

Embedded Programmierung

Methoden und Verfahren

Jürgen Plate, 5. September 2014

Inhaltsverzeichnis

1	Besonderheiten der Embedded-Programmierung	5
1.1	Grundprinzipien	5
2	Variablen	7
2.1	Speicherklassen	8
3	Bitmanipulation und -verknüpfung	11
3.1	Bitmanipulation	11
3.2	Bitverknüpfungen	12
3.3	Schiebeoperationen	12
3.4	Nützliche Makrodefinitionen	14
4	Programmierprinzipien	15
4.1	Mapping-Funktion	15
4.2	Magic Numbers	15
4.3	Enumeration	16
4.4	Variante Strukturen (union)	18
5	Typische Algorithmen	21
5.1	Statemaschine	21
5.1.1	Implementierung mit einer switch – case-Anweisung	21
5.1.2	Implementierung mit einer Zustandstabelle	23
5.2	FIFO-Speicher	27
6	Ein- und Ausgabe	29
6.1	Software-Entprellung	29
6.2	Über den Tellerrand blicken	31
6.3	LED als Ausgabegerät	34
6.4	Beschleunigungssensoren auswerten	38
6.5	Dokumentation	41
Anhang		45
A.1	Literatur	45
A.1.1	Schnittstellen	45
A.1.2	Schaltungstechnik	45
A.2	Links	45
A.3	Bezugsquellen	46
	Stichwortverzeichnis	49

1

Besonderheiten der Embedded-Programmierung

1.1 Grundprinzipien

In diesem Skript soll auf einige Besonderheiten der Programmierung eingegangen werden – nicht nur bei Linux, sondern bei Mikrocontrollern ganz allgemein. Dies sind einerseits die bei Controllern geringeren Ressourcen und andererseits Zeitbedingungen beim Programmablauf. Hinzu kommen oft noch Besonderheiten von Compiler und Bibliotheken.

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,
 - boolesche 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`).

2

Variablen

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.

2.1 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 Quellencoddatei 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.

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

¹In 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 $X_{i+1} = (a \cdot X_i + c) \bmod p$. 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 Schiebefehle 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 Siebensegment-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 notwendig 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 Enumerations-typ 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 Gültigkeitsbereich 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 Ausgaberroutine fuer die einzelnen 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 weiteres 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;
}
```

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 implementieren:

- 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 Statemaschine 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 Statemaschine ist verboten. Wenn die Statemaschine 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:

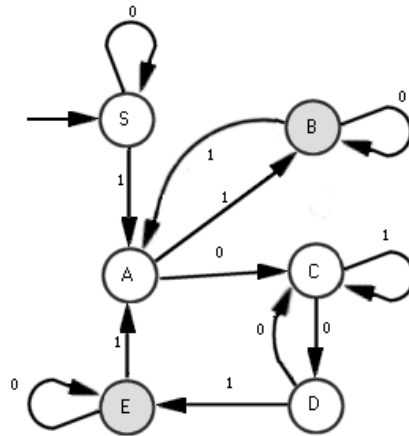


Bild 5.1: Zustandsfolgen des Beispielautomaten (B und E sind Endzustände)

```

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

/*Definition des Automaten */
#define N_ZUST 6 /* Anzahl Zustaeende */
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:
                    if (in == 0) state = E;
                    else state = A;
                    break;
            }
        }
        input++;
    }

    return state;
}

```

```

        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	ja
A	C	B	
B	B	A	
C	D	C	
D	C	E	ja
E	E	A	

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>

/*Definition des Automaten */
#define N_ZUST 6 /* Anzahl Zustände */
#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))

/* Statemaschine 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>

/*Definition des Automaten */
#define N_ZUST 6 /* Anzahl Zustände */
#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

```

5.2 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).

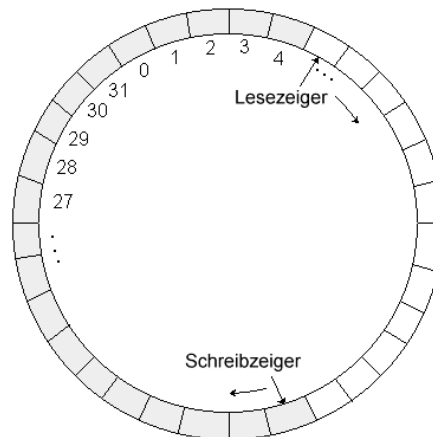


Bild 5.3: Schema des Ringpuffers

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 Objekt-orientierung:

```
#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 Tastendrucke 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 Pellen zu detektieren. Dadurch kann eine kürzere Wartezeit (mindestens $8 \cdot 1 \text{ ms} = 8 \text{ ms}$) erreicht werden. Die Funktion testet 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 21) 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 */
}
...
```

6.2 Über den Tellerrand blicken

12 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 Tastenmatrix werden zwei Widerstandsnetzwerke angelötet, so dass die Tastatur nur noch über drei Leitungen mit dem Controller verbunden ist: Vcc, GND und ein Analogausgang. Der Kondensator am Ausgang soll Störungen unterdrücken.

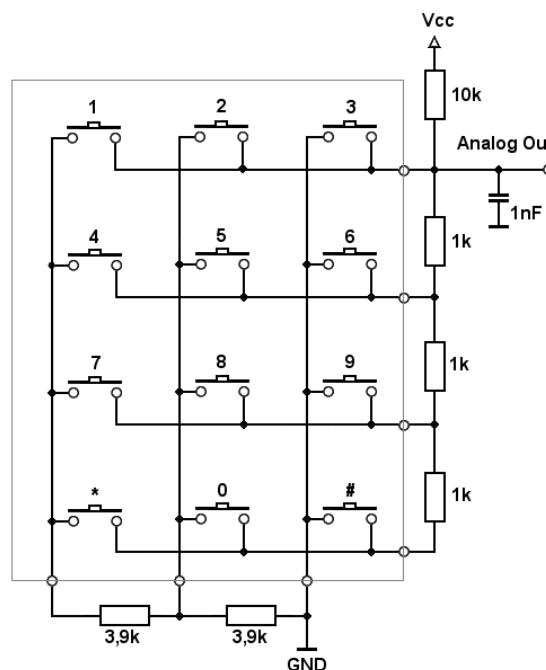


Bild 6.1: Schaltung der Tastatur

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 ($V_{cc} = 5\text{ V}$). 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 V_{cc} 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 gedrückt?
  {
    digitalWrite(LED, HIGH);    // Tastensignal
    if (key != oldkey)          // neue Taste gedrückt?
    {
      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-Farbtintensitäten 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 einer Schleife bis zum gewünschten Wert vorarbeiten müsste.

Wenn sie die LEDs bei den beiden vorangegangenen Beispielen genau beobachten, stellen Sie fest, dass die Farbübergänge nicht gleichmäßig sind. Das liegt 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,
  // logarithmisch 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 Mi-

nustemperaturen bis zum warmen Rot bei sommerlicher Hitze. Dann wäre es viel praktischer, wenn man im Programm nur einen einzigen 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 umrechnen
    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 rechnet einen HSV-Wert in die entsprechenden
    // RGB-Werte um. Diese werden im Array 'colors' zurückgegeben
    // colors[0] = Rot, 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 $\pm 180^\circ$ 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.

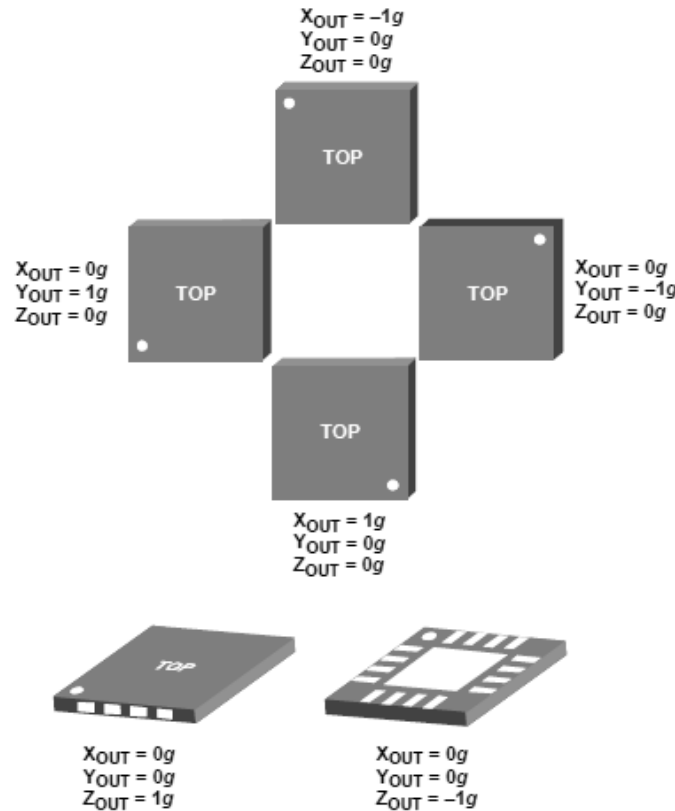


Bild 6.2: Ausgabe des Sensors je nach Lage im Raum

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

Bei Messungen auf der Erde können prinzipiell immer nur zwei Achsen berücksichtigt 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 ADC1, ADC2 und ADC3 gelieferten Digitalwerte gilt wegen der (typischerweise vorhandenen) ADC-Auflösung von 10 Bit:

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

mit $U_{ref} = 3,3V$. Ein Wert von $U = U_{ref}/2 = 1,65V$ entspricht dabei laut Datenblatt einer Beschleunigung von $0g$. 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.

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 Unterla-

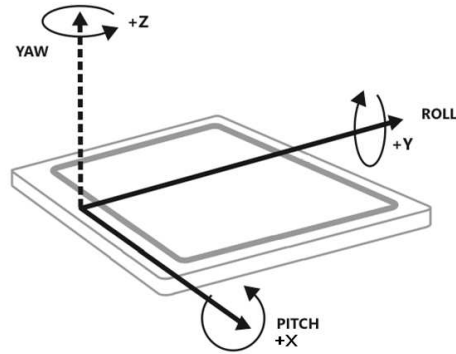


Bild 6.3: Yaw, Pitch und Roll bezeichnen die Z-, X- und Y-Achse

ge 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:

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

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):

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

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

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

pitch: Winkel der Drehung um die Längsachse, waagrecht ist pitch = 0, positive Werte ergeben sich bei einer Drehung im Uhrzeigersinn.

roll: Winkel beim Kippen nach vorn bzw. hinten, waagrecht ist roll = 0, positive Werte ergeben sich bei einem Kippen nach vorn.

yaw: Winkel der Abweichung von der Waagerechten, waagrecht ist yaw = 0, 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 Dreiachsens-Beschleunigungsmesser anpassen.

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:

```
// 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);
}

```

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

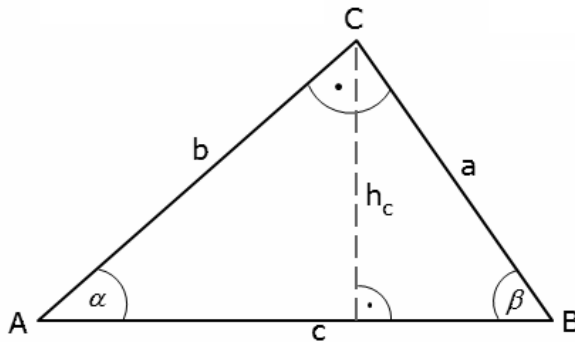
Übrigens basiert die ganze Mathematik auf der Dreiecksberechnung (Trigonometrie, Schulmathema-
 tik). Das Bild 6.4 fasst alle Gleichungen zusammen.

6.5 Dokumentation

People often write less readable code because they think it will produce faster code. Unfor-
 tunately, 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 Funk-
 tion oder Variable tut, sondern vermittelt auch, wie und warum die Software in einer bestimmten Art

rechtwinkliges Dreieck**allgemeines Dreieck (Sinussatz)**

$$\frac{a}{\sin \alpha} = \frac{b}{\sin \beta} = \frac{c}{\sin \gamma} = 2r = \frac{abc}{2F}$$

Umfang
 $u = a + b + c$

Fläche
 $A_{\text{Rechtw. inlig}} = \frac{1}{2} a \cdot b$

$$A_{\text{Allgemein}} = \frac{1}{2} a \cdot h_a = \frac{1}{2} b \cdot h_b = \frac{1}{2} c \cdot h_c$$

Im rechtwinkligen Dreieck gelten folgende trigonometrische Gleichungen:

$$\sin \alpha = \frac{\text{Gegenkathete von } \alpha}{\text{Hypotenuse}} = \frac{a}{c}$$

$$\cos \alpha = \frac{\text{Ankathete von } \alpha}{\text{Hypotenuse}} = \frac{b}{c}$$

$$\tan \alpha = \frac{\text{Gegenkathete von } \alpha}{\text{Ankathete von } \alpha} = \frac{a}{b}$$

Satz des Pythagoras:

$$c^2 = a^2 + b^2$$

Bild 6.4: Die zugrundeliegenden Trigonometrie-Formeln

und Weise implementiert wurde. Es werden bei der Implementierung eines Algorithmus eine 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.

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.

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 klarer 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 gewünschte
 * Tastverhaeltnis ergibt.
 *
 * \ param uint16 DutyCycle - Tastverhaeltnis 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 Kopfkomentar, 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 soll 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, dass auch beim Ausdrucken des Codes ein lesbares Dokument entsteht (also z. B. keine überlangen Zeilen).

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: *I²C-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

Das Elektronik-Kompodium: <http://www.elektronik-kompodium.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>

A.3 Bezugsquellen

Platinenfertigung (PCB-Pool, Beta-Layout)	http://www.pcb-pool.com
Frontplattenfertigung (auch Muster)	http://www.wk-mechanik.de
Conrad Electronic (Elektronikbauteile, Sensoren, Module usw.)	http://www.conrad.de
Reichelt Elektronik (Elektronikbauteile, Sensoren usw.)	http://www.reichelt.de
Farnell in one (Elektronikbauteile aller Art)	http://www.farnellinone.de
ELV Elektronik AG (Elektronikbauteile, Module usw.)	http://www.elv.de
Boards, Bausätze und Bauteile für den Robotik-, Microcontroller-Bereich	http://www.robotikhardware.de
Robotik-Shop (Funkmodule, Ultraschall, Kameras, Sensoren usw.)	http://www.roboter-teile.de
Robot Electronics (UK)	http://www.robot-electronics.co.uk
Lumitronix (LEDs, LES-Strips usw.)	http://www.leds.de/
LC-Displays	http://www.electronic-assembly.de http://www.lcd-store.de http://www.data-modul.com/de/
LDC-Steuerplatinen, seriell, I ² C	http://www.elabinc.com http://www.channaa.com
Holtec Encoder und Decoder	http://www.holtek.com
Funkmodule	http://www.rfsolutions.co.uk http://www.vadgmbh.de
Servo-Steuerbausteine	http://www.elabinc.com http://www.sytec-net.de www.micronics.com http://www.techdesign.be
USB-Chips und -Module	http://www.ftdichip.com http://www.codemercs.com
Profi-Mess GmbH (Sensoren aller Art)	http://www.profimess.de
Pepperl + Fuchs (Sensoren aller Art)	http://www.pepperl-fuchs.com
Feuchte- und Temperatursensoren	http://www.hygrosens.de http://www.humirel.com http://www.sensirion.com
Devantech (Kompass-Sensoren und anderes)	http://www.robot-electronics.co.uk
Ultraschall-Sensormodule	http://www.robotikhardware.de http://www.robot-electronics.co.uk
Gas-Sensoren	http://www.figarosensor.com http://www.alphasense.com
Geiger-Müller-Zählrohre	http://www.schuricht.de http://www.graetz.com
Farbsensoren	http://www.avago.com http://www.MAZeT.de

Stichwortverzeichnis

Symbole

7-Segment-Codierung 15

A

Accelerometer 38
ADXL335 38
Analogschnittstelle 31
Aufzählungstyp 16
Automat 21

B

Beschleunigungssensor 38
Besonderheiten 5
Bitmanipulation 11
Bitmaske 11
Bitoperationen, Makros 14
Bitoperatoren 11
Bitverknüpfungen 12
Busy Wait 6

D

Dokumentation 41
Dreiecksberechnung 38

E

Entprellung 29
enum 16
Enumeration 16
Exor-Verknüpfung 12

F

FIFO-Speicher 27
First-In-First-Out 27

H

HSV-Farbraum 34

I

Interrupts 5

K

keypressed 29

L

LED Fading 34

LED-Ansteuerung 34

M

Magic Numbers 15, 17
Mapping-Funktion 15
Mikrocontroller 5

N

Negation 12

O

Oder-Verknüpfung 12

P

Polling 5

R

Ressourcen, Mikrocontroller 5
RGB-Farbraum 34
RGB-LED 34
Ringpuffer 27

S

Schiebeoperationen 12
Shift 12
Siebensegment-Codierung 15
Software-Entprellung 29
Statemaschine 21

T

Tastatur 31
Trigonometrie 38

U

Und-Verknüpfung 12

W

waitkey 30
Warteschleife 29

Z

Zustandsautomat 21
Zustandstabelle 23
Zustandsvariable 29