ff == 0) /* 11111111 erreicht? */
    {
    ff = 1;                        /* Zustand "gedrueckt" */
    return 1;
    }
  if(delayline == 0 && ff == 1)    /* 00000000 erreicht? */
    {
    ff = 0;                        /* Zustand "losgelassen" */
    return 0;
    }
  delayline <<= 1;                 /* schieben */
  prev = value;                    /* alten Wert merken */
  _delay_ms(1);                    /* warten */
  value = *inreg & (1 << inbit);   /* Port lesen */
  if(value == prev)                /* alter Wert == neuer Wert */
     delayline |= 0x01;
  return 0;
  }
```

die Funktion `keypressed()` „verbraucht“ nun nur noch etwas mehr als 1 ms oder sogar noch weniger, wenn man die Delayzeit noch vermindert. Sie dürfen nun nur nicht im Hauptprogramm die Bemühungen zunichte machen, indem Sie einfach eine Schleife der Form `while(! keypressed(...))` bilden. Vielmehr muss `main()` etwa so aussehen:

```
   ...
if (keypressed(PORT, BIT))
  {
  /* Taste bearbeiten */
  }
else
  {
  /* sonstwas machen */
  }
   ...
```

Eine weitere Möglichkeit ist die Flankenerkennung. Vom Prinzip her existieren bei einer Taste vier mögliche Zustände:

1. OFF: Taste ist nicht gedrückt und war vorher auch nicht gedrückt
2. RISING: Taste ist gedrückt und war vorher nicht gedrückt (steigende Flanke)
3. ON: Taste ist gedrückt und war vorher auch gedrückt
4. FALLING: Taste ist nicht mehr gedrückt und war vorher gedrückt (fallende Flanke)

Diese Zustände lassen sich in einer kleinen Statemachine behandeln. Die Entprellung geschieht dabei durch die Laufzeit des Programms zwischen zewi Aufrufen von `check_taste()`. Die folgende Funktion gibt für den Zustand RISING = „steigende Flanke“ den Wert „1“, für FALLING = „fallende Flanke“ den Wert „-1“ und sonst „0“ zurück. Die Taste ist wie üblich „active low“ (mit Pullup-Widerstand), „0“ bedeutet demnach, dass die Taste gedrückt ist.

Bei einen typischen Controller wird meist ein Bit eines Ports ausgelesen und entsprechend maskiert. Das könnte dann folgendermassen aussehen:

```
/* Bit y von Port x auslesen, an dem die Taste haengt,
   liefert False (gedrueckt) oder True (offen) */
#define CHECK_KEY (PORTx & (1 << BITy))
```

Bei anderen Systemen ist jedes Bit eines Ports einzeln zugänglich (z. B. bei Arduino oder Raspberry Pi). In diesem Fall können die Makros dann einfacher sein:

```
/* Bit von GPIOx auslesen, an dem die Taste haengt,
   liefert False (gedrueckt) oder True (offen) */
#define CHECK_KEY (GPIOx)
```

Für die Zustände definiert man einen Aufzählungstyp:

```
typedef enum States
  { OFF = 0,
    RISING = 1,
    ON = 2,
    FALLING = 3
  } states_t;
```

Der Zustandsautomat für die Taste wird dann als Funktion definiert, die immer wieder in der Hauptschleife des Programms aufgerufen wird. Die Entprellung erfolgt, wie schon erwähnt, durch die Laufzeit des restlichen Programms. Der Ausdruck `!CHECK_KEY` ist negiert, weil der Tastendruck ja „0“ liefert.

```
int check_taste(void)
  {
  static states_t state = OFF;
  int ret_val = 0;

  if(state == OFF && !CHECK_KEY)   /* Taste gedrueckt (steigende Flanke) */
    {
    state = RISING;
    ret_val = 1;
    }
  else if (((state == RISING) || (state == ON)) && !CHECK_KEY)   /* Taste gehalten */
```

```
  {
  state = ON;
  ret_val = 0;
  }
else if (state == ON && CHECK_KEY)   /* Taste losgelassen (fallende Flanke) */
  {
  state = FALLING;
  ret_val = -1;
  }
else if (state == FALLING && CHECK_KEY)   /* Taste unbetaetigt */
  {
  state = OFF;
  ret_val = 0;
  }
return ret_val;
}
```

6.2 Über den Tellerrand blicken

Zwölf Tasten an nur einem Pin: Wie soll das gehen? Gar nicht so selten tritt der Fall auf, dass bei einer Controller-Appliance einfach zu wenige Ports zur Verfügung stehen. Die Digitalports sind alle belegt und dort werden auch schon durch Ausgabemultiplexen Ports gespart soweit nur irgend möglich. Man kann zwar fast immer problemlos auch alle Analogeingänge als Digitaleingänge nutzen, aber was tun, wenn die auch schon größtenteils belegt sind. Auch ein Wechsel auf den nächstgrößeren Controller ist nicht immer möglich, sei es aus Platz- oder Kostengründen, weil die Boards schon in der Fertigung sind oder weil durch die Anpassung der Software an den neuen Typ der Zeitplan überschritten wird.

Die folgende Anwendung zeigt, wie sich an nur einem Analogport ein Tastenfeld mit zwölf Tasten anschließen und auswerten lässt. Betrachten Sie dazu das Schaltbild 6.1. Die Tasten sind als 3x4-Matrix angeordnet. An die Anschlüsse: der Tstenmatrix werden zwei Widerstandsnetzwerke angelötet, sodass die Tastatur nur noch über drei Leitungen mit dem Controller verbunden ist: Vcc, GND und ein Analogausgang. Der Kondensator am Ausgang soll Störungen unterdrücken.

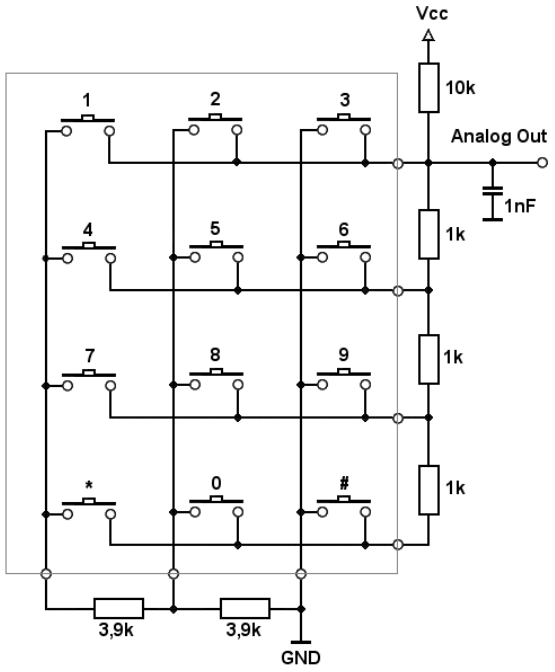

Bild 6.1: Schaltung der Tastatur

Nun wird eine Tabellenkalkulation bemüht und der Widerstand ausgerechnet, der sich beim Drücken einer jeden Taste gegen GND ergibt (Tabelle 6.1. Da noch ein weiterer Widerstand 10 kOhm gegen

Vcc geschaltet ist, ergibt sich ein Spannungsteiler, dessen Ausgangsspannung in der dritten Spalte der Tabelle aufgelistet ist:

Tabelle 6.1: Widerstandswerte der Tasten

Taste	Widerstand gegen GND	Ausgangs-Spannung
1	7,8	2,19
2	3,9	1,40
3	0,0	0,00
4	8,8	2,34
5	4,9	1,64
6	1,0	0,45
7	9,8	2,47
8	5,9	1,86
9	2,0	0,83
*	10,8	2,60
0	6,9	2,04
#	3,0	1,15

Nun wird die Tabelle zuerst nach der Ausgangsspannung sortiert und dann um weitere Spalten ergänzt, was Tabelle 6.2 ergibt. In der vierten Spalte wird die Spannungsdifferenz zwischen zwei (spannungsmäßig) aufeinander folgender Tasten ermittelt. Die Werte werden zwar nicht direkt für die Programmierung gebraucht, geben aber Aufschluss darüber, ob es eventuell irgendwo „eng“ wird. Interessant für die Programmierung sind die letzten beiden Spalten. In der vorletzten Spalte wird aus der Ausgangsspannung der entsprechende Ausgangswert des A/D-Wandlers (10 Bit) errechnet (Vcc = 5 V). Die letzte Spalte gibt die Schwellen zwischen zwei (spannungsmäßig) nebeneinander liegenden Tasten an. Sie wurde bis auf die ersten beiden Werte berechnet, indem zum errechneten Wandlerwert die Hälfte der Differenz zum nächsten Wert addiert wurde. Ist keine Taste gedrückt, liegt der Analogeingang über den 10-kOhm-Widerstand an Vcc und der Wandler sollte einen Wert nahe 1023 liefern. Die Schwelle kann man hier großzügig wählen.

Tabelle 6.2: Widerstandswerte der Tasten (erweitert)

Taste	Widerstand gegen GND	Ausgangs-Spannung	Differenz zum Vorgänger	Wandler-Wert	Schwelle zum Nachfolger
3	0,0	0,00	0	0	70
6	1,0	0,45	0,45	92	131
9	2,0	0,83	0,38	170	203
#	3,0	1,15	0,32	236	262
2	3,9	1,40	0,25	287	312
5	4,9	1,64	0,24	336	359
8	5,9	1,86	0,22	381	400
0	6,9	2,04	0,18	418	434
1	7,8	2,19	0,15	449	464
4	8,8	2,34	0,15	479	493
7	9,8	2,47	0,13	506	519
*	10,8	2,60	0,13	532	600

Ein Nachteil dieser Schaltung sei nicht verschwiegen: Drückt man zwei Tasten gleich-zeitig, ergibt sich u. U. nicht der richtige Wert. Auch ist die Störsicherheit nicht so groß wie bei einer rein digitalen Tastatur, in der Regel aber mehr als ausreichend.

Im Programm (diesmal für Arduino) werden Schwellenwert und die zugehörige ASCII-Repräsentation der Taste in einem zweidimensionalen Array gespeichert. Um einen Analogwert in einen Tastencode umzuwandeln, wird das Array in einer Schleife so lange durchlaufen, bis der Analogwert kleiner als der Schwellenwert ist. Dann wird der zugehörige Code zurückgegeben. Diese Umwandlung

erledigt die Funktion `getkey()`. Die übergeordnete Funktion `readKey()` entspricht der gleichen Funktion für eine digitale Eingabe.

Um das Loslassen einer Taste erkennen zu können, wird die aktuell erkannte Taste in einer Variablen gespeichert. Ein neuerlicher Tastendruck wird dadurch erkannt, dass die aktuell betätigte Taste ungleich der gespeicherten ist. Das funktioniert auch, wenn eine Taste mehrmals hintereinander betätigt wird, das zwischendurch NOKEY als Wert auftritt. `readKey()` wartet auch nicht, bis eine Taste gedrückt wurde, sondern kehrt immer sofort zum aufrufenden Programm zurück. Ist keine Taste gedrückt, liefert sie NOKEY als Ergebnis.

```
// Portnummer der internen LED
#define LED 13

// Anzahl Tasten
#define NUM_KEYS 12

// Schwellenwert fuer 'keine Taste gedrueckt'
#define NIL 600

// Code fuer 'keine Taste'
#define NOKEY -1

// Zuordnung Schwellenwerte - Tasten
int key_val[NUM_KEYS][2] =
  { { 70, '3'}, {131, '6'}, {203, '9'},
    {262, '#'}, {312, '2'}, {359, '5'},
    {400, '8'}, {434, '0'}, {464, '1'},
    {493, '4'}, {519, '7'}, {600, '*'} };

int taste;

void setup()
  {
  Serial.begin(9600);
  pinMode(13, OUTPUT);              // Debug-LED als Anzeige
  }

void loop()
  {
  // Tastatur lesen
  taste = readKey();
  if (taste != NOKEY)
    {
    Serial.println(taste);          // Testausgabe
    }
  }


// Taste einlesen. Liefert NOKEY, falls keine Taste gedrueckt
// wurde, andernfalls den Tastencode
int readKey()
  {
  int analogwert;                   // Eingabewert von Tastatur
  int key = NOKEY;                  // aktuelle Taste
  static int oldkey = NOKEY;        // Merker fuer letzte Taste (static!)

  analogwert = analogRead(0);       // Tasten-Spannung lesen
  key = getkey(analogwert);         // in Tastencode umsetzen
  if (key != NOKEY)                 // Taste gedrÃ¼ckt?
    {
    digitalWrite(LED, HIGH);        // Tastensignal
    if (key != oldkey)              // neue Taste gedrÃ¼ckt?
      {
      delay(100);                   // Entprellzeit
      analogwert = analogRead(0);   // Tasten-Spannung lesen
      key = getkey(analogwert);     // in Tastencode umsetzen
      if (key != oldkey)           // neue Taste!
        oldkey = key;
      else
        key = NOKEY;                // key = oldkey: Taste gehalten
      }
    digitalWrite(LED, LOW);
    }
  return key;
```

```
  }

// Konvertieren Analogwert in Tastencode (ASCII)
int getkey(int input)
  {
  int k;
  // Schwellenwert suchen, Taste zurueckgeben
  for (k = 0; k < NUM_KEYS; k++)
    {
if (input < key_val[k][0])
      return key_val[k][1];
    }
  return NOKEY;  // input oberhalb oberster Schwelle
  }
```

Für die Dauer der Tastenbetätigung wird die auf dem Board vorhandene LED eingeschaltet. Man könnte anstelle dieser Anzeige auch einen Tasten-Piep implementieren.

6.3 LED als Ausgabegerät

Vielfach erfolgt die analoge Ausgabe nicht über einen D/A-Wandler, sondern per Pulsweitenmodulation (PWM). Bei Gleichstrommotoren ist das sogar von Vorteil, diese lassen sich über PWM wesentlich besser steuern, als mit einer analogen Spannung. Auch ist PWM mit wesentlich geringerer Verlustleistung verbunden. Will man eine „echte“ analoge Spannung, lässt sich diese über einen Tiefpaßfilter relativ einfach aus dem PWM-Signal gewinnen.

Das Arduino-Board hat die PWM-Ausgabe auf sechs seiner Pins direkt implementiert, weshalb es für die folgenden Beispiele zum Einsatz kommt. Das Ausgangstastverhältnis der PWM-Ports kann über die Funktion `analogWrite()` mit Werten zwischen 0 und 255 entsprechend zwischen 0 und 100 Prozent festgelegt werden. Für die folgenden Versuche werden drei LEDs mit den Farben Rot, Grün und Blau an die PWM-Ausgangspins 9, 10 und 11 angeschlossen. Es kann aber auch eine integrierte Dreifarben-LED verwendet werden.

Das erste Beispiel realisiert einen Soft-Blinker, indem die LED per PWM mit einem sinusförmigen Signal angesteuert wird. Per Programm wird in 1-Grad-Schritten der Sinus im Bereich von 0 bis 180° berechnet und mit 255 multipliziert. Da der Sinus in diesem Bereich nur Werte zwischen 0 und 1 annehmen kann, ergibt dass die ideale Ansteuerung für den Analogausgang.

```
// Pin der roten LED
#define LEDRot 9

// eine bekannte Konstante
#define Pi 3.14159265

int wert;      // Ausgabewert
int i;         // Schleifenzaehler

void setup()
  {
  // muss nicht sein, kann aber ...
  pinMode(ledPin, OUTPUT);
  }

void loop()
  {
  for (i = 0; i < 180; i++)
    {
    // Sinus berechnen (Achtung: Bogenmass!)
    wert = int(sin(i*Pi/180)*255);
    analogWrite(ledPin, wert);
    delay(10);
    }
  }
```

Das Programm belastet den Prozessor durch die Sinusberechnung relativ stark. Eine Alternative wäre ein Array mit den 180 Sinuswerten (gleich mit 255 multipliziert), auf das dann mit Index i zugegriffen wird.

Nachdem das so schön klappt, werden im nächsten Programm alle drei LEDs eingesetzt und ein Farbwechsel realisiert. In der Funktion `setLeds()` gibt es eine sinnvolle Anwendung der Speicherklasse

`static`. Die aktuellen LED-Farbintensitäten werden in den Variablen red, green und blue von Aufruf zu Aufruf weitergegeben. Ohne `static` funktioniert es nicht, es würde immer mit Rot begonnen werden. Die Funktion kann mit Werten zwischen 0 und 764 aufgerufen werden.

```
// Pinzuordnung der LEDs
#define LEDBlau  11
#define LEDGruen 10
#define LEDRot   9

void setup()
  {
  pinMode(LEDBlau, OUTPUT);
  pinMode(LEDGruen, OUTPUT);
  pinMode(LEDRot, OUTPUT);
  }

void loop()
  {
  int licht;
  for (licht = 0; licht < 765; licht++)
    {
    setLEDs(licht);
    delay(20);
    }
  }

void setLEDs(int i)
  {
  // Farbwerte mit Vorbesetzung, begonnen wird mit rot
  static int red   = 255;
  static int green = 0;
  static int blue  = 0;

  if (i < 255)      // Phase 1: von rot nach grün
    {
    red--;       // red down
    green++;     // green up
    blue = 0;    // blue low
    }
  else if (i < 510) // Phase 2: von grün nach blau
    {
    red = 0;     // red low
    green--;     // green down
    blue++;      // blue up
    }
  else if (i < 766) // Phase 3: von blau nach rot
    {
    red++;       // red up
    green = 0;   // green low
    blue--;      // blue down
    }
  analogWrite(LEDRot,   red);
  analogWrite(LEDGruen, green);
  analogWrite(LEDBlau,  blue);
  }
```

Ein kleiner Nachteil des Programms liegt darin, dass man keinen bestimmten Farbwert einstellen kann, sondern sich immer in einem Schleife bist zum gewünschten Wert vorarbeiten müsste.

Wenn sie die LEDs bei den beiden vorangegangenen Beispielen genau beobachten, stellen Sie fest, dass die Fsarbübergänge nicht gleichmäßig sind. Das legt daran, dass das Auge die Lichtintensität nicht linear, sondern eher logarithmisch wahrnimmt. Ist die LED ganz dunkel, werden kleine Änderungen gut wahrgenommen, ist die LED dagegen schon ziemlich hell, merkt man die Zu- oder Abnahme erst nach mehreren Schritten.

Um das auszugleichen, kann bei der Ausgabe der Farbwert entsprechend angepasst werden. Dies erfolgt am Besten mithilfe einer Umrechnungstabelle, die in einem Array gespeichert wird. Im Programm wird eine Tabelle aus dem Buch von Odendahl et al.: „Arduino - Physical Computing für Bastler, Designer & Geeks“, O'Reilly, 2010, stammt und leicht modifiziert wurde. Die Umcodierung erfolgt in der Funktion `setColor()`, in der auch die Ausgabe erfolgt. In `setLEDs()` wird dann der auszugebende Farbwert berechnet. Im Gegensatz zum vorhergehenden Programm kann hier jeder individuelle Wert gesetzt werden. Für die Eingabewerte von 0 bis 255 durchläuft die Funktion einmal den Farbkreis.

```
// Pinzuordnung der LEDs
#define LEDBlau  9
#define LEDGruen 10
#define LEDRot   11

void setup()
  {
  pinMode(LEDBlau, OUTPUT);
  pinMode(LEDGruen, OUTPUT);
  pinMode(LEDRot, OUTPUT);
  }

void loop()
  {
  int licht;
  for (licht = 0; licht < 256; licht++)
    {
    setLEDs(licht);
    delay(100);
    }
  }

// Wählt für einen Wert (0 .. 255) eine Farbe aus dem Farbkreis
void setLEDs(int value)
  {
  int red, green, blue;

  if(value < 64)           // rot nach gruen
    {
    red = 63 - value;
    green = value;
    blue = 0;
    }
  else if(value < 128)     // gruen nach blau
    {
    red = 0;
    green = 127 - value;
    blue = value - 64;
    }
  else if(value < 192)     // blau nach rot
    {
    red = value - 128;
    green = 0;
    blue = 191 - value;
    }
  else                     // rot nach weiss
    {
    red = 63;
    green = 255 - value;
    blue = 255 - value;
    }
  setColor(red, green, blue);
  }

// Stellt die LED-Helligkeiten logarithmisch ein
void setColor(int red, int green, int blue)
  {
  // Tabelle der Helligkeitswerte,
  // lograrithmisch ansteigend
  int logValue[64] =
    { 0,  1,  2,  3,  4,  5,  6,  7,
      8,  9, 10, 11, 12, 13, 14, 16,
     18, 20, 22, 25, 28, 30, 33, 36,
     39, 42, 46, 53, 56, 60, 64, 68,
     72, 77, 81, 86, 90, 95,100,105,
     110,116,121,127,132,138,144,150,
     156,163,169,176,182,189,196,203,
     210,218,225,233,240,248,253,255  };

  analogWrite(LEDBlau, logValue[blue]);
  analogWrite(LEDGruen, logValue[green]);
  analogWrite(LEDRot, logValue[red]);
  }
```

Manchmal ist das RGB-Farbschema jedoch hinderlich. Angenommen, Sie wollen – wie bei manchen Wetterberichten – die aktuelle Temperatur durch eine Farbe repräsentieren: vom kalten Blau bei Minustemperaturen bis zum warmen Rot bei sommerlicher Hitze. Dann wäre es viel praktischer, wenn man im Programm nur einen einizgen Wert für die Farbe hätte und nicht drei. Genau das erreichen Sie mit dem folgenden Farbschema.

Der HSV-Farbraum beschreibt eine Farbe mit Hilfe des Farbtons (englisch hue), der Farbsättigung (saturation) und des Hellwerts (value). Ähnliche Definitionen führen zu einem HSL-Farbraum mit der relativen Helligkeit (lightness), einem HSB-Farbraum mit der absoluten Helligkeit (brightness) und einem HSI-Farbraum mit der Lichtintensität (intensity).

Bei der Farbdarstellung wird der HSV-Farbraum gegenüber den Alternativen RGB (Rot, Grün, Blau) und CMYK (Cyan, Magenta, Yellow, Black) bevorzugt, weil es der menschlichen Farbwahrnehmung ähnelt. So fällt es leichter, eine Farbe zu finden: Man kann für die Farbmischung unmittelbar den Farbton wählen und dann entscheiden, wie gesättigt und wie hell (oder dunkel) dieser sein soll, oder ob eine andere Farbnuance passender ist.

Für die Beschreibung des Farbortes in diesem Farbraum werden folgende Parameter benutzt:

- Farbton (Hue) als Winkel H auf dem Farbkreis (z. B. 0° = Rot, 120° = Grün, 240° = Blau). Die Farbtoneinstellung erfolgt durch Weiterdrehen auf eine der benachbarten Farbwerte. So kann z. B. ein Blau in Richtung Cyan und Grün oder in Richtung Violett und Rot verschoben werden. Der Farbton wird als Position auf dem Standard-Farbkreis angegeben und daher in Werten zwischen 0° und 360° ausgedrückt.
- Sättigung (Saturation) S in Prozent (0 % = Neutralgrau, 50 % = wenig gesättigte Farbe, 100 % = gesättigte, reine Farbe). Alternativ wird ein Intervall von 0 bis 1 verwendet. Vom Sättigungsgrad einer Farbe ist es abhängig, ob wir einen Farbton als satt und kräftig oder als matt und schwach empfinden. Sie beschreibt also das Verhältnis zwischen Farbe und Grauanteil. Auf dem Farbkreis nimmt die Sättigung vom Rand zur Mitte hin zu.
- Helligkeit (Brightness) V als Prozentwert (0 % = keine Helligkeit, 100 % = volle Helligkeit). Alternativ wird auch hier ein Intervall von 0 bis 1 verwendet, das auch „Dunkelstufe“ genannt wird.

Das folgende Programm rechnet die HSV-Angaben in Werte für Rot, Grün und Blau um, die dann zur Ansteuerung der LEDs dienen. Weitere Informationen, Grafiken und die Umrechnungsformeln finden Sie bei der deutschen Wikipedia unter dem Link *http://de.wikipedia.org/wiki/HSV-Farbraum* bzw. noch ausführlicher in der englischen Version unter *http://en.wikipedia.org/wiki/HSV_color_space*. Das Programm weicht vom oben angegebenen Schema insofern ab, als dass anstelle der Prozentangaben bereits die bei der Ausgabe möglichen Werte (0 ... 255) verwendet werden. Auf diese Weise kommt das Programm auch mit Integer-Arithmetik aus, was der Rechenzeit und dem Speicherbedarf zu Gute kommt.

```
// Pinzuordnung der LEDs
#define LedBlau  9
#define LedGruen 10
#define LedRot   11

void setup()
  {
  pinMode(LedBlau, OUTPUT);
  pinMode(LedGruen, OUTPUT);
  pinMode(LedRot, OUTPUT);
  }

void loop()
  {
  int licht;
  for (licht = 0; licht < 360; licht++)
    { // kompletten Farbkreis durchlaufen
    setLED(licht,255);
    delay(100);
    }
  }

void setLED(int hue, int l)
  {  // LED-Farben festlegen nach HUE und Intensität.
     // Sättigung ist hier immer auf Maximum gesetzt
```

```
  int col[3] = { 0,0,0 };

  getRGB(hue, 255, l, col);             // HSV in RGB umrechen
  analogWrite(LedRot, 255 - col[0]);    // und ausgeben
  analogWrite(LedGruen, 255 - col[1]);
  analogWrite(LedBlau, 255 - col[2]);
  }

void getRGB(int hue, int sat, int val, int colors[3])
  {  // Diese Funktion rechent einen HSV-Wert in die ensprechenden
     // RGB-Werte um. Diese werden im Array 'colors' zurückgegeben
     // colors[0] = ROg, colors[1]} = Gruen, colors[2] = Blau
     // hue: 0 - 359, saturation: 0 - 255, val (lightness): 0 - 255
  int red, green, blue, base;

  if (sat == 0)
    { // Sättigung = 0 --> Grauwert
    colors[0] = val;
    colors[1] = val;
    colors[2] = val;
    }
  else
    {
    base = ((255 - sat) * val) >> 8;
    if (hue < 60)
      {
      red = val;
      green = (((val - base)*hue)/60) + base;
      blue = base;
      }
    else if (hue < 120)
      {
      red = (((val - base)*(60-(hue%60)))/60) + base;
      green = val;
      blue = base;
      }
    else if (hue < 180)
      {
      red = base;
      green = val;
      blue = (((val - base)*(hue%60))/60) + base;
      }
    else if (hue < 240)
      {
      red = base;
      green = (((val - base)*(60 - (hue%60)))/60) + base;
      blue = val;
      }
    else if (hue < 300)
      {
      red = (((val - base)*(hue%60))/60) + base;
      green = base;
      blue = val;
      }
    else if (hue < 360)
      {
      red = val;
      green = base;
      blue = (((val - base)*(60 - (hue%60)))/60) + base;
      }
    colors[0] = red;
    colors[1] = green;
    colors[2] = blue;
    }
  }
```

6.4 Beschleunigungssensoren auswerten

Mit Beschleunigungssensoren (Accelerometern) kann man nicht nur die Beschleunigung bewegter Körper messen, sondern wir erhalten auch ein Signal, wenn der Sensor in Ruhe ist. Dann ist die einzige Beschleunigung, die der Beschleunigungsmesser erkennt, aufgrund der Schwerkraft nach unten

gerichtet (Bild 6.2). Wie Sie sehen, sind nur Ausgangswerte des Beschleunigungssensors im Bereich vom ±180° sinnvoll, Werte darüber sind nur ein Spiegelbild. Trotzdem lassen sich auch Richtungen im 360-Grad-Radius erfassen. Der Trick besteht dabei im Zusammenspiel aller drei Achsen.

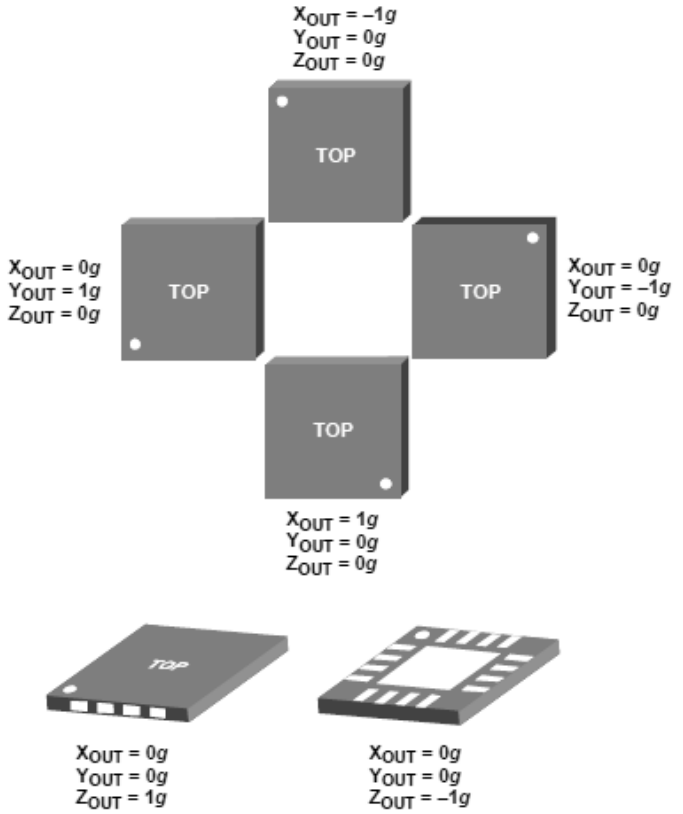

Bild 6.2: Ausgabe des Sensors je nach Lage im Raum

Beispielhaft für diverse Beschleunigungssensoren wird hier der ADXL335 verwendet. Es handelt such um einen Dreiachsen-Beschleunigungssensor mit Analogausgang von Analog Devices. Der ADXL335-Messbereich beträgt ±3 g. Die drei Analog-Ausgänge für X-, Y- und Z-Achse liefern eine zur Beschleunigung proportionale Spannung für jede Achse. Für den Mittenwert der Beschleunigung 0 g beträgt die Ausgangsspannung typischerweise die halbe Versorgungsspannung.

Bei Messungen auf der Erde können prinzipiell immer nur zwei Achsen berücksichtig werden. Machen wir ein Gedankenexperiment: Legen Sie den Sensor flach auf den Boden. Egal wie Sie den Sensor drehen, der Z-Wert bleibt immer gleich. Wenn Sie die Z-Achse (Yaw, siehe unten) auch messen wollten, würden Sie einen Gyrometer-Sensor benötigen. Wird der Sensor nun um die Längs- oder Querachse gekippt, misst er jeweils nur den entsprechenden Anteil der Erdbeschleunigung, woraus sich der Neigungswinkel bestimmen lässt. In der Praxis wird man den Chip aber nie ausschließlich um die eine oder die andere Achse kippen (Bild 6.2). Hier kommt die Messung in Z-Richtung ins Spiel, mit der man die Abweichung von der Waagerechten bestimmen kann. Aus allen drei Beschleunigungswerten lassen sich die Neigungswinkel in X- und Y-Richtung sauber berechnen. Der Chip darf aber nicht auch noch in eine Raumrichtung beschleunigt, sondern nur gekippt werden. Wenn er erschüttert wird (Stoß, Vibration, freier Fall etc.), basiert die Messung des Sensors nicht mehr auf der reinen Schwerkraft und die Messung ist fehlerhaft. Oft werden die drei Achsen mit Begriffen aus der Luftfahrt bezeichnet, wie es in Bild 6.3 gezeigt ist.

Nun kommt etwas Mathematik. Für die vom Sensor auf die Eingänge ADCl, ADC2 und ADC3 gelieferten Digitalwerte gilt wegen der (typischerweis vorhadenen) ADC-Auflösung von 10 Bit:

$$Val_{x,y,z} = U_{x,y,z} * 1024 / U_{ref} \tag{6.1}$$

mit U_{ref} = 3,3 V. Ein Wert von U = $U_{ref}/2$ = 1,65 V entspricht dabei laut Datenblatt einer Beschleunigung von 0 g. In der Praxis muss man das System vor jeder Messreihe kalibrieren, was in diesem Fall bedeutet, die realen Maximal- und Minimalwerte (für jede der drei Achsen) zu bestimmen.

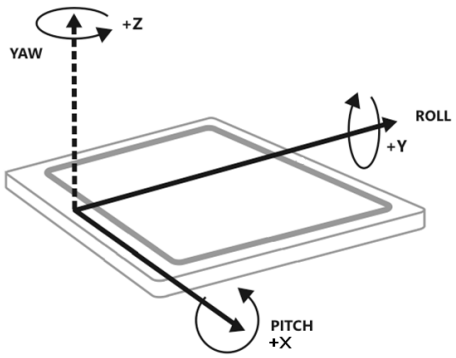

Bild 6.3: Yaw, Pitch und Roll bezeichen die Z-, X- und Y-Achse

Die drei Achsen werden einzeln kalibriert. Im Falle der X- und Y-Achsen wird die Platine mit dem Sensor einmal in beide Richtungen um 90° gekippt (am besten auf eine feste, waagrechte Unterlage legen). Am Ende sind alle Werte für $ADCX_{max}$, $ADCY_{max}$ und $ADCZ_{max}$ sowie für $ADCX_{min}$, $ADCY_{min}$ und $ADCZ_{min}$ gespeichert.

Zur Glättung eines Messwerts wird der Mittelwert von 16 aufeinander folgenden ADC-Werten genommen, andernfalls könnten kleine Vibrationen des Sensors zu Messfehlern führen. Mit den aktuellen, gemittelten Messwerten berechnet man:

$$Gval_{x,y,z} = \frac{ADCval_{x,y,z} - ADCmid_{x,y,z}}{ADCmax_{x,y,z} - ADCmin_{x,y,z}} \tag{6.2}$$

mit $ADCmid_{x,y,z} = (ADCmax_{x,y,z} + ADCmin_{x,y,z})/2$.

$4Gval_x$, $Gval_y$ und $Gval_z$ ist die gemessene Beschleunigung entlang der drei Achsen. In der Application Note AN3461 von Freescale wird umfassend beschrieben, wie aus den obigen Werten die Winkel *pitch*, *roll* und *yaw* berechnet werden können (*http://www.freescale.com/files/sensors/doc/app_note/AN3461.pdf*):

$$tan(pitch) = Xgval/sqrt(Ygval^2 + Zgval^2) \tag{6.3}$$

$$tan(roll) = Ygval/sqrt(Xgval^2 + Zgval^2) \tag{6.4}$$

$$tan(yaw) = sqrt(Xgval^2 + Ygval^2)/Zgval \tag{6.5}$$

pitch: Winkel der Drehung um die Längsachse, waagerecht ist pitch = O, positive Werte ergeben sich bei einer Drehung im Uhrzeigersinn.
roll: Winkel beim Kippen nach vorn bzw. hinten, waagerecht ist roll = O, positive Werte ergeben sich bei einem Kippen nach vorn.
yaw: Winkel der Abweichung von der Waagerechten, waagerecht ist yaw = O, positiv bei jeder Drehung oder jedem Kippen.

Der folgende Code für den Arduino berechnet die Winkel für Pitch, Roll and Yaw. Dieser Code ist nicht spezifisch für den ADXL335, er läßt sich auf einen beliebigen analogen Dreiachsen-Beschleunigungsmesser anpassen.

```
// Konstante (sollten in der Bibliothek vorhanden sein)
// #define PI 3.14159265
// #define RAD_TO_DEGREE(r) ((r *180.0) / PI)

// Analoge Datenpins des Arduino
#define xPin 0
#define yPin 1
#define zPin 2

// Zu erwartende Minimal- und Maximalwert
// -- muss gegebenenfalls angepasst werden --

int minVal = 260;
int maxVal = 410;

// Datenwerte fuer die Berechnung
```

```
double x;
double y;
double z;


void setup()
  {
  Serial.begin(9600);
  }

// 16 aufeinanderfolgende Analogwerte lesen,
// und Mittelwert zurueckgeben
int get_pin(const int Pin)
  {
  long sum = 0;
  int i;
  for (i = 0; i < 16; i++)
    sum = sum + analogRead(Pin);
  return sum >> 4;  // Division durch 16
  }

void loop()
  {
  // Beschleunigungswerte einlesen
  int xRead = get_pin(xPin);
  int yRead = get_pin(yPin);
  int zRead = get_pin(zPin);

  // Anpassen auf den Bereich -90 .. +90 (wegen atan2()
  int xAng = map(xRead, minVal, maxVal, -90, 90);
  int yAng = map(yRead, minVal, maxVal, -90, 90);
  int zAng = map(zRead, minVal, maxVal, -90, 90);

  // Winkel brechnen (atan2() liefert den Bereich von -Pi bis PI)
  // Umrechnen in Grad
  x = RAD_TO_DEG * (atan2(-yAng, -zAng) + PI);
  y = RAD_TO_DEG * (atan2(-xAng, -zAng) + PI);
  z = RAD_TO_DEG * (atan2(-yAng, -xAng) + PI);

  // Fuer den Test Ergebnisse seriell ausgeben
  Serial.print("Pitch (x): ");   Serial.print(x);
  Serial.print(" | Roll (y): "); Serial.print(y);
  Serial.print(" | Yaw (z): ");  Serial.println(z);

  delay(100);
  }
```

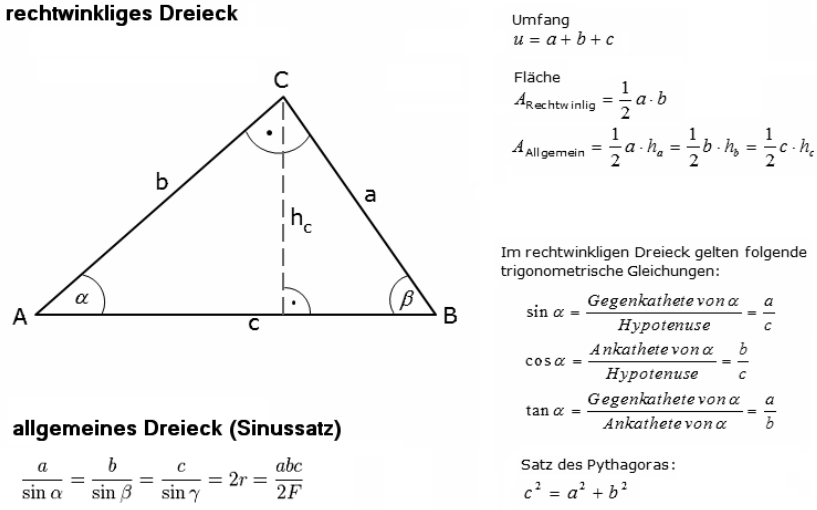

Bild 6.4: Die zugrundeliegenden Trigonometrie-Formeln

Die Analogeingänge liefern Werte zwischen 0 und 1023. Da der ADXL335 für maximal ± 3 g ausgelegt ist erhält man nur einen Ausschnitt des Wertebereichs. Vor ein numerisches Problem stellt den Programmierer noch die Funktion `atan2()`, die einen Bereich von -90° bis 90° bevorzugt. Deshalb

„mappe“ ich die Analogwerte auf diesen Bereich. Nicht zu vergessen, dass die trigonometrischen Funktionen im Bogenmass arbeiten, weshalb noch eine Umrechnung in Grad erfolgen muss:

Eine schnell zu realisierende Anwendung wäre das Ansteuern einer RGB-LED, wobei die Farbhelligkeit der drei Grundfarben von den drei Winkeln abhängig gesteuert wird.

Übrigens basiert die gesamte oben verwendete Mathematik auf der Dreiecksberechnung (Trigonometrie, Schulmathematik). Das Bild 6.4 fasst alle Gleichungen zusammen.

7

Programmierung der seriellen Schnittstelle

In Linux-Systemen werden die seriellen Schnittstellen über `/dev/ttySx` angesprochen, wobei hier bei x mit Null mit dem Zählen begonnen wird. Die serielle Emulation von USB-Geräten heißt dann meist /dev/ttyUSBx. Bei Windows heißen die Schnittstellen `COMx`, wobei hier das x mit 1 beginnt. `/dev/ttyS0` von Linux entspricht demnach `COM1` von Windows. Unter Linux/Unix kann man auch einen Shell-Login auf eine serielle Schnittstelle definieren, was für wichtige Server-Systeme einen Notzugang im Fehlerfall ermöglicht.

Die nötigen Grundlagen für die Programmierung liefern die beiden HOWTO-Dokumente, das „Linux Serial HOWTO“ von David Lawyer, das „Linux Serial Programming HOWTO“ von Peter H. Baumann und das „Text-Terminal-HOWTO“ von David Lawyer (Die HOWTOs werden bei den meisten Distributionen schon mit installiert oder lassen sich über das Dokumentationspakete nachinstallieren - sie sind aber auch im Internet oder hier auf der Webseite zu finden). Da es sich bei den seriellen Schnittstellen um normale Gerätedateien handelt, gelten natürlich auch die entsprechenden Regeln der Programmierung für das Ansprechen von Geräten. Dafür können wir auf POSIX-standardisierte Funktionen zurückgreifen. Eine sehr gute Beschreibung finden Sie im GNU C-Library Reference Manual, welches den meisten Distributionen beiliegt (dort Kap. 12). Hier will ich Ihnen eine kurze Zusammenfassung des Terminal-IO bieten, was für die meisten Anwendungen ausreichen sollte.

7.1 Allgemeines

Um mit einem User-Programm auf die serielle Schnittstelle zugreifen zu können, muss der entsprechende User auch die nötige Schreib- und Leseberechtigung für das Device haben (was normalerweise nicht der Fall ist). In der Regel reicht es, die User, die mit der seriellen Schnittstelle arbeiten, mit in die Gruppe *uucp* aufzunehmen (ggf. hilft ein Blick auf die Zugriffsrechte der Devices und in die Datei */etc/group*). Sinnvoll ist auch das Anlegen eines eigenen (Pseudo-)Users für die Steuerungsprogramme.

Fast alle Veränderungen an den Übertragungsparametern von Terminals oder seriellen Schnittstellen erzielen Sie mit der Struktur `termios` (Terminal-IO-Settings). Diese Struktur besteht aus fünf Teilen, nämlich vier 32-Bit-Masken für die verschiedenen Flags, der line discipline und dem c_cc-Array, das weitere Parameter, z. B. Wartezeiten nach dem Senden bestimmter Zeichen und die Definition von Steuerzeichen, aufnimmt (adressiert wird es über Präprozessor-Definitionen in */usr/include/termbits.h*). Sie hat demnach folgendes Aussehen:

```
struct termios
  {
  /* Bitmaske fuer die Eingabe-Flags */       tcflag\_t c\_iflag;
  /* Bitmaske fuer die Ausgabe-Flags */       tcflag\_t c\_oflag;
  /* Bitmaske fuer die Control-Flags */       tcflag\_t c\_cflag;
  /* Bitmaske fuer lokale Einstellungen */    tcflag\_t c\_lflag;
```

```
/* line discipline */                    char    \_\_c\_line;
/* Array fuer Sonderzeichen/-funktionen */  cc\_t c\_cc[NCCS];
}
```

Mit dem Shell-Kommando `stty -a </dev/ttyS0` können Sie sich jederzeit alle Werte dieser Struktur anzeigen lassen. Mit diesem Kommando kann man auch zahlreiche Parameter setzen. Für den Zugriff auf die `termios`-Daten gibt es zwei Bibliotheksfunktionen:

```
int tcgetattr(int filedes, struct termios *termios\_p)
/* liefert aktuelle Terminal-Settings */

int tcsetattr(int filedes, int when, const struct termios *termios\_p)
/* setzt Terminal-Settings */
```

Beiden Funktionen wird neben dem Zeiger auf eine Variable vom Typ `termios` auch der Dateideskriptor des Terminal-Devices übergeben. `tcsetattr` erwartet zusätzlich den Parameter `when`, mit dem sich festlegen lässt, wann die neuen Einstellungen übernommen werden sollen. Es gibt drei Möglichkeiten:

`TCSANOW:`	Einstellungen sofort ändern
`TCSADRAIN:`	Einstellungen ändern, nachdem alle eventuell noch gepufferten Daten gesendet wurden
`TCSAFLUSH:`	Einstellungen sofort ändern und Eingabepuffer löschen

`TCSAFLUSH` wäre also neben `TCSANOW` eine gute Wahl. Theoretisch ließe sich die Übertragungsgeschwindigkeit auch mit der `termios`-Struktur und `tcsetattr()` einstellen. Davon wird im GNU-Handbuch jedoch ohne Angabe von Gründen abgeraten. Zum Einstellen der Übertragungsgeschwindigkeit gibt es nämlich eine weitere Bibliotheksfunktion:

```
int cfsetspeed(struct termios *termios\_p, speed\_t speed)
```

Damit wird die Datenrate richtig in die Variable eingetragen. Für Hardware, die getrennte Einstellung von Sende- und Empfangsdatenrate erlaubt, gibt es übrigens noch die Funktionen `cfsetospeed()` und `cfsetispeed()`, die im Prinzip auch verwendet werden könnten (beim PC natürlich beide mit der gleichen Datenrate).

Alle genannten Funktionen liefern im Erfolgsfall den Wert 0 zurück und im Fehlerfall -1. Die Komplexität des Terminal-IO entsteht durch zahllose Flags, aus denen die vier Bitmasken zusammengesetzt sind. Die Flags und auch die `termios`-Struktur sind in der Datei */usr/include/termbits.h* definiert und unter anderem im Mini-HOWTO dokumentiert.

Normalerweise werden nicht alle Flags benötigt, um die serielle Schnittstelle zum Laufen zu bringen. Deshalb will ich im Folgenden nur die wichtigsten von ihnen behandeln. Beginnen wir mit den Eingabeflags in `c_iflag`:

```
IGNBRK   ignoriere Breaks
BRKINT   beachte Breaks
IGNPAR   ignoriere Parität
INLCR    ersetze NL durch CR
IGNCR    ignoriere CR
ICRNL    ersetze CR durch NL
IUCLC    Großbuchstaben in Kleinbuchstaben umwandeln
IXON     XON/XOFF-Flusssteuerung einschalten
IXANY    Ausgabe fortsetzen mit einem beliebigen Zeichen
IXOFF    XON/XOFF-Flusssteuerung ausschalten
IMAXBEL  akustisches Signal, wenn der Puffer voll ist (Zeilenende)
```

Die Ausgabeflags (`c_oflag`) sind noch wesentlich zahlreicher als die Eingabeflags, doch brauchen wir in der Regel nur wenige von ihnen. Meist reichen die folgenden:

```
ONLCR    ersetze NL durch CR
OCRNL    ersetze CR durch NL
ONOCR    Unterdrücken von CR in Spalte 0
ONLRET   eein CR senden
OFILL    Füllzeichen NUL senden anstelle einer Pause
OFDEL    Füllzeichen ist DEL statt NUL
```

Die dritte Gruppe von Flags, `c_cflag`, ist für die Übertragungsgeschwindigkeit und das Datenformat zuständig. Zunächst die Flags für die Geschwindigkeit:

```
B0      hang up       B50      50 bps
B75     75 bps        B110     110 bps
B150    150 bps       B300     300 bps
B600    600 bps       B1200    1200 bps
B1800   1800 bps      B2400    2400 bps
B4800   4800 bps      B9600    9600 bps
B19200  19200 bps     B38400   38400 bps
B57600  57600 bps     B115200  115200 bps
```

Die anderen Flags dieser Gruppe steuern das Datenformat:

```
CS5      5 Bit
CS6      6 Bit
CS7      7 Bit
CS8      8 Bit
CSTOPB   2 Stoppbits statt einem
CREAD    Empfangsteil aktivieren
PARENB   Paritätsbit erzeugen
PARODD   ungerade Parität statt gerader
HUPCL    Verbindungsabbruch bei Ende des letzten Prozesses
CLOCAL   Terminal lokal angeschlossen (ignoriere CD)
CRTSCTS  Hardware-Handshake einschalten
CIGNORE  ignoriere Controlflags
```

Aus der letzten Gruppe Flags (`c_lflag`) brauchen wir nur wenige:

```
ECHO    Einschalten der ECHO-Funktion
ICANON   Zeilenorientierter Eingabemodus (kanonischer Modus)
ISIG    bestimmte Sonderzeichen lösen ein Signal aus (\zb  Ctrl-C)
XCASE    Umwandeln von eingegebenen Groß- in Kleinbuchstaben
```

Grundsätzlich unterscheidet man beim Terminal-IO zwei Arten:

- **kanonischer Modus:** (cooked mode) Hier erfolgt das Lesen und Schreiben auf das Device zeilenorientiert. Eine Eingabe wird erst weitergereicht, wenn ein Zeilenabschluss (Linefeed, NL) oder Carriage Return (CR)) übertragen wurde. Für diesen Mode benötigt man die Steuerzeichen des `c_cc`-Arrays. Ein Programm wartet beim Lesen in diesem Modus so lange, bis tatsächlich eine komplette Zeile empfangen wurde. Wird kein Zeilenabschluss gelesen, so wird für immer und ewig gewartet. Die Aufgabe des Zwischenspeicherns übernimmt der Kernel.
- **nichtkanonischer Modus:** (raw mode) Hier wird nicht zeilenweise gelesen, sondern entweder auf eine bestimmte Anzahl von Bytes gewartet oder nach einer gewissen Zeit die bis dahin eingetroffenen Bytes abgeliefert. Hierfür müssen zwei Felder des Arrays `c_cc` gesetzt werden. In `c_cc[VTIME]` wird die Wartezeit in Zehntelsekunden und in `c_cc[VMIN]` das Minimum der zu lesenden Bytes angegeben.

Im nichtkanonischen Modus sind die folgenden vier Konstellationen möglich:

- **1. Fall: c_cc[VTIME] ¿ 0 und c_cc[VMIN] ¿ 0**
  `read()` liefert MIN Bytes, bevor die Zeit TIME abläuft oder `read()` liefert weniger als MIN Bytes, weil die Zeit TIME abgelaufen ist. Sind noch keine Daten empfangen worden, wartet `read()` auf min. ein Byte. Wenn das erste Byte gelesen wurde, läuft der Timer los, wobei jedes ankommende Byte den Timer wieder neu startet. Diese Methode ist günstig, wenn man große Datenmengen lesen, aber auch auf einzelne Zeichen reagieren muss. Aber es kann eine Blockierung stattfinden.
- **2. Fall: c_cc[VTIME] = 0 und c_cc[VMIN] ¿ 0**
  `read()` liefert mindestens MIN Bytes, sobald diese eingetroffen sind. Dieser Modus ist günstig, wenn möglichst viele Bytes mit einem `read()` gelesen werden sollen. Andererseits kann man auch auf ein einziges Byte reagieren (MIN = 1). Ist MIN größer als die Anzahl der bei `read()` angegebenen Zeichen, wird gewartet, bis MIN Bytes gelesen, aber nur n Bytes an `read()` geliefert wurden; ein zweites `read()` liefert dann den Rest. Auch hier kommt es zur Blockierung, wenn nicht genügend Bytes eintreffen.
- **3. Fall: c_cc[VTIME] ¿ 0 und c_cc[VMIN] = 0**
  Diese Einstellung erlaubt es, das Lesen mit Timeout zu programmieren. Sobald ein Byte eintrifft, liefert `read()` dieses ab. Wenn die Zeit TIME seit dem Aufruf von `read()` verstrichen ist, liefert `read()` 0 (gelesene Bytes) zurück.

- **4. Fall: c_cc[VTIME] = 0 und c_cc[VMIN] = 0**
  `read` liefert die Anzahl Bytes, die anliegen. Sind keine Daten vorhanden, wird sofort 0 (gelesene Bytes) zurückgegeben. Der Treiber wartet also niemals auf Daten, sondern kehrt immer sofort zurück.

Wenn das Programm nicht ewig auf eine Eingabe warten soll, nimmt man also am besten den dritten Fall.

Die im Folgenden verwendeten Funktionen zum Öffnen, Lesen, Schreiben und Schließen der Geräteschnittstelle sind im Skript zur C-Programmierung ausführlicher beschrieben:
http://www.netzmafia.de/skripten/programmieren/ad8.html#5.9

7.2 Serielle Schnittstelle öffnen

Da es sich bei den seriellen Schnittstellen nicht um normale Dateien handelt, können beim `open()`-Aufruf gegebenenfalls dateiuntypische Fehler auftreten. So kann zum Beispiel der Treiber den Fehlercode `EBUSY` zurückmelden, wenn gerade ein anderer Prozess das Device benutzt. Oder er hält das Programm so lange an („blocking open“), bis die Carrier-Leitung des Modems aktiv wird (was bei direkt angeschlossenen Geräten durch das Fehlen dieser Leitung scheinbar auftritt). Es gibt jedoch einen Mechanismus, um das Blockieren zu umgehen: beim `open()`-Aufruf muss das Flag `O_NDELAY` mitgegeben werden. Das sieht folgendermaßen aus:

```
file\_descr = open("/dev/ttyS0", O\_RDWR | O\_NDELAY | O\_NOCTTY);
/*                    Modus:    read      nicht     nicht              */
/*                              write     warten    controlling entity */
```

Sobald die Gerätedatei erfolgreich geöffnet ist, stellen Sie `O_NDELAY` sofort wieder ab, da sonst zukünftige `read()`-Kommandos nicht auf Daten warten, sondern immer sofort zurückkommen und damit ein lastintensives „busy waiting“ durchführen (`fcntl( filedescriptor, F_SETFL, O_RDWR );`). Eine Funktion zum Öffnen eines seriellen Ports könnte also folgendermaßen aussehen:

```
int open\_port(int port)
  {
  /*
   * Oeffnet seriellen Port
   * Gibt das Filehandle zurueck oder -1 bei Fehler
   * der Parameter port muss 0 .. 7 sein
   *
   * RS232-Parameter
   * - 19200 baud
   * - 8 bits/byte
   * - no parity
   * - no handshake
   * - 1 stop bit
   */
   int fd;
   struct termios options;
   switch (port)
     {
  /* nach Gegebenheit anpassen, ggf. Device als Parameter uebergeben */
     case 0: fd = open("/dev/ttyS0",   O\_RDWR | O\_NOCTTY | O\_NDELAY); break;
     case 1: fd = open("/dev/ttyS1",   O\_RDWR | O\_NOCTTY | O\_NDELAY); break;
     case 2: fd = open("/dev/ttyS2",   O\_RDWR | O\_NOCTTY | O\_NDELAY); break;
     case 3: fd = open("/dev/ttyS3",   O\_RDWR | O\_NOCTTY | O\_NDELAY); break;
     case 4: fd = open("/dev/ttyUSB0", O\_RDWR | O\_NOCTTY | O\_NDELAY); break;
     case 5: fd = open("/dev/ttyUSB1", O\_RDWR | O\_NOCTTY | O\_NDELAY); break;
     case 6: fd = open("/dev/ttyUSB2", O\_RDWR | O\_NOCTTY | O\_NDELAY); break;
     case 7: fd = open("/dev/ttyUSB3", O\_RDWR | O\_NOCTTY | O\_NDELAY); break;
     default: fd = -1;
     }
   if (fd >= 0)
     {
     /* get the current options */
     fcntl(fd, F\_SETFL, 0);
     if (tcgetattr(fd, &options) != 0) return(-1);
     bzero(&options, sizeof(options)); /* Structure loeschen, ggf vorher sichern
                                          und bei Programmende wieder restaurieren */
```

```
    cfsetspeed(&options, B19200);       /* setze 19200 bps */
    /* Alternativ:                                        */
     * cfsetispeed(&options, B19200);                     *
     * cfsetospeed(&options, B19200);                     */

    /* setze Optionen */
    options.c\_cflag &= ~PARENB;         /* kein Paritybit */
    options.c\_cflag &= ~CSTOPB;         /* 1 Stoppbit */
    options.c\_cflag &= ~CSIZE;          /* 8 Datenbits */
    options.c\_cflag |= CS8;
    options.c\_cflag |= (CLOCAL | CREAD);/* CD-Signal ignorieren */
    /* Kein Echo, keine Steuerzeichen, keine Interrupts */
    options.c\_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
    options.c\_oflag &= ~OPOST;          /* setze "raw" Input */
    options.c\_cc[VMIN]  = 0;            /* warten auf min. 0 Zeichen */
    options.c\_cc[VTIME] = 10;           /* Timeout 1 Sekunde */
    tcflush(fd,TCIOFLUSH);
    if (tcsetattr(fd, TCSAFLUSH, &options) != 0) return(-1);

    }
  return(fd);
  }
```

Benötigt werden meist die folgenden Headerdateien:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
#include <strings.h>
```

Bei Bedarf werden noch die drei Headerdateien

```
#include <signal.h>
#include <string.h>
#include <errno.h>
```

zusätzlich benötigt.

Nach dem Öffnen können Sie nach Herzenslust mit `read()` und `write()` seriell Daten empfangen und senden. Jedoch kann – wie schon angedeutet – immer mal etwas Ungewöhnliches passieren.

7.3 Daten senden und empfangen

7.3.1 Bytes senden

Für das Senden wird in der Regel die Funktion `write()` verwendet, deren erster Parameter der Filedeskriptor ist. Weitere Parameter sind die Adresse des Sendepuffers und die Anzahl der zu sendenden Bytes. Es muss auf jeden Fall überprüft werden, wieviele Bytes gesendet wurden (Rückgabewert von `write()`) und ob auch alle Bytes gesendet wurden.

```
int sendbytes(char * Buffer, int Count)
/* Sendet Count Bytes aus dem Puffer Buffer */
  {
  int sent;  /* return-Wert */
  /*  Daten senden */
  sent = write(fd, Buffer, Count);
  if (sent < 0)
    {
    perror("sendbytes failed - error!");
    return -1;
    }
  if (sent < Count)
    {
    perror("sendbytes failed - truncated!");
    }
  return sent;
  }
```

7.3.2 Bytes empfangen

Für das Empfangen wird in der Regel die Funktion `read()` verwendet, deren erster Parameter der Filedeskriptor ist. Weitere Parameter sind die Adresse des Sendepuffers und die maximale Anzahl der zu empfangenden Bytes. Die Funktion gibt die Anzahl der empfangenen Bytes zurück, wobei dieser Wert auch 0 sein kann. Das Verhalten von `read()` hängt von den Konfigurationswerten `c_cc[VTIME]` und `c_cc[VMIN]` ab. Bei der in `open_serial()` getroffenen Einstellung kehrt `read()` auf jeden Fall nach einer Sekunde zurück, ggf. ohne Zeichen empfangen zu haben. Dies ist bei der Programmierung zu berücksichtigen.

Das erste Programmfragment liest bis zu 100 Zeichen in einen Puffer:

```
char buf[101];   /* Eingabepuffer */
int anz;         /* gelesene Zeichen */
   ...

anz = read(fd, (void*)buf, 100);
if (anz < 0)
  perror("Read failed!");
else if (anz == 0)
   perror("No data!");
else
  {
  buf[anz] = '\0';      /* Stringterminator */
  printf("%i Bytes: %s", anz, buf);
  }
   ...
```

Das Verfahren eignet sich insbesondere dann, wenn Sie wissen, wieviele Bytes zu erwarten sind. Andernfalls gehen Sie vorsichtiger vor und lesen zeichenweise. Diese Methode eignet sich auch gut, wenn auf ein bestimmtes empfangenes Byte reagiert werden soll (Enter, Newline etc.):

```
char buf[101];   /* Eingabepuffer für die komplette Eingabe */
int anz;         /* gelesene Zeichen */
char c;          /* Eingabepuffer fuer 1 Byte */
int  i;          /* Zeichenposition bzw Index */
   ...

i = 0;
do               /* Lesen bis zum Carriage Return, max. 100 Bytes */
  {
  anz = read(fd, (void*)&c, 1);
  if (anz > 0)
    {
    if (c != '\r')
      buf[i++] = c;
    }
  }
while (c != '\r' && i < 100 && anz >= 0);

if (anz < 0)
  perror("Read failed!");
else if (i == 0)
   perror("No data!");
else
  {
  buf[i] = '\0';        /* Stringterminator */
  printf("%i Bytes: %s", i, buf);
  }
   ...
```

Sie sehen schon, das Empfangen wirft mehr Probleme auf, als das Senden. Hier muss immer eine speziell an die Kommunikation angepasste Lösung entwickelt werden. Das folgende Beispiel vermeidet das byteweise Lesen und füllt den Eingabepuffer, bis ein Zeilenende gesendet wurde (danach wartet die andere Station normalerweise, bis sie wieder etwas bekommt).

```
char buf[1000];    /* Eingabepuffer für die komplette Eingabe */
char *bufptr;      /* aktuelle Position in buf */
int  nbytes;       /* Number of bytes read */
int  tries;        /* Number of tries so far */
int anz;           /* gelesene Zeichen */
char c;            /* Eingabepuffer fuer 1 Byte */
int  i;            /* Zeichenposition bzw Index */
```

```
   ...

/* Bytes in den Puffer einlesen, bis ein CR or NL auftaucht */
bufptr = buf;
/* Achtung:               Etwas seltsame Pointer-Arithmetik */
while ((anz = read(fd, bufptr, buf + sizeof(buf) - bufptr - 1)) > 0)
  {
  if (anz < 0)
    {
    perror("Read failed!");
    return -1;
    }
  bufptr += anz;
  /* CR oder NL am Ende? */
  if ((bufptr[-1] == '\n') || (bufptr[-1] == '\r'))
    break; /* Schleife verlassen */
  }
/* Stringterminator anhaengen */
*bufptr = '\0';
printf("%s", buf);
   ...
```

Noch besser ist es, das Einlesen einer Zeile in eine Funktion zu verlagern. Die folgende Funktion `get_line()` liest von einer vorhergeöffneten Schnittstelle genau eine Zeile ein (der Unterstrich in `get_line()` ist notwendig, weil `getline()` eine Bibliotheksfunktion ist). Als Parameter werden neben dem Filedescriptor das Array und dessen maximale Länge übergeben:

```
int get_line(int fd, char *buffer, unsigned int len)
  {
  /* read a '\n' terminated line from fd into buffer
   * of size len. The line in the buffer is terminated
   * with '\0'. It returns -1 in case of error and -2 if the
   * capacity of the buffer is exceeded.
   * It returns 0 if EOF is encountered before reading '\n'.
  */
  int numbytes = 0;
  int ret;
  char buf;

  buf = '\0';
  while ((numbytes <= len) && (buf != '\n'))
    {
    ret = read(fd, &buf, 1);   /* read a single byte */
    if (ret == 0) break;       /* nothing more to read */
    if (ret < 0) return -1; /* error or disconnect */
    buffer[numbytes] = buf;    /* store byte */
    numbytes++;
    }
  if (buf != '\n') return -2;   /* numbytes > len */
  buffer[numbytes-1] = '\0';    /* overwrite '\n' */
  return numbytes;
  }
```

Nachteil dieser Lösung ist die Geschwindigkeit bzw. deren Fehlen. Durch die vielen `read()`-Aufrufe ist die Funktion ziemlich langsam. Besser wäre eine Lösung, bei der ein Datenpaket komplett eingelesen und dann Zeile für Zeile ans aufrufende Programm weitergereicht wird. Genau das macht die folgende Funktion, bei der die Parameter die gleiche Aufgabe haben wie oben. Diese Funktion hat einen internen Puffer, der mittels `read()` gefüllt wird und dessen Inhalt Stück für Stück bei jedem Aufruf weitergegeben wird. Dazu verwendet die Funktion die statischen Variablen `bufptr`, `count` und `mybuf`, deren Werte erhalten bleiben und bei jedem Aufruf wieder zur Verfügung stehen. Werden mit `read()` mehrere Zeilen gelesen, bleibt der jeweilige Rest in `mybuf` erhalten und wird beim nächsten Aufruf der Funktion verarbeitet:

```
int readline(int fd, char *buffer, unsigned int len)
  {
  /* read a '\n' terminated line from fd into buffer
   * bufptr of size len. The line in the buffer is terminated
   * with '\0'. It returns -1 in case of error or -2 if the
   * capacity of the buffer is exceeded.
   * It returns 0 if EOF is encountered before reading '\n'.
   * Notice also that this routine reads up to '\n' and overwrites
```

```
 * it with '\0'. Thus if the line is really terminated with
 * "\r\n", the '\r' will remain unchanged.
*/
static char *bufptr;
static int count = 0;
static char mybuf[1500];
char *bufx = buffer;
char c;

while (--len > 0)            /* repeat until end of line  */
  {                             /* or end of external buffer */
  count--;
  if (count <= 0)            /* internal buffer empty --> read data */
    {
    count = read(fd, mybuf, sizeof(mybuf));
    if (count < 0) return -1;/* error or disconnect */
    if (count == 0) return 0;   /* nothing to read - so reset */
    bufptr = mybuf;             /* internal buffer pointer    */
    }
  c = *bufptr++;                /* get c from internal buffer  */
  if (c == '\n')
    {
    *buffer = '\0';             /* terminate string and exit  */
    return buffer - bufx;
    }
  else
    {
    *buffer++ = c;              /* put c into  external buffer */
    }
  }
return -2;                      /* external buffer to short */
}
```

7.3.3 Timeout erkennen und behandeln

Was tun, wenn irgend etwas schiefgeht und die Zeichenkette, auf die das Programm wartet, nie kommt? Das Programm würde warten, bis der Anwender es manuell abbricht. Was aber bei Programmen, die im Hintergrund laufen, nicht sinnvoll ist. Die Lösung ist recht einfach: mit den Bibliotheksfunktionen `alarm()` und `signal()` installiert man einen „Alarmtimer“, der bei Bedarf einen Timeout erzeugt und der Routine sagt, dass die Wartezeit vorbei ist:

```
void alarm\_handler(void)
  { timeout = 1; }

signal(SIGALRM, alarm\_handler);
alarm(60); /* Wartezeit setzen */

...

if (read(file\_descr,buffer,1) != 1 && errno == EINTR && timeout)
  {
  fprintf(stderr,"TIMEOUT!\n"); break;
  }
...

alarm(0);  /* Alarm abschalten */
```

Mit diesen Informationen lassen sich nicht-interaktive Programme für die serielle Schnittstelle schreiben. Was noch fehlt, ist die Möglichkeit, mehrere Schnittstellen gleichzeitig zu überwachen, beispielsweise gleichzeitig Schnittstelle und Tastatur. Je nach Programm würde `read()` so lange warten, bis etwas an der Schnittstelle eintrifft, auch wenn Sie in der Zwischenzeit wie wild auf die Tastatur hämmern. Eine Möglichkeit wäre, über `O_NDELAY` zu arbeiten oder `VMIN` auf 0 zu setzen.

Das ist jedoch beides nicht effizient, weil das Programm „busy waiting“ mit voller CPU-Leistung angestrengt auf Daten wartet. Glücklicherweise gibt es einen Mechanismus, mit dem man warten kann, bis auf einem File-Deskriptor etwas zum Lesen bereitliegt, nämlich `select()`. Der Aufrufer übergibt der `select()`-Funktion eine Liste mit File-Deskriptoren, von denen gelesen oder auf die geschrieben werden soll. Sobald es möglich ist (Daten eingetroffen oder Ausgabewarteschlange frei), kehrt `select()` mit der entsprechenden Information zurück. Zusätzlich kann man einen Timeout definieren, der angibt, nach welcher Zeit die Funktion in jedem Fall aufgeben soll. Das folgende Beispiel ist dem Serial-Programming-HOWTO entnommen:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
main()
  {
  int fd1, fd2; /* input sources 1 and 2 */
  fd\_set readfs; /* file descriptor set */
  int maxfd; /* maximum file desciptor used */
  int loop=1; /* loop while TRUE */

  /* open\_input\_source opens a device, sets the port correctly, and
     returns a file descriptor */
  fd1 = open\_input\_source("/dev/ttyS0");
  if (fd1<0) exit(0);
  fd2 = open\_input\_source("/dev/ttyS1");
  if (fd2<0) exit(0);
  maxfd = MAX (fd1, fd2)+1; /* maximum bit entry (fd) to test */

  /* loop for input */
  while (loop)
    {
    FD\_SET(fd1, &readfs); /* set testing for source 1 */
    FD\_SET(fd2, &readfs); /* set testing for source 2 */
    /* block until input becomes available */
    select(maxfd, &readfs, NULL, NULL, NULL);

    if (FD\_ISSET(fd1)) /* input from source 1 available */
    handle\_input\_from\_source1();
    if (FD\_ISSET(fd2)) /* input from source 2 available */
    handle\_input\_from\_source2();
    }
  }
```

Statt der zweiten seriellen Schnittstelle könnten Sie auch mittels `FD_SET(STDIN, &readfs);` die Tastatur in `select()` einklinken und dann wahlweise Daten der Tastatur und der RS232 bearbeiten. Das Beispiel blockiert, bis Daten eintreffen. Will man nach einer bestimmten Zeit abbrechen, muss nur der `select()`-Aufruf geändert werden:

```
int res;
struct timeval Timeout;

/* set timeout value within input loop */
Timeout.tv\_usec = 0; /* milliseconds */
Timeout.tv\_sec = 1; /* seconds */
res = select(maxfd, &readfs, NULL, NULL, &Timeout);

if (res == 0)
  /* number of file descriptors with input = 0, timeout occurred. */
```

Bei diesem Beispiel gibt es nach einer Sekunde einen Timeout. In diesem Fall liefert `select()` 0 zurück.

7.4 Zugriff auf die Steuerleitungen

In manchen Fällen wollen Anwendungen einzelne Kontrollleitungen gezielt auf bestimmte Pegel setzen oder einzelne Pegel abfragen. Man kann so z. B. Tasten einlesen, ein Relais schalten oder die Signale eines Funkuhr-Empfängers detektieren (da reicht dann oft ein Transistor als Pegelwandler, und man kann auch noch auf den MAX232 verzichten).

Die Behandlung der seriellen Steuerleitungen hängt stark von der Rechnerarchitektur ab. Bei Systemen mit Intel-Architektur kann mit `ioctl`-Aufrufen auf die Steuerleitungen zugegriffen werden. Bei anderen Systemen werden oft auch allgemein verwendbare digitale Leitungen dafür verwendet. Für PCs ist die Behandlung unter der folgenden ULR beschrieben:
http://www.netzmafia.de/skripten/hardware/PC-Schnittstellen/seriell.html

8

Dokumentation

People often write less readable code because they think it will produce faster code. Unfortunately, in most cases, the code will not be faster.
– Felix von Leitner

Es gibt kaum eine größere Herausforderung als diejenige, einen wenig bis garnicht dokumentierten Code pflegen zu müssen. Die Dokumentation sagt einem nicht einfach nur, was eine bestimmte Funktion oder Variable tut, sondern vermittelt auch, wie und warum die Software in einer bestimmten Art und Weise implementiert wurde. Es werden bei der Implementierung eines Algorithmus einen Vielzahl von Entscheidungen getroffen und es kann entscheidend für die Pflege der Software sein, dass Sie von diesen Entscheidungsprozessen soviel wie möglich mitbekommen.

Viele Probleme mit der Dokumentation beruhen auf Zeitdruck, unsachgemäßem Software-Design und der Tatsache, dass die Beschreibung, wie etwas funktioniert einfach nicht so spannend ist, wie die Entwicklung selbst. Viele Entwickler hassen das Dokumentieren, aber die Dokumentation ist ein wesentlicher Teil der Entwicklung, nicht einfach so nebenher erstellt werden sollte. Es gibt einige Tricks, mit denen Sie sich das Leben in Bezug auf die Dokumentation erleichtern können.

Dokumentieren Sie sofort

Der Zeitdruck, ein Produkt ausliefern zu müssen führt oft zu einem Wild-West-Stil beim Codieren – Hauptsache, es funktioniert irgendwie und kann rausgehen. Während der hektischen Codierungsphase nimmt man sich wenig Zeit aufzuschreiben, was der Code eigentlich macht. Sobald das Produkt ausgeliefert ist, macht sich das Design-Team ans Dokumentieren. Das Problem dabei ist, dass Wochen oder sogar Monate vergangen sind, seit der Code geschrieben wurde! Es ist oft schon eine Herausforderung für einen Entwickler, sich daran zu erinnern, was er gestern zum Mittagessen hatte und er erinnert sich schon gar nicht, was ein Stück Code macht, das vor zwei Wochen auf seinem Bildschirm zu sehen war. Das Ergebnis ist ungenaue Dokumentation, später kommt es dann zu Missverständnissen und Fehlern.

Der Trick ist natürlich, gleich zu dokumentieren, was zum jeweiligen Algorithmus geführt hat und welche Randbedingungen berücksichtigt wurden. Ein formalisiertes Verfahren mit externer Dokumentation würde auf jeden Fall die Entwicklung hemmen, aber das Hinzufügen von Kommentaren in den Code nimmt kaum Zeit in Anspruch. Das erste, was ein Entwickler tun sollte, ist zu beschreiben, was der Code (z. B. bei einer Funktion) tut. Später kann der Kommentar dann erweitert und ergänzt werden. Bei jeder Änderung muss der Entwickler den Kommentar sofort aktualisieren. Die ständige Aktualisierung spart auf längere Sicht gesehen Zeit, da einerseits immer aktuelle Info vorliegt und bei einer späteren externen Dokumentation darauf zurückgegriffen werden kann.

Optimal ist es, vor dem Codieren einer Funktion schon den Kopfkommentar zu schreiben. Dann werden Sie sich nochmal klar darüber, was die zu programmierende Funktion eigentlich wirklich machen soll.

Dokumentation automatisieren

Trotz des kommentierten Codes wird oft noch eine externe Dokumentation benötigt, etwa wenn Bibliotheken nur in binärer Form weitergegeben werden sollen. Auch will sich nicht jeder durch den Quelltext arbeiten. Es existieren zahlreiche Tools zur Aufbereitung von Quelltext-Kommentaren. Ein sehr bekanntes Freeware-Tool ist Doxygen (www.doxygen.org). Beim Schreiben des Programms formatieren die Entwickler die Kommentare entsprechend und markieren so diejenigen Teile, die in eine externe Dokumentation übernommen werden sollen. Dann erzeugt Doxygen automatisch eine Dokumentation im HTML-, RTF- oder PDF-Format aus dem Quelltext. Das Bestechende dabei ist, dass die externe Dokumentation immer auf dem gleichen Stand ist wie Ihre Kommentare.

Kommentare richtig formulieren

Es ist immer gut, wenn Entwickler ihren Code ausführlich dokumentiert, aber oft stellt die Dokumentation nichts anderes dar als eine Wiederholung eines Variablen- oder Funktionsnamen bzw. eine verbale Beschreibung der jeweiligen Anweisung. Kommentare sollte aussagekräftig sein und zusätzliche Details über das Offensichtliche hinaus offenbaren. Geben Sie so viele Informationen wie möglich und vergessen Sie nicht, relevante Beschreibungen und auch Querbezüge zu nennen. Der Leser des Kommentars sollte in der Lage sein, daraus zu verstehen, wie sich die Software verhält und welche Vorbedingungen erwartet werden.

Erklären Sie das Warum

Der Mensch tendiert dazu, zu beschreiben, was eine Reihe von Befehlen macht, anstatt zu erklären, warum der Code das macht, was er macht. Ein beliebtes Beispiel ist das Isolieren eines Bits durch Schiebebefehle. Der Code und der Kommentar sehen in der Regel etwa so aus:

```
// Bit 6 x schieben und in PortB ausgeben
GpioPortB |= (1 << 6);
```

Der Kommentar ist eigentlich sinnlos. Jeder, der ein grundlegendes Verständnis von C besitzt, weiß aus Erfahrung, was die Codezeile tut. Viel eher stellt sich die Frage, eine Verschiebung um 6? Warum PortB? Ein Entwickler, der diese Codezeile später mal wieder liest, hat kaum eine Vorstellung, was diese Zeile wirklich bezwecken soll. Also wird Zeit damit verschwendet, dem nachzugehen. Etwas besser wäre die folgende Variante:

```
// PortB steuert das Motorrelais, das den Antriebsmotor aktiviert.
// Setzen von Bit 6 schaltet den Motor ein, Loeschen des Bits schaltet ihn aus.
GpioPortB | = (1 << 6);
```

Dieser Kommentar ist nicht perfekt, aber er erklärt, warum der Entwickler Bit 6 von PortB setzt. Noch besser wäre es gewesen, die „6“ durch eine „sprechende“ Konstante zu ersetzen (Stichwort: „magic numbers“):

```
// PortB steuert das Motorrelais, das den Antriebsmotor aktiviert.
// Setzen von Bit (MOTOR) schaltet den Motor ein, Loeschen des Bits schaltet ihn aus.
GpioPortB | = (1 << MOTOR);
```

Beispiele verwenden

Es kann sehr hilfreich sein, wenn Funktionen oder Variablen Kommentare Beispiele enthalten, die zeigen, wie diese verwendet werden soll. So ein Beispiel kann oft die Anwendung einer Funktion oder Methode viel klarerer zeigen, als eine lange Beschreibung. Auch verhindert es manchmal, dass der Code für einen falschen Zweck eingesetzt wird. Etwa folgendermaßen:

```
/*************************************************************************
* Function: PWM_SetDutyCycle( )
*/

/**
* \ section Description Beschreibung:
*
* Diese Funktion setzt das Tastverhaeltnis eines PWM-Ausgangs.
* Das Tastverhaeltnis und die Frequenz des Signals haengen voneinander ab.
* Die Funktion berechnet Timing und Frequenz so, dass sich das gewuenschte
* Tastverhaeltnis ergibt.
*
* \ param uint16 DutyCycle - Tastverhaltnis 0-1000 = 0-100%
*               uint8  Port      - PWM-Port
*
* \ return      None
*
* \ section Example Beispiele:
```

```
* \ code
*    // Setze Tastverh. 50% fuer Port 1
*    PWM_ SetDutyCycle(500, PWM1);
*
*    // Setze Tastverh. 65,3% fuer Port 2
*    PWM_ SetDutyCycle(653, PWM2);
* \endcode
*
* \see PWM_Init
* \see PWM_Disable
* \see PWM_Enable
*
*****************************************************************************/
void PWM_SetDutyCycle(uint 16 DutyCycle, uint8 Port)
  {
   ...
  }
```

Verwenden Sie einen Dokumentationsstandard

Genau wie beim Schreiben von Code sollte es einen Standard für die Kommentare geben. Nehmen Sie diese Punkte (es werden nicht besonders viele sein) gleich mit in den Codierungsstandard auf. Damit wird sichergestellt, dass jeder im Team auf die gleiche Weise kommentiert und dokumentiert.

Der einfachste Weg, den Dokumentationsstandard sicherzustellen, ist eine Vorlage für Kopfkommentar, Rumpf-Quelle und Dokumentation zu erstellen. Wenn ein neues Modul erstellt wird, nimmt man die Vorlage und fügt dann die relevante Information hinzu. Ein weiterer Vorteil an diesem Ansatz ist, dass er auch noch eine Menge Zeit spart.

Forward- und Back-Annotation

Vor der Implementierungsphase eines Projekts steht die Software-Design-Phase. Hier werden viele hübsche Dokumente und Diagramme produziert (Flussdiagramme, Zustandsdiagramme, Klassendiagramme usw.), die während der eigentlichen Implementierung benötigt werden. Während der Entwicklung und dem Test ergeben sich fast immer Abweichungen. Leider finden diese Veränderungen nur selten ihren Weg zurück in die Design-Dokumente. Deshalb aktualisieren Sie während der Implementierungs- und Testphase auch diese Dokumente auf der Stelle. Zu einem späteren Zeitpunkt wird das nie mehr geschehen.

Mit dem Einsatz von automatisierten Tools ist es möglich, Coding-Standards, Abkürzungen, Projektdetails, Anforderungen und eine Vielzahl von anderen Elementen in die integrierte Dokumentation aufzunehmen. Packen Sie auch die Dokumentation der Design-Phase mit in das Dokumentationspaket für den Code. Nur so haben Sie auch später noch alles beisammen.

Richtlinien für Quellcode-Kommentare verwenden

So seltsam es klingt, es gab schon Religionskriege über die Art und Weise der Kommentierung. Ebenso wird immer noch darüber diskutiert, ob die öffnende geschweifte Klammer bei C in einer neuen Zeile stehen soo oder nicht. Egal, welchem Guru Sie auch folgen, es läuft alles auf Konsistenz hinaus. Wenn das Team beschlossen hat, nur `/* ... */` zu verwenden, dann verwenden Sie nur diesen Stil. Wenn dagegen Kommentierung mittels `//` vorzunehmen, dann machen Sie das auch. Auch eine Mischung ist möglich, `/* ... */` für den Kopf von Funktionen und Methoden und `//` für einzeilige Kommentare. Was auch immer präferiert wird, halten Sie sich daran.

Beim Code wird sehr darauf geachtet, dass er strukturiert und leicht zu lesen ist (so ist Einrückung Pflicht usw.). Kommentare sind nicht anders zu behandeln. Nur manchmal strukturierte Kommentare machen es schwer, den Kommentar zu finden und zu erfassen. Kommentare sollten so formatiert sein, das auch beim Ausdrucken des Codes ein lesbares Dokument entsteht (also z. B. keine überlangen Zeilen).

Kommentieren Sie nicht jede Zeile

Mal ehrlich: Entwickler haben nicht wirklich Lust, ihre Software zu kommentieren. Es kostet Zeit und ist langweilig. Alles andere macht mehr Spaß (außer natürlich, in einer Sitzung hocken). Als gut dokumentiert wird oft der Code angesehen, bei dem jede einzelne Codezeile kommentiert ist. Dies kann aber zu viel sein. Der Zweck des Kommentars ist, zukünftigen Entwicklern das Wie und das Warum der Software nahezubringen. Eine ausführliche Abhandlung ist weder erforderlich noch gewollt. Schreiben Sie das Wichtigste in den Kopfkommentar einer Funktion – was oft völlig

ausreichend ist. Sogar bei kleinen Änderungen im Code behält dieser Kommentar seinen Sinn. Kommentieren Sie dann nur die Zeilen, bei denen der Code etwas „tricky“ ist.

Anhang

A.1 Literatur

A.1.1 Schnittstellen

- Jan Axelson: *Serial Port Complete*, Lakeview Research
- Jan Axelson: *USB Complete*, Lakeview Research
- Jürgen Hulzebosch: *USB in der Elektronik*, Franzis
- H.-J. Kelm (Hrsg.): *USB 2.0*, Franzis
- S. Furchtbar, W. Hackländer: *I^2C-Bus angewandt*, Elektor

A.1.2 Schaltungstechnik

- Dieter Zastrow: *Elektronik*, Vieweg-Verlag
- G. Koß, W. Reinhold, F. Hoppe: *Lehr- und Übungsbuch Elektronik*, Fachbuchverlag Leipzig
- U. Tietze, Ch. Schenk: *Halbleiter-Schaltungstechnik*, Springer-Verlag
- Helmut Lindner: *Taschenbuch der Elektrotechnik und Elektronik*, Hanser
- E. Prohaska: *Digitaltechnik für Ingenieure*, Oldenbourg
- Ch. Siemers, A. Sikora: *Taschenbuch der Digitaltechnik*, Hanser
- Don Lancaster: *Das CMOS-Kochbuch*, VMI Buch AG
- Don Lancaster: *TTL-Cookbook*, Sams Publishing
- Hans-Dieter Stölting, Eberhard Kallenbach: *Handbuch Elektrische Kleinantriebe*, Hanser
- Elmar Schrüfer: *Elektrische Messtechnik*, Hanser
- *Zeitschrift Elektor*, Elektor-Verlag, Aachen
- *Elrad-Archiv 1977–1997 DVD*, eMedia GmbH, Hannover

A.2 Links

PC-Schnitstellen: *http://www.netzmafia.de/skripten/hardware/PC-Schnittstellen/*

Serielle Schnitstelle: *http://www.netzmafia.de/skripten/hardware/PC-Schnittstellen/seriell.html*

Raspberry Pi: *http://www.netzmafia.de/skripten/hardware/RasPi/*

Das Elektronik-Kompendium: *http://www.elektronik-kompendium.de/*

Kabel- und Stecker-FAQ: *http://www.kabelfaq.de/*

Maxim-Datenblätter: *http://www.maxim-ic.com* und *http://datasheets.maxim-ic.com*

Datenblätter aller Art: *http://www.datasheets.org.uk/* und *http://www.alldatasheet.com*

Einführung in SPS: *http://www.studet.fh-muenster.de/~diefrie/einfh.html*

Stichwortverzeichnis