Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Technik & Architektur

Mikrocontroller, C-Programmierung

Selbststudium und Übungen 3

Martin Vogel, Peter Sollberger V7.1

Sie kennen die Datentypen in C und deren Verwendung und Zugriffe; nicht-elementare Datentypen Strukturen, Aufzählungstypen, Union und Bitfelder.

1. Selbststudium

Gehen Sie nochmals die Folien der Vorlesung durch.

2. Übungen und Aufgaben

(zu diesen Übungen werden durch die Studierenden Musterlösungen präsentiert und abgegeben).

1. **Liste mit Namen/Vornamen**
   Schreiben Sie ein Programm, welches in einer einfach verketteten Liste eine beliebige Anzahl von Namen/Vornamen festhält. Allokieren Sie den Speicherplatz für Ihre Daten dynamisch. Schreiben Sie ein Testprogramm, wo Sie diese Liste einsetzen, z.B. indem Sie über die Konsole Namen/Vornamen eingeben und anschliessend die Liste ausgeben.

2. **Entwurf einer Übung zum Thema Union**
   Entwerfen Sie eine Übung analog der Aufgabe in Kapitel 3 wo Sie Unions einsetzen. Schreiben Sie eine Aufgabenstellung und erarbeiten Sie die Musterlösung.

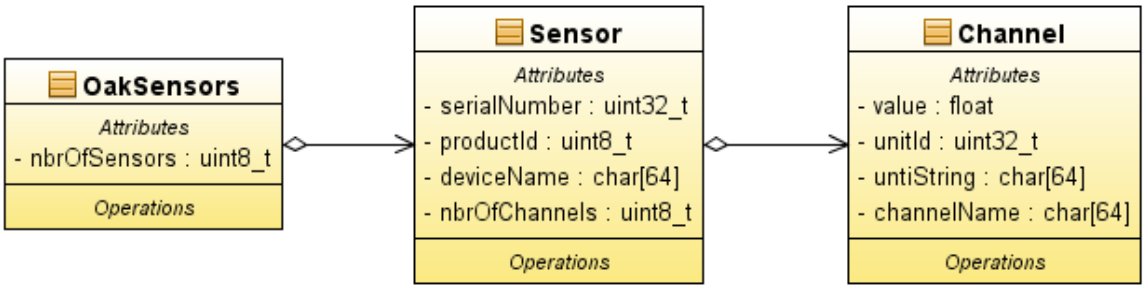
3. **Wörter Zählen**
   Diese Übungen hier lehnen an die Übungen im Leitprogramm (Binärbäume) aus Programmieren 2 an. Entsprechend finden Sie dort zusätzliche hilfreiche Informationen dazu.

   a) Binärbaum erzeugen
   Schreiben Sie ein Programm, welches die Häufigkeit von Wörtern in einem Textfile zählt. Verwenden Sie dabei die Datenstruktur eines Binärbaums. Der Knoten soll dabei einen Zeiger auf ein Wort und die Häufigkeit des Wortes festhalten. Sehen Sie dazu auch das Beispiel im Buch K&R Seite 134 ff (im Kapitel 4 hier beigefügt)
   Es soll eine Funktion programmiert werden, welche ein File ausliest und einen Binärbaum entsprechend aufbaut. Verwenden Sie dazu die im Anhang vorgegebenen Funktionen getword, getch, ungetch.

   b) Ausgabe in Postorder
   Programmieren Sie eine Funktion, welche in postorder vom Baum die Schlüssel-Worte und deren Häufigkeit ausgibt z.B. mittels printf() in der Funktion treeprint (siehe Anhang).

4. Structure Tree

Erstellen Sie eine baumartige, dynamische Datenstruktur zum Verwalten von Sensoren und dessen Messwerte. Dynamisch bedeutet hier, dass die Anzahl Sensoren und Anzahl Kanäle pro Sensor erst zur Laufzeit bekannt sind.
Die Sensoren[1] und deren Daten sind in folgendem UML Diagramm ersichtlich.



Schreiben Sie nun zwei Methoden, die diese Strukturen nutzen:

… `scanSensors`:

Erstellt die Datenstruktur. Simulieren Sie den Scan Vorgang, indem sie z.B. folgende Daten verwenden:

5 Sensors

- Atmospheric preasure (SN 111111, PID 1) has 2 channels
  - Pressure measures Pa (23)
  - Temperature measures K (37)
- Humidity (SN 22222, PID 2) has 2 channels
  - Temperatur measures K (37)
  - Humidity measures % relative (49)
- Acceleration (SN 33333, PID 3) has 3 channels
  - x measures m/s^2 (55)
  - y measures m/s^2 (55)
  - z measures m/s^2 (55)
- Current (SN 44444, PID 4) has 1 channels
  - Current (4..20 mA) measures A (13)
- Luminosity (SN 55555, PID 5) has 1 channels
  - Illuminance measures Lux (87)

`int readValues(…)`:

In die übergebene Datenstruktur wird pro Sensor und pro Kanal der aktuelle Messwert eingetragen. Simulieren Sie auch diesen Vorgang z.B. mit folgenden Daten:

Measurements:

- Atmospheric preasure:
  - Pressure: 101539.000000 Pa
  - Temperature: 297.700012 K
- Humidity:
  - Temperatur: 297.299988 K
  - Humidity: 58.700001 % relative
- Acceleration:
  - x: 0.010000 m/s^2
  - y: -0.020000 m/s^2
  - z: 9.810000 m/s^2
- Current:
  - Current (4..20 mA): 0.007500 A
- Luminosity:
  - Illuminance: 12746.000000 Lux

Testen Sie diese Funktionen mit einer main Funktion, die die entsprechendn Ausgaben erzeugt.

[1] Dieses Beispiel lehnt sich an die Oak Sensorfamilie der Firma Toradex (www.toradex.com) an.

Anhang zu Aufgabe 3 – K&R Seite 134, 135, 131 und Seite 77.
Datei `main.c: Wörter zählen`

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define MAXWORD 100
#define BUFSIZE 100

struct tnode{
          char *word;
          int count;
          struct tnode *left;
          struct tnode *right;
};
/* siehe auch Buch K&R S. 77 */
char buf[BUFSIZE];    /* Buffer fuer ungetch() */
int bufp = 0;           /* naechste freie Position für buf */

int getch(void){   /* naechstes Zeichen holen */
   return (bufp > 0 ? buf[--bufp] : getchar());
}

void ungetch(int c){
  if (bufp >= BUFSIZE)
      printf("ungetch: too many characters\n");
  else
      buf[bufp++] = c;
}

/* siehe auch Buch K&R S. 131 */
int getword(char *word, int lim){
   int c;
   char *w = word;
   while(isspace(c = getch()))
          ;
   if (c != EOF)
          *w++ = c;
   if (!isalpha(c)) {
          *w = '\0';
          return(c);
   }
   for( ; --lim > 0; w++)
          if(!isalnum(*w = getch())){
             ungetch(*w);
             break;
          }
   *w ='\0';
   return word[0];
}

struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);

 /* Haeufigkeit von Woerter zaehlen */
main(){
struct tnode *root;
char word[MAXWORD];

root = NULL;
printf("File wird eingelesen...\n");

 if (freopen("inputfile1.txt", "r", stdin) == NULL){
        printf("kann File nicht öffnen\n");
    }

   while(getword(word, MAXWORD) != EOF)
          if(isalha(word[0]))
                    root = addtree(root, word);
   treeprint(root);
   return(0);
}
```

3. Weitere freiwillige Aufgabe

Umwandlung byte[] von/nach float

Sie haben vier Bytes (z.B. via IIC aus einem Flash ausgelesen), welche Sie als float Wert gemäss IEEE 754 interpretieren müssen, resp. umgekehrt.
Schreiben Sie Funktionen, welche diese „Umrechnungen“ mit Hilfe einer Union durchführen.
Zum Kreieren von Testdaten können Sie zum Beispiel den Rechner von http://www.h-schmidt.net/FloatApplet/IEEE754de.html verwenden.

Musterlösung (HCS08):

```
/*
 * Einsatz von Unions
 * Author: Martin Vogel, Dozent HSLU Luzern
*/

#include <hidef.h> /* for EnableInterrupts macro */
#include "platform.h" /* include peripheral declarations */
#include "clock.h"
#include "sci.h"
#include <stdio.h>
#include <stdlib.h>
#include <termio.h>
#include <libdefs.h>
#include <string.h>
#include <float.h>

#define MAXLENGTH 100

typedef union Protocoll_{
        long iVal;
        char cVal[4];
        float fVal;
}Protocoll_t;

enum uFormat{
        INTEGER, CHAR, FLOAT
};

void sendProto(enum uFormat myFormat, Protocoll_t theVal) {

        switch (myFormat){
                case(INTEGER):
                        (void)printf("integer: %d\n",theVal.iVal);
                        break;
                case(CHAR):
                        (void)printf("Char: %c ",theVal.cVal[0]);
                        (void)printf("%c ",theVal.cVal[1]);
                        (void)printf("%c ",theVal.cVal[2]);
                        (void)printf("%c \n",theVal.cVal[3]);
                        break;
                case(FLOAT):
                        (void)printf("float: %f",theVal.fVal);
                        break;
                default:
                        printf("ERROR");
                        break;
        }
}


main() {
    int  len = 0;               // Anzahl eingelesene Zeichen
    char line[MAXLENGTH];       // Zeichenkette einer Zeile
    char s[30];                                 // read input from scanf

    char* str = &s;
    char c;
```

```
    Protocoll_t myVal;
    myVal.fVal = 0.0f;

    initClock();
    sci2Init(9600);
    EnableInterrupts;

    (void)printf("Hello Terminal!\n");
    (void)printf("I am the MC car - Press ENTER to start\n");
    (void)printf("=====================================\n");

    (void)scanf("%s",str);

    do {
            (void)printf("\n");
            (void)printf("S --> store a float value\n");
            (void)printf("I --> Send Integer\n");
            (void)printf("C --> Send Char\n");
            (void)printf("F --> Send Float\n");
            (void)printf("A --> Send All\n");
            (void)printf("Q --> Quit\n");
            (void)printf("\n");

            while (!isalnum(c = getchar()));
            // Solange einlesen bis Zahl oder Buchstabe
            c = toupper(c);
            while(getchar() != '\n');
            //read with getchar till stream reaches the \n end

            switch (c) {
                case 'S':
                    (void)printf("Enter a float number: \n");
                    (void)scanf("%f", &myVal);
                    break;
                case 'I':
                    sendProto(INTEGER, myVal);
                    break;
                case 'C':
                    sendProto(CHAR, myVal);
                    break;
                case 'F':
                    sendProto(FLOAT, myVal);
                    break;
                case 'A':
                    sendProto(INTEGER, myVal);
                    sendProto(CHAR, myVal);
                    sendProto(FLOAT, myVal);
                    break;
                default:
                    break;
            }
        } while (c != 'Q');

   (void)printf("program stopped");
   for(;;){ }
   return(TRUE);
}
```

4. Rekursive Strukturen (K&R Seite 134 ff - Kapitel 6.5)

Nehmen wir an, daß wir das allgemeinere Problem lösen wollen, die Häufigkeit *aller* Wörter in einem Text zu zählen. Da die Liste der Wörter von vornherein nicht bekannt ist, können wir sie nicht einfach sortieren und binäre Suche verwenden. Andrerseits können wir auch nicht jedes eingegebene Wort linear suchen, um zu sehen, ob das

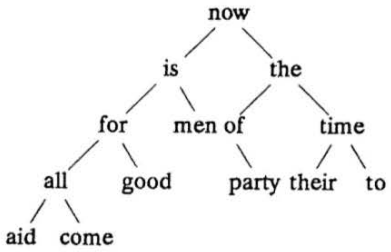
Eine mögliche Lösung ist, die bereits eingegebenen Wörter grundsätzlich sortiert aufzubewahren, indem man jedes Wort unmittelbar nach der Eingabe an die richtige Stelle speichert. Dazu sollte man allerdings nicht Wörter in einem linearen Vektor verschieben – auch dies benötigt zuviel Zeit. Wir werden statt dessen eine Datenstruktur verwenden, die man einen *binären Baum* nennt.

Der Baum enthält einen „Knoten" für jedes unterschiedliche Wort; ein Knoten enthält folgende Information:

einen Zeiger auf den Text des Worts
einen Zähler für die Häufigkeit
einen Zeiger auf den linken Nachkommen
einen Zeiger auf den rechten Nachkommen

Ein Knoten kann dabei nicht mehr als zwei Nachkommen besitzen; ein oder gar kein Nachkomme kann allerdings auftreten.

Die Knoten werden so angeordnet, daß bei jedem Knoten der linke Unterbaum nur Wörter enthält, die alphabetisch vor dem Wort im Knoten stehen, und der rechte Unterbaum nur die „größeren" Wörter. Für den Satz „now is the time for all good men to come to the aid of their party" entsteht folgender Baum, wenn man die Wörter in Reihenfolge der Eingabe einträgt:



Um festzustellen, ob sich ein neues Wort bereits im Baum befindet, beginnt man an der Wurzel und vergleicht das neue Wort mit dem Wort, das im Wurzelknoten gespeichert ist. Handelt es sich um das gleiche Wort, ist die Frage bereits positiv beantwortet. Ist das neue Wort kleiner als das Wort im Baum, wird die Suche mit dem linken Nachkommen fortgesetzt; andernfalls wird beim rechten Nachkommen weiter untersucht. Gibt es in der gewünschten Richtung keine Nachkommen, befindet sich das neue Wort nicht im Baum, und tatsächlich ist die unbesetzte Position die richtige Stelle, um das neue Wort als Nachkomme einzutragen. Der Suchvorgang ist rekursiv, da die Suche von einem Knoten aus eine Suche von einem der Nachkommen aus verwendet. Rekursive Funktionen sind deshalb für Einfügen und Ausgeben wohl am natürlichsten.

Zurück zur Beschreibung eines Knotens; er wird am einfachsten als Struktur mit vier Komponenten repräsentiert:

```
struct tnode {        /* der Knoten eines Baumes: */
    char *word;           /* zeigt auf den Text */
    int count;            /* Haeufigkeit */
    struct tnode *left;   /* linker Nachkomme */
    struct tnode *right;  /* rechter Nachkomme */
};
```

Diese „rekursive" Vereinbarung eines Knotens sieht vielleicht riskant aus, sie ist aber korrekt. Eine Struktur darf sich zwar selbst nicht enthalten, aber

```
struct tnode *left;
```

vereinbart **left** als Zeiger auf einen Knoten **tnode**, nicht als Knoten **tnode** selbst.

Manchmal braucht man eine andere Art von rekursiven Strukturen: zwei Strukturen, die gegenseitig aufeinander verweisen. Dies kann folgendermaßen erreicht werden:

```
struct t {
    ...
    struct s *p;    /* p zeigt auf ein s */
};
struct s {
    ...
    struct t *q;    /* q zeigt auf ein t */
};
```

Das Programm selbst ist überraschend klein, wenn wir eine Handvoll Hilfsfunktionen wie **getword** benutzen, die wir bereits früher geschrieben haben. Das Hauptprogramm liest die Wörter mit **getword** und fügt sie in den Baum mit **addtree** ein.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define MAXWORD 100
struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);

/* Haeufigkeit von Worten zaehlen */
main()
{
    struct tnode *root;
    char word[MAXWORD];

    root = NULL;
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            root = addtree(root, word);
    treeprint(root);
    return 0;
}
```

addtree ist rekursiv. **main** präsentiert ein Wort an der obersten Ebene (der Wurzel) des Baumes. Auf jeder Ebene wird dieses Wort mit dem Wort verglichen, das im Knoten bereits gespeichert ist, und wird mit einem rekursiven Aufruf von **addtree** entweder zum linken oder zum rechten Unterbaum durchgereicht. Schließlich wird das Wort entweder im Baum gefunden (dann wird **count** inkrementiert) oder wir finden einen Nullzeiger, wo dann ein Knoten erzeugt und zum Baum hinzugefügt werden muß. Wird

ein neuer Knoten erzeugt, dann liefert **addtree** einen Zeiger auf diesen Knoten, der dann beim vorhergehenden Knoten eingefügt wird.

```
struct tnode *talloc(void);
char *strdup(char *);

/* addtree:  einen Knoten mit w bei oder nach p einfuegen */
struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;

    if (p == NULL) {    /* ein neues Wort */
        p = talloc();   /* neuen Knoten erzeugen */
        p->word = strdup(w);
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++;     /* Wort ist schon vorgekommen */
    else if (cond < 0)  /* kleiner: links darunter */
        p->left = addtree(p->left, w);
    else                /* groesser: rechts darunter */
        p->right = addtree(p->right, w);
    return p;
}
```

Eine Funktion **talloc** beschafft Speicherplatz für den neuen Knoten. Sie liefert einen Zeiger auf einen freien Speicherbereich, der geeignet ist, einen Knoten aufzunehmen. Das neue Wort wird mit Hilfe von **strdup** an eine verborgene Stelle kopiert. (Wir werden diese Funktionen in Kürze noch besprechen.) Der Häufigkeitszähler wird initialisiert, und die zwei Zeiger auf die Nachkommen werden auf **NULL** gesetzt. Dieser Teil der **addtree**-Funktion wird nur an den Blättern des Baumes ausgeführt, wenn ein neuer Knoten hinzugefügt wird. Wir haben (unvorsichtigerweise) auf Fehlerbehandlung bei entsprechenden Resultatwerten von **strdup** und **talloc** verzichtet.

treeprint gibt den Baum sortiert aus; bei jedem Knoten wird zunächst der linke Unterbaum ausgegeben (alle Wörter, die dem momentanen Wort vorausgehen), dann das Wort selbst, und dann der rechte Unterbaum (alle Wörter, die noch nachfolgen). Macht Rekursion Sie unsicher, dann sollten Sie nachvollziehen wie **treeprint** den oben gezeigten Baum abarbeitet.

```
/* treeprint:  Baum p sortiert ausgeben (Inorder) */
void treeprint(struct tnode *p)
{
    if (p != NULL) {
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}
```

Ein praktischer Hinweis: Wird der Baum „unbalanciert", weil die Wörter nicht durcheinander ankommen, dann kann die Laufzeit des Programms zu schnell wachsen. Im schlimmsten Fall sind die Wörter bereits sortiert, und dann ist dieses Programm die aufwendige Simulation einer linearen Suche. Es gibt Verallgemeinerungen von binären Bäumen, die nicht derartig in eine lineare Liste ausarten, aber wir wollen diese Bäume hier nicht beschreiben.