

Modulares Mikrocontrollersystem mit AVR ATmega32-16

Teil 4 Multitasking Basis-Software für die Praxis

Teil 4 wurde mit einer erweiterten Zielsetzung vollständig überarbeitet und neu strukturiert. Die Anwenderkommunikation bildet zwar immer noch den Schwerpunkt, aber im Ganzen gesehen beschreibt Teil 4 eine Basis- Software (man kann auch Betriebssystem sagen), auf der robuste praktische Anwendungen aufbauen können.

Sinnvoll ist eine Basis-Software deshalb, weil sie Lösungen für immer wiederkehrende Problemstellungen bietet. Neben der Anwenderkommunikation gehören dazu die Datenspeicherung und das Fehlermanagement. Um die Basis-Software mit Leben zu erfüllen, ist sie mit zwei typischen Applikationen ausgestattet: einer Zeitschaltuhr und einer Temperaturüberwachung, der eine Messung zugrunde liegt. Manche der vielfältigen Möglichkeiten der Anwenderkommunikation werden demonstriert, auch ohne mit einer Applikation verbunden zu sein.

Es empfiehlt sich, die Teile 1 bis 3 der Artikelserie zu kennen, um den größten Nutzen aus Teil 4 zu ziehen. Die Programmbeispiele laufen auf der in Teil 2 entwickelten Hardware.

Die Software im Überblick

Wie schon an anderer Stelle dargestellt, ist es vorteilhaft, jedes Hauptprogramm `main.c` zu nennen und Projekte anhand von Verzeichnisnamen zu unterscheiden. Die zum Teil 4 gehörende Software befindet sich im Verzeichnis `avrt4`. Es handelt sich im Grunde um *ein* Programm, nämlich `main4.c`. Für Demonstrationszwecke gibt es abgespeckte Versionen von `main4`, nämlich `main3.c`, `main2.c` und `main1.c`.

Weil `main.c` in `makefile` eingetragen ist, speichert man am besten das Hauptprogramm, mit dem man aktuell arbeiten möchte, unter `main.c`. Damit lässt sich dann auch experimentieren, ohne das Original zu beeinträchtigen.

Quelldateien, die zur Basis-Software gehören, sind `mtsys.c`, `mtlcd.c` und `mteecall.c`. Sie tragen also das Prefix `mt`. `convert.c` ist der ständige Begleiter von `mtlcd.c`. Die Quelldateien `temp.c` und `dtime.c` betreffen Anwendungen.

Die Konfiguration in mtsys.h und makefile ist für das komplexeste Programm, also main4.c gemacht und muss bei Verwendung der reduzierten Hauptprogramme nicht angepasst werden.

Das Prinzip der mt-Anwenderkommunikation

Die mt-Anwenderkommunikation hat eine so weitgehende Bedeutung, dass es sinnvoll ist, sie zunächst allgemein zu beschreiben und ihre Möglichkeiten erst später anhand der Programmbeispiele.

Bei der Frage, wie ich ihr Prinzip erläutern könnte, ist mir Windows in den Sinn gekommen. Auch Windows ist ein Multitasking-Betriebssystem. Seine Fenster waren die Attraktion der PCs der 90er Jahre. Fenster erfreuen jedoch nicht nur bis heute die Anwender. Sie sind einfach notwendig, um jeder Task so etwas wie einen eigenen Bildschirm zu bieten.

Glücklicherweise muss man für eine angemessene mt-Anwenderkommunikation nicht Fenstertechnik auf einem LCD mit 2 Zeilen realisieren. Eine Task, die auf einem MC läuft, kann nicht davon ausgehen, dass sie einem Anwender jederzeit etwas mitteilen und ihn zur Kommunikation auffordern kann. Ganz einfach deshalb, weil MCs, im Gegensatz zu üblichen PCs, meistens für Daueraufgaben eingesetzt werden und es niemand gibt, der pausenlos das LCD anschaut. Außerdem dürfen Tasks niemals ihre Arbeit einstellen, um auf eine Anwenderreaktion zu warten.

Deshalb sind Funktionen, mit denen Tasks etwas am LCD ausgeben oder von der Tastatur einlesen könnten, so gut wie überflüssig. Die Initiative zur Kommunikation ergreift nicht die Task, sondern allein der Anwender, was ausreichend ist und eine enorme Vereinfachung darstellt. Im Programmcode von Tasks kommt deshalb Kommunikation nicht vor. Die Koordination der Zugriffe auf die knappe Ressource LCD entfällt, und um die Zuteilung der Tastendrücke an die richtige Task muss sich das System auch nicht kümmern. Was den Tasks als Kommunikationsschnittstelle bleibt, sind die Daten, selbst dann, wenn Fehler zu melden sind.

Task LCD

Die gesamte Anwenderkommunikation ist Sache einer einzigen Task, und zwar der Task LCD. Sie heißt so, weil sie mit dem LCD eine Menge zu tun hat. Aber ihr Herz schlägt für die Abfrage der Tastatur, und zwar im Zyklus von 70 ms. Am LCD wird nur etwas als Folge eines Tastendrucks ausgegeben, oft auch indirekt. Wenn man so will, stellt die regelmäßige Aktualisierung von angezeigten Daten eine Ausnahme von dieser Regel dar.

LCD Anzeigeliste

Das zweite Ding ist eine Liste, in der alle Kommunikationsfälle zusammengestellt sind. Der Anwender wählt daraus den Fall aus, der für ihn von Interesse ist, denn die Initiative liegt bei ihm.

Wie sich zeigen wird, schränkt eine Liste, deren Komponenten von gleicher Struktur sein müssen, die Vielfalt der Kommunikationsfälle keineswegs ein. Ich nehme es vorweg: Anzeigen und Editieren von Zahlen verschiedenster Art sind möglich, bis hin zu Datum und Uhrzeit. Menüs gibt es, und geschützte Listenbereiche ebenso. Damit nicht genug: das Auslösen eines Vorgangs, der aus mehreren auszuwählen ist und der eine einfache

Quittung sein kann oder das Starten einer Task, ist genauso ein Kommunikationsfall wie das „Durchblättern“ und Editieren von Daten in Arrays.

Texte, auch veränderliche, werden nur angezeigt. Editieren ist überflüssig, weil MCs mit editierten Texten kaum etwas anfangen, ausgenommen Passwörter, die aber durch PINs ersetzt werden können.

val/sel Funktionen

Die Vielfalt der Kommunikationsfälle wird, neben vielen nur für die Task LCD geschaffenen Typen (LCD-Typen), mit Funktionen erzielt. Obwohl sie der Programmierer für den Einzelfall schreibt, sind sie doch Bestandteil der Task LCD. Je nach Zweck (Validierung oder Selektion) werden sie val- oder sel-Funktionen genannt. Die Sammlung dieser Funktionen ist das dritte Standbein der mt-Anwenderkommunikation.

Grenzen gibt es auch. Wenn der Anwender einen Kommunikationsfall ausgewählt hat, steht das gesamte LC-Display nur diesem zur Verfügung, und zwar solange bis ein anderer Fall gewählt wird. Die erste Zeile des Displays ist grundsätzlich einem konstanten Text vorbehalten und die zweite einer variablen Date (ich benutze diesen Begriff als Singular von *Daten*). Die Anzeige von zwei oder mehr Daten gleichzeitig ist also nicht möglich. Von dieser Festlegung kann kein Kommunikationsfall abweichen. (Statt dieses sperrigen Begriffs werde ich synonym auch Listenpunkt oder Listenposition verwenden, manchmal auch nur Punkt oder Position).

Minimalprogramm

Aus der Beschreibung oben folgt, dass bei der Programmierung der mt-Anwenderkommunikation, genauso wie bei der Programmierung von Tasks (siehe Teil 2), ein Programmierschema vorgegeben ist. Es betrifft die LCD-Anzeigeliste und die val/sel-Funktionen, die beide aus Gründen der Zweckmäßigkeit ihren Platz im Hauptprogramm haben.

Die Datei mtlcd.h und damit auch Datei mtlcd.c, die den Programmcode von Task LCD enthält, sind über mtsys.h eingebunden. Der Kopf der Headerdatei mtlcd.h muss passend zur verwendeten Hardware konfiguriert werden. Es gibt unterschiedliche Passagen für den AVR ATmega und den R8C von Renesas. Die letztere ist auskommentiert, weil sie hier keine Rolle spielt.

Falls erforderlich können die 6 Ausgänge für den Anschluss des LCD auch anders festgelegt werden als in mtlcd.h. Bit 0 des 4-Bit-Datenbusses kann auf einem der Bits 0-4 eines beliebigen Ports liegen. Ebenso beliebig können Ports und Bits den Steuersignalen E und RS zugeordnet werden. Mit den Programmzeilen unter INITPORTBITS werden die betreffenden 6 Bits als Ausgänge konfiguriert.

Mit dem Symbol KEYB_CHAN wird der Analogkanal festgelegt, an den die Tastatur angeschlossen ist.

Hauptprogramm

Das folgende Hauptprogramm main1.c demonstriert den Minimalfall, d.h. die LCD-Anzeigeliste enthält nur einen Kommunikationsfall, und es gibt keine val-/sel-Funktionen.

```

//AVR
#include <avr/interrupt.h>
#include <avr/pgmspace.h>

//mt
#include "mtsys.h"

#define NONAV 0
#define HIDDEN 0

//LCD-List
const struct lcdliststruct lcdlist[] PROGMEM = {
  {"clock", (void*) & clock, ULONG, NN, NN },
};

const unsigned char maxlistpos = sizeof (lcdlist) /
  sizeof (struct lcdliststruct) -1 ;
const unsigned char minlistpos = NONAV + HIDDEN;

////////////////////////////////////

unsigned int adc (unsigned char chan) {
  unsigned char az;
  ADMUX = ADMUX | (1<<REFS0); //interne Referenzspg.
  ADMUX = ADMUX & 0xF0; //Kanalwähler 0 setzen
  ADMUX = ADMUX + (chan & 0x07); //Kanal setzen
  ADCSRA = ADCSRA | 0x87 ; //ADEN und Prescaler setzen
  ADCSRA = ADCSRA | 0x40; //Start Conversion
  while ( ADCSRA & 0x40 ) NOP // Ende abwarten
  az = ADCL; //ganz wichtig: immer zuerst ADCL lesen!!
  return ((ADCH<<8) + az );
}

//////////////////////////////////Interrupt Service Routine //////////////////////////////////
ISR ( TIMER0_COMP_vect ) {
  isrmt ();
}

int main(void) {
  //////////////////////////////////Init Timer0 1kHz, CLOCK 16 MHz//////////////////////////////////
  TCCR0 = 0x08; //CTC-Mode
  //Timerinterrupts von
  TCCR0 = TCCR0 | 0x03; //Prescaler 16 MHz : 64 = 250 kHz,
  OCR0 = 249; //Vergleichswert
  TCNT0 = 0; //Timerzähler auf 0
  TIMSK |= 1<<OCIE0; //enable CTC-Interrupt Timer0

  //////////////////////////////////Init mt////////////////////////////////////
  initmt ();
  EI //enable Interrupt
  // STARTMT //Tasks starten
  start (LCD, 10, lcd1);

  schedule (); //läuft endlos
}

```

Die Aufgabe des Programms ist, die Variable clock, also die Laufzeit des Programms in ms, laufend anzuzeigen. Die Variable clock wird von mtsys.h zur Verfügung gestellt. Anzeigen heißt bei der mt-Anwenderkommunikation auch Aktualisieren, was etwa im Zyklus von 0,5 s geschieht.

Statt clock könnte zum Beispiel auch idle angezeigt werden. idle verhält sich jedoch bei kleinen Programmen recht statisch, so dass die Aktualisierung nicht so deutlich sichtbar ist wie bei der Variablen clock, die sich fortlaufend ändert.

Für diese einfache Aufgabe ist nichts weiter als ein Eintrag in der LCD-Anzeigeliste erforderlich, der hier nochmals dargestellt ist:

```

const struct lcdliststruct lcdlist[] PROGMEM = {
  {"clock", (void*) & clock, ULONG, NN, NN},

```

```
};
```

Programmtechnisch ist die LCD-Anzeigeliste ein konstantes Array `lcdlist[]`, dessen Elemente Strukturen sind, von denen jede einen Kommunikationsfall repräsentiert. Kommunikationsfälle zu generieren, ist Sache des Programmierers. Task LCD liest den jeweils benötigten Listenpunkt aus dem Programmspeicher aus, denn dort ist die Liste mit Hilfe von `PROGMEM` (s.u.) hinterlegt.

Die Bedeutung von `minlistpos`, `maxlistpos`, `NONAV` und `HIDDEN` wird im Rahmen des wesentlich komplexeren Programms `main2.c` erklärt.

Komponenten

Die ersten drei Komponenten der Struktur der Listenpunkte:

Text, der statisch in der ersten Zeile des LC-Displays angezeigt wird

Er dient der Beschreibung des Anzeigepunktes. Es ist zweckmäßig und in manchen Fällen unbedingt erforderlich, dafür immer 16 Zeichen vorzusehen. Mit weniger lässt sich kein Speicher einsparen. Es gibt eine Möglichkeit diesen Text vor der Anzeige zu verändern.

Zeiger auf die anzuzeigende Variable

Da diese Strukturkomponente nur für *einen* Zeigertyp definiert werden kann, aber die Variablen ganz verschiedene Typen haben können, werden ihre Zeiger durch eine explizite Typwandlung (`cast`) zu `void`-Zeigern gemacht, wie `(void*) &clock` zeigt.

LCD-Typ

LCD-Typen sind in `convert.h` definiert. Sie zeigen der Task LCD an, wie sie die betreffende Date behandeln soll. Der LCD-Typ `ULONG` zum Beispiel veranlasst Task LCD, die Variable `clock` als vorzeichenlose Zahl im Bereich der `unsigned long` Zahlen von C auszugeben.

Jeder der zur Zeit 19 LCD-Typen ist einem C-Typ zugeordnet. Zum Beispiel kann der C-Typ `unsigned long` je nach LCD-Typ als Zahl, Datum, Tageszeit oder Pseudo-Dezimalzahl behandelt werden.

In der vorletzten Komponente der Struktur kann ein Zeiger auf eine `val`-Funktion gespeichert werden und in der letzten ein Zeiger auf eine `sel`-Funktion. Wenn keine Funktion verwendet wird, wird `NN (0)` eingetragen. Näheres zu `val`-/`sel`-Funktionen ist bei den weiteren Hauptprogrammen zu finden.

Diverses

Ansonsten enthält das vorstehende Hauptprogramm die für `mt`-Programme übliche `main`-Funktion und die Interrupt Service Routine `ISR`. `STARTMT` ist auskommentiert, weil im Minimalprogramm nicht alle Tasks des Projektes benötigt werden, sondern nur Task LCD, die im Anschluss gestartet wird. Dieser Weg, die Komplexität zu verringern bzw. zu steigern, ist bequemer als jeweils `makefile` und die Konfiguration von `STARTMT` in `mtsys.h` zu ändern.

Neu ist die Funktion `adc()`. Wenn sie aufgerufen wird, startet sie die Analog-Digital-Wandlung für den spezifizierten Analogkanal des MC und gibt den Messwert zurück. Sie ist

sehr hardware-spezifisch. Quelldateien, die eine extern-Deklaration von `adc()` enthalten, können diese Funktion benutzen. Task LCD setzt `adc()` zur Tastaturabfrage ein.

Die inkludierte Bibliothek `avr/pgmspace.h` ermöglicht bei Datendefinitionen den Zusatz `PROGMEM`, durch den die Date vom Compiler im ROM abgelegt wird, und sie stellt Funktionen bereit, um damit umzugehen. Die Definition von `lcdlist[]`, also der LCD-Anzeigeliste, macht von `PROGMEM` Gebrauch.

Applikation Schaltuhr, Fehler und EEPROM

Mit der Anwendung Schaltuhr und dem, was sie praxistauglich macht, befasst sich das Hauptprogramm `main2.c`. Die Praxistauglichkeit ist nicht wenig und hat dazu geführt, dass die Darstellung von `main2.c` ziemlich umfangreich ist.

Schaltuhr

Doch zunächst ein Blick auf den folgenden Programmtext, der ein Auszug aus der Quelldatei `dtime.c` ist und der die Schaltuhr mit Hilfe der Task `DTIME` realisiert.

```
void timer (void) {
    if (isok(XTIM) ) {
        if ( tmfrom < tmto) {          //Mitternacht liegt nicht im Bereich from-to
            if (tmfrom <= tm && tm <= tmto ) TOUTON else TOUTOFF
        }
        else if (tmfrom > tmto) {      //Mitternacht liegt im Bereich from-to
            if ( !(tmto < tm && tm < tmfrom) ) TOUTON else TOUTOFF
        } else TOUTOFF
    }
    else TOUTOFF
}

void dtime2 (void) {                  //Uhr
    if (++tm == 86400) tm = 0;
    timer ();
    mtdelay (1000, dtime2);
}

void dtime1 (void) {
    INITTOUT
    callstr[XTIM] = ERROR;
    mtcoop ( dtime2);
}
```

Taskfunktion `dtime2()` ist eine einfache Uhr, die in der Variablen `tm` die Sekunden eines Tages hoch zählt. Eine Tageszeit wird nicht nur hier, sondern ganz generell durch die Anzahl der seit Mitternacht vergangenen Sekunden angegeben. Bei jeder neuen Sekunde prüft die Funktion `timer()`, ob die aktuelle Tageszeit innerhalb eines durch `tmfrom` und `tmto` vorgegebenen Zeitraums liegt. Falls ja, wird ein in `dtime.h` konfigurierbarer Ausgang gesetzt, sonst rückgesetzt, und zwar aus Sicherheitsgründen jedes Mal neu.

Das Besondere ist, dass in dem betreffenden Zeitraum Mitternacht liegen darf. Mitternacht bedeutet einen Bruch in der Zeitzählung, der die Abfrage erschwert (auf 86399 folgt 0). Deshalb wird zunächst ermittelt, ob der Zeitraum Mitternacht enthält. Falls ja, wird abgefragt, ob die Uhrzeit *nicht* im komplementären Zeitraum liegt, denn der hat diesen Bruch nicht.

Es geht aber in main2.c um mehr als die Schaltuhr. Vielmehr werden all die Software-Dinge eingeführt, die die Schaltuhr (und auch andere Anwendungen) praxistauglich machen:

Die Schaltuhr darf nach Reset nicht arbeiten, da die Uhr bei 0 zu zählen beginnt, was nicht der aktuellen Tageszeit entspricht. Also startet Task DTIME mit einem Fehler, der dann verschwindet, wenn die Uhr gestellt wird.

Auch darf nach Reset der eingestellte Zeitraum nicht verloren gehen. Deshalb müssen tmfrom und tmto dauerhaft gespeichert werden, wofür der ATmega32 EEPROM vorsieht.

Dann ist da noch der GAU, der Absturz des Programms. Es muss in diesem Fall sichergestellt sein, dass Ausgänge zurückgesetzt werden. Wenn beispielsweise mit der Schaltuhr ein Heizelement betrieben wird, darf es nicht eingeschaltet bleiben, wenn das Programm abstürzt.

LCD-Anzeigeliste

Die LCD-Anzeigeliste hat nun 9 Einträge. Nach dem Programmstart kann man die Liste mit der Auf- oder Ab-Taste umlaufend durchgehen, was ich als einfache Navigation bezeichne. Die Konstanten minlistpos und maxlistpos legen den Bereich der Liste fest, in dem die einfache Navigation ohne besondere Mittel möglich ist. Im einfachsten Fall steht in maxlistpos die Position des letzten Eintrags und in minlistpos die des ersten, also 0. Im Hauptprogramm main2.c ist allerdings minlistpos gleich 1, weil der erste Eintrag nicht durch einfache Navigation erreicht werden soll. Weshalb, wird später zur Sprache kommen.

```
//LCD-List
const struct lcdliststruct  lcdlist[] PROGMEM = {
//NONAV
  {"          ", (void*) callinfobuf2,STRING,  NN,          scallinfobuf},
//HIDDEN
  {"clock      ", (void*) & clock,    ULONG,   NN,    NN },
  {"idle       ", (void*) & idle ,    UINT,   NN,    NN },
  {"012345678 ERRORS", (void*) callstr,  STRING,  NN,    scallstr },
  {"DemoProgAbsturz ", (void*) "do it" ,  STRING,  NN,    sdo},
  {"Uhrzeit    ", (void*) & tm,      TIME,    vtm,    NN},
  {"Timer ON von ", (void*) & tmfrom,  TIME,    vtmfrom, NN},
  {"Timer ON bis ", (void*) & tmto,   TIME,    vtmto,  NN},
  {"TOutp..V PortB ", (void*) & PORTB,  BITS,   NN,    NN},
};

const unsigned char maxlistpos = sizeof (lcdlist) /
                                sizeof (struct lcdliststruct) -1 ;
const unsigned char minlistpos = NONAV + HIDDEN;
```

Die bekannte Variable idle ist in die Liste aufgenommen worden. Ihr C-Typ ist unsigned int, dem der LCD-Typ UINT entspricht.

Natürlich sind die drei Variablen der Schaltuhr in der Liste zu finden. Der LCD-Typ TIME bewirkt, dass dem Anwender z.B. die Variable tm, die vom C-Typ unsigned long ist, im üblichen Uhrzeitformat hh:mm:ss präsentiert wird. Wenn das Uhrzeit-Format editiert

wurde, wird es umgekehrt in eine unsigned long Zahl verwandelt, die zu speichern ist. Kalenderdaten (LCD-Typ DATE) werden ganz analog behandelt.

Der Ausgang, der von der Schaltuhr betätigt wird, liegt auf Bit 0 von Port B. Hardware ist dort in der Entwicklungsphase sicher nicht angeschlossen. Um testen zu können, bewirkt deshalb der Eintrag "Toupt..V..." mit dem LCD-Typ BITS in der Liste die bitweise Anzeige von Port B. Dazu einige Hinweise:

Wenn statt &PORTB der Eingangsport &PINB angegeben wird, können Aus- und Eingänge des Ports beobachtet werden. Offene Eingänge verändern ständig die Anzeige, was störend sein kann. Deshalb sollten offene (nicht gebrauchte) Eingänge zu Ausgängen konfiguriert werden, oder man beschaltet sie mit einem Pull Up Widerstand.

Bei Verwendung einer val-Funktion können Aus- und Eingänge editiert werden. Auf die möglicherweise entstehenden Konflikte ist natürlich zu achten.

Der LCD-Typ BITS kann auch auf Ganzzahlen angewandt werden, was sicher kaum praktische Bedeutung hat. In diesem Fall wird das niederwertigste Byte bitweise angezeigt.

val-Funktion und Editieren

Die Angabe einer val-Funktion in einem Listenpunkt der LCD-Anzeigeliste führt dazu, dass die betreffende Date editierbar ist, Strings ausgenommen. Der Anwender bemerkt das daran, dass der Cursor mit der Links-Taste in die Anzeige geholt werden kann.

Nach Abschluss des Editierens (Cursor wird mit der Rechts-Taste aus dem Anzeigefeld hinausgeschoben) wird die val-Funktion aufgerufen. Ihre wichtigste Aufgabe ist, die Date auf Gültigkeit zu prüfen (zu validieren) und zu speichern. Wenn eine Date editiert wurde, steht die entsprechende Zahl der val-Funktion in der Variablen num zur Verfügung.

Ich mache es mir jetzt einfach, indem ich die Passage über das Editieren aus der Datei mtlcd.h einfüge. Erläuterungen dazu sind kaum nötig.

```
Editieren (alle LCD-Typen, außer Strings):
```

```
Editieren einer Ziffer:
```

```
Mit der Auf-Taste wird die Ziffer inkrementiert und mit  
der Ab-Taste dekrementiert. Auf die höchste Ziffer  
folgt die niedrigste und umgekehrt.
```

```
Stellenzahl einer Zahl vergrößern durch Einfügen der Ziffer 0:
```

```
Falls Cursor auf Zeichen ' ' oder auf '.' bei float-Zahlen steht,  
fügen Auf- und Ab-Taste die Ziffer 0 nach dem betreffenden Zeichen ein.  
Das gilt auch für LCD-Typ CHARNUM, falls keine Feldbreite verwendet wird. Es  
wird allerdings die Ziffer eingefügt, die 0 entspricht.
```

```
Das gleiche geschieht, wenn der Cursor auf '+' steht und die Auf-Taste  
betätigt wird, oder wenn Cursor auf '-' steht und die Ab-Taste  
betätigt wird.
```

```
Editieren von '+' und '-':
```

```
Die Ab-Taste bei '+' erzeugt '-' und die Auf-Taste bei '-'  
erzeugt '+'.
```

```
In allen anderen Situationen haben Auf- und Abtaste keine Wirkung.
```

Die drei Variablen der Schaltuhr sind in der LCD-Anzeigeliste mit den val-Funktionen vtm(), vtmfrom() und vtmtto() ausgestattet. Die beiden letzteren verhalten sich gleich, nur mit unterschiedlichen Variablen, weshalb vtmtto() nicht dargestellt ist.

```

////////Uhrzeit
void vtm (void) {
    if (istime (num) ) {
        callstr[XTIM] = OK1 ;
        tm = num;
    }
    RLCD
}

////////Timer ON von
void savtmfrom (void) {
    wree ( EETMFROM );
    RLCD
}

void vtmfrom (void) {
    if (istime (num) ) {
        tmfrom = num;
        mtwait (EESEM, savtmfrom );
    }
    else RLCD
}

```

In jeder der drei val-Funktionen prüft istime(), ob num überhaupt eine Zeit darstellt. Denn die Eingabe von 99:xx:xx wird beim Abschluss des Editierens akzeptiert, was bedeutet, dass *keine Zeit* eingegeben wurde. Ausgegeben wird eine solche Eingabe mit 99:99:99. Bei den drei Variablen der Schaltuhr ist die Eingabe von *keiner Zeit* unzulässig, daher die Abfrage mit istime().

Die eigentliche Aufgabe der val-Funktionen ist das Speichern der eingegebenen Date. Das leitet über zum Thema des Speicherns im EEPROM.

EEPROM

Zugriffe wie auf RAM-Daten sind auf EEPROM-Daten nicht möglich, was mit der Architektur des ATmega32 zu tun hat. Das Programm arbeitet daher mit RAM-Daten, die eine Spiegelung der EEPROM Daten sind. Der Abgleich erfolgt durch wechselseitige Kopiervorgänge. Der erste findet gleich nach Reset statt, und zwar vom EEPROM zum RAM, später nur noch umgekehrt, was typisch für Eingabedaten ist. Nichts spricht dagegen, dass Tasks auch Log-Daten in den EEPROM zu schreiben.

Das nötige Werkzeug stellt die AVR-Definitionsdatei eeprom.h zur Verfügung. Die Speicherbelegung des EEPROM führt der Compiler aus, indem die betreffenden Datendeklarationen mit dem Zusatz EEMEM versehen werden. Der folgende Auszug aus dtime.c zeigt die Definitionen von tmfrom und tmtto im RAM und die gespiegelten im EEPROM.

```

unsigned long tm = 0;
unsigned long tmfrom = 0;
unsigned long tmtto = 0;

unsigned long eetmfrom EEMEM;
unsigned long eetmtto EEMEM;

```

Lesen und Schreiben eines EEPROM-Bytes kostet Zeit. Deshalb signalisiert ein Register-Bit das Ende jedes Lese- oder Schreibvorgangs.

Die Funktion rdee() kopiert einen Speicherbereich vom EEPROM in den RAM. Der Vorgang ist so kurz, dass er in einem Zug, also ohne kooperativ zu werden, durchgeführt werden kann.

Der umgekehrte Kopiervorgang, also das Schreiben in den EEPROM dauert etwa 8 ms je Byte. Er wird von der Funktion wree() angestoßen, die die Task EE startet, um ihn kooperativ auszuführen. Die Funktionen und die Task EE sind in mteecall.c zu finden. Task EE muss in mtsys.h nur definiert, aber nicht unter STARTMT eingetragen werden.

Die Funktionen wree() und rdee() haben die gleiche aufwändige Argumentenliste. Es ist zweckmäßig, dafür jeweils Symbole zu definieren, wie z.B. EETMFROM und EETMTO im folgenden Auszug aus dem Hauptprogramm, der den EEPROM betrifft.

```
//////////////////////////////////////EEPROM
#define EETMFROM      (void*)&eetmfrom, (void*)&tmfrom, sizeof(tmfrom)
#define EETMTO        (void*)&eetmto, (void*)&tmtto, sizeof(tmtto)

void eeinit (void) {          //nur vor dem Start des Schedulers aufrufen!!!
                              //am besten unter STARTMT
    keylevel = adc (KEYB_CHAN);
    if (KUP < keylevel && keylevel < KLEFT) { //Linkstaste gedrückt?
        initee ( EETMFROM );           //RAM > EEPROM, EEPROM enthält keine gültigen Daten
        initee (EETMTO );
    }
    else {
        rdee (EETMFROM);               //EEPROM > RAM, Normalfall
        rdee ( EETMTO );
    }
}
```

Ganz wichtig ist, dass kein neuer Schreib- oder Lesevorgang begonnen wird, solange der vorherige noch läuft. Deshalb gibt es die Semaphore EESEM, die das Ende jedes Vorgangs signalisiert. Sie ist in mtsys.h definiert und unter STARTMT mit signal (EESEM) anfänglich zu setzen.

Jeder Aufruf von wree() und rdee(), muss daher mit mtwait (EESEM,...) eingeleitet werden, was eine Fortsetzungsfunktion erfordert. Erst diese führt dann wree() oder rdee() aus. Das zeigt die val-Funktion vtmfrom() mit der Fortsetzungsfunktion savtmfrom(), siehe val-Funktion oben.

Der oben beschriebene Abgleich nach Reset wird ganz einfach mit der Funktion rdee() ausgeführt, und zwar bevor der Scheduler gestartet ist.

Wenn aber der EEPROM keine oder fehlerhafte Daten enthält, darf dieser Abgleich nicht stattfinden. Das ist z.B. der Fall, wenn ein fabrikneuer MC verwendet wird oder wenn sich die Reihenfolge oder Größe der abzugleichenden Daten im Laufe des Programmierfortschritts geändert hat. Ob dieser Fall vorliegt, kann die Software nicht mit Sicherheit feststellen.

Wie die Funktion eeinit() zeigt, werden in diesem Fall die initialisierten RAM-Daten in den EEPROM geschrieben. Die sind zwar oft nicht zutreffend, dafür aber gültig und vorübergehend brauchbar. Der Anwender oder der Programmierer entscheidet sich bei Reset mit der gedrückten Links-Taste für diese Art des Abgleichs. (Es gibt sicher noch andere Möglichkeiten, den EEPROM korrekt zu initialisieren.)

Weil der Abgleich vor dem Start des Schedulers erfolgen muss, sind dafür Task EE und damit die Funktion wree() nutzlos. Deshalb gibt es dafür die Funktion initee(), die auf Kooperation verzichtet, also 8 ms je Byte wartet. Da sie die gleiche Argumentenliste hat wie rdee() und wree(), können die dafür definierten Symbole verwendet werden.

Fehler

Für den Umgang mit Fehlern wird größtenteils die LCD-Anzeigeliste eingesetzt. Unter einem Fehler verstehe ich aber auch andere Nachrichten, die der Anwender unverzüglich erhalten muss. Weil per Definition die mt-Anwenderkommunikation diese Initiative nicht ergreifen kann, gibt es die kleine Task CALL in mteecall.c, deren Aufgabe es ist, den Anwender bei einem Fehler herbei zu rufen. Das geschieht durch die erhöhte Blinkfrequenz einer LED, die ansonsten so etwas wie ein Lebenszeichen des Systems ist. Die LED kann dort angebracht sein, wo eine gute Chance besteht, dass der Anwender sie bemerkt (Sammelstörmeldung). Der Ausgang für die LED wird in mteecall.h konfiguriert.

Der String callstr[] ist der zentrale Fehlerspeicher. Ein String als Fehlerspeicher hat den Vorteil, dass er einen vollständigen Überblick über alle bestehenden und nicht bestehenden Fehler, sowie deren Varianten liefert. Und mit Hilfe der LCD-Anzeigeliste kann der String ganz einfach angezeigt werden, wie die folgende Zeile zeigt:

```
{"012345678 ERRORS", (void*) callstr, STRING, NN, scallstr },
```

Jede Zeichenposition von callstr[] ist einem bestimmten Fehler zugeordnet. Es ist zweckmäßig für den Index zum Zugriff ein Symbol zu definieren. Zum Beispiel kann mit callstr[XTIM] (Position 3) auf den Fehler, den die noch nicht gestellte Uhr verursacht, zugegriffen werden.

Jede Task, die einen Fehlerspeicherplatz belegt, initialisiert ihn beim Start. Task DTIME macht das in der Funktion dtime1() mit ERROR (E), weil sie mit einem Fehler startet. Tasks, die einen Fehler erkennen, schreiben zweckmäßigerweise einen mnemonischen Großbuchstaben in den Fehlerspeicher. Wenn ein Fehler verschwindet, wird eines der Zeichen '-' ':' ';' für Fehlerfreiheit eingetragen. Es gibt drei Stück, um auch Varianten der Fehlerfreiheit angeben zu können.

Ob die Uhr gestellt wurde, kann Task DTIME nicht erkennen. Deshalb setzt die val-Funktion vtm(), mit der die Uhr gestellt wird, die Fehlerposition auf OK1, was dem Zeichen '-' entspricht.

Task EE muss vor den Tasks starten, die Anwendungen realisieren, weil sie den String callstr[] mit dem Zeichen '.' initialisiert, was "nicht belegt" oder "noch nicht belegt" bedeutet.

Mit der Funktion isok() kann jede Position von callstr[] auf Fehlerfreiheit abgefragt werden. Die Funktion timer() der Schaltuhr macht davon Gebrauch, weil die Abfrage bei nicht gestellter Uhr fehlerhafte Ergebnisse liefern würde.

Die kompakte Fehleranzeige mit Hilfe eines Strings gibt zwar einen guten Überblick, aber kaum einen Hinweis, um welchen Fehler es sich bei einer bestimmten Position handelt. Da hilft die sel-Funktion scallstr().

sel-Funktionen und Selektieren

Die Angabe einer sel-Funktion führt dazu, dass ein angezeigter String zu einem Wählfeld wird. Bei anderen LCD-Typen wird ein 2stelliges Wählfeld vorangestellt. Wie beim Editieren kann der Cursor mit der Rechts-Taste in die Anzeige geholt und verschoben werden. Wenn die Auf- oder Ab-Taste betätigt wird, ist die Position, auf der der Cursor steht, ausgewählt, und die sel-Funktion wird aufgerufen. In der Variablen curs steht die

aktuelle Cursorposition zur Verfügung, die um 16 erhöht ist, falls die Auf-Taste benutzt wurde.

Der erste Teil der sel-Funktion scallstr(), die im folgenden Auszug aus main2.c zu sehen ist, behandelt m ersten Teil die Selektion mit der Ab-Taste. Sie dient zur Anzeige einer Information über den ausgewählten Fehler, bestehend aus 2 Zeilen Klartext. Der gesamte Text ist im Array callinfo[] im ROM gespeichert. Der Text zum Fehler wird in die beiden Zeilenpuffer callinfobuf1[] und callinfobuf2[] kopiert.

```
//////////////////////////////////////LCD-Anzeigeliste
unsigned char oldpos = 0;
unsigned char oldcpos = 0;

#define NONAV 1
#define HIDDEN 0

#define CALLINFOPOS 0

/////
void scallinfobuf (void) {
    nextpos (oldpos);
    nextcurs (oldcpos);
    RLCD
}

/////012345678 ERRORS
char callinfo [] [LCDSIZE+1] PROGMEM =
{ "0qu Es gab einen",
  "Watchdog Reset  ",
  "1 nicht belegt  ",
  "",
  "2 nicht belegt  ",
  "",
  "3do   Uhr ist   ",
  "nicht gestellt  ",
  "4 nicht belegt  ",
  ""
};

char callinfobuf1 [LCDSIZE+1];
char callinfobuf2 [LCDSIZE+1];

void scallstr (void) {
    if (curspos < LCDSIZE) { // Info anzeigen
        for (unsigned char i=0; i <= LCDSIZE; i++) {
            callinfobuf1 [i] = pgm_read_byte_near (callinfo[curspos<<1] + i);
            callinfobuf2 [i] = pgm_read_byte_near (callinfo[(curspos<<1)+1] + i);
        }
        oldcpos = curspos; //zur Verw. in scallinfobuf
        oldpos = getpos(); // dto.
        nextpos (CALLINFOPOS);
        nextins ( (unsigned int) callinfobuf1, STRING);
        nextcurs (curspos);
        RLCD
        return;
    }
    switch (curspos) { //Quittierungen
        case XWD + 16 : callstr[ XWD ] = OK1;
            break;
    }
    RLCD
}
```

Es geht nun darum, den Listenpunkt vorzubereiten, der die beiden Zeilenpuffer zur Anzeige bringen soll. Es handelt sich um den folgenden (namenlos):

```
{ "          ", (void*) callinfobuf2, STRING, NN,          scallinfobuf},
```

Die Anzeige der zweiten Zeile ist mit `callinfobuf2` schon eingetragen. Die `next`-Funktionen bereiten die nächste Anzeige vor, die automatisch im Rahmen der zyklischen Aktualisierung erfolgt. Dabei kann auch ein anderer Listenpunkt angezeigt werden.

`nextins()` bewirkt in `scallstr()`, dass der statische Text der ersten Zeile mit dem Text in `callinfobuf1[]` rechtsbündig überschrieben wird. `nextins()` kann die erste Zeile auch mit einer Zahl, einer Uhrzeit oder einem Kalenderdatum überschreiben. Ganz wichtig ist, dass der vorhandene Text aus `LCDSIZE` Zeichen bestehen. Wären es weniger, würde die End-Null überschrieben.

Weil der Cursor zu sehen sein soll, wird `nextcurs()` aufgerufen, wodurch seine Position vorbereitet wird.

Schließlich wird mit `nextpos()` die Position des anzuzeigenden Listenpunktes angegeben. Die technischen Einzelheiten der drei `next`-Funktionen sind in `mtlcd.h` zu finden.

Bei der nächsten Aktualisierung, also praktisch sofort, ist der 2zeilige erklärende Text des betreffenden Fehlers im Display zu sehen.

Weil auch der Cursor zu sehen ist und eine `sel`-Funktion bei dem namenlosen Listenpunkt eingetragen ist, kann die Anzeige der beiden Textzeilen ganz einfach mit der Auf- oder Ab-Taste verlassen werden. Mit den gespeicherten Positionen `oldpos` und `oldcpos` sowie den `next`-Funktionen sorgt `scallinfobuf()` dafür, dass die Fehleranzeige, also `callstr[]` wieder genauso erscheint wie sie verlassen wurde.

Listenpunkte, deren Anzeige nur sinnvoll ist, wenn sie mit `next`-Funktionen vorbereitet wurden, dürfen nicht mit einfacher Navigation zugänglich sein, was für den besagten zutrifft. Erreicht wird das mit `NONAV 1` und damit durch `minlistpos` gleich 1. Solche Listenpunkte am Anfang der LCD-Anzeigeliste anzuordnen, hat den Vorteil, dass deren Positionen im Laufe des Programmierfortschritts nicht geändert werden müssen.

Watchdog

Um ein Programm nach einem Absturz neu zu starten, wird die Watchdog-Hardware eingesetzt, die wohl jeder MC haben dürfte. Das Prinzip ist einfach: Ein Timer wird zyklisch vom Programm neu gestartet. Dadurch kommt es nur zu einem Timeout, wenn dieser Neustart ausbleibt, was beim Programmabsturz oder bei einer Endlosschleife der Fall ist. Der Timeout löst dann einen Reset aus.

Task `CALL` in `mteecall.c` kümmert sich neben der Sammelstörmeldung auch um die Watchdog-Funktion, wobei die AVR-Definitionsdatei `avr/wdt.h` hilft. Wenn es einen Watchdog-Reset gibt, dann sollte sich das startende Programm gefahrlos auf den wie auch immer bestehenden Prozesszustand synchronisieren. Danach wäre der Watchdog-Reset vergessen, wenn es nicht ein Register-Bit gäbe, das ihn speichert.

Wird das gesetzte Bit beim Programmstart entdeckt, dann setzt Taskfunktion `call1()` das Bit zurück und dafür auf Position `XWD (0)` von `callstr[]` den Fehler `MESSAGE (M)`. Es handelt sich nicht wirklich um einen Fehler, sondern um eine Mitteilung, die für den Programmentwickler nicht uninteressant sein dürfte.

Das Programm kann von sich aus diesen Fehler nicht zurücksetzen, weil es nicht erkennen kann, wann dieser Mensch die Mitteilung wahrgenommen hat. Deshalb bietet die `sel`-Funktion des Fehlerstrings `scallstr()` in ihrem zweiten Teil die Möglichkeit, solche Fehler mit der Auf-Taste händisch zu quittieren.

Natürlich sollte der Watchdog auch getestet werden können. Dazu dient:

```
{ "DemoProgAbsturz ", (void*) "do it" , STRING, NN, sdo},
```

Wegen des Eintrags der sel-Funktion sdo() ist der Text "do it" ein Wählfeld, ohne dass die Cursorposition eine Rolle spielt. Nachdem der Cursor in die Anzeige geholt wurde, ruft die Auf- oder Ab-Taste die Funktion sdo() auf, die nichts weiter als eine Endlosschleife enthält:

```
////////DemoProgAbsturz  
void sdo (void) {  
    while (1) NOP ;  
}
```

Innerhalb 1 s entsteht ein Reset des Programms. An Position 0 der Fehleranzeige steht der Buchstabe M, ein Hinweis, dass zu quittieren ist.

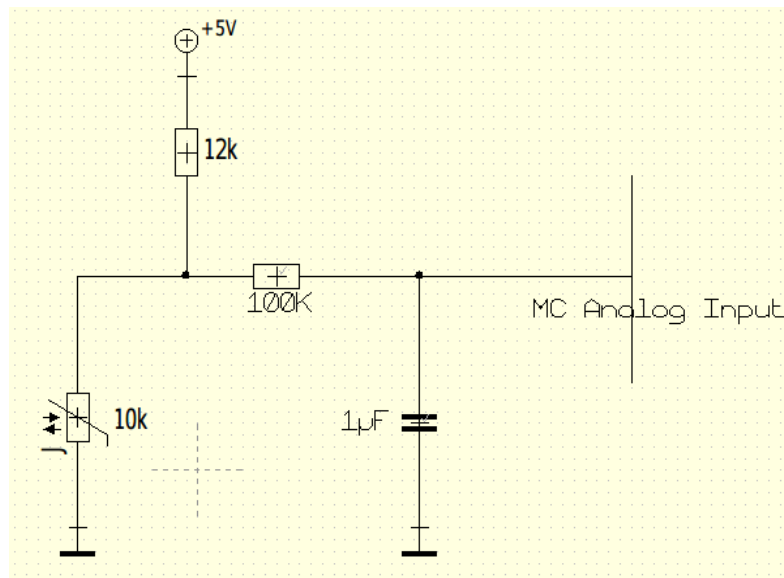
An der Sammelstörmeldung von Task CALL sind jetzt 2 Fehler beteiligt: Der Watchdog-Reset und die nicht gestellte Uhr.

Temperatur

Mit dem Hauptprogramm zuvor wurde der gesamte Rahmen deutlich, in dem sich die Programmierung auch größerer Anwendungen abspielt. In main3.c steht eine für die Lösung haustechnischer Probleme wichtige Anwendung im Vordergrund, nämlich die Messung einer Temperatur und deren operative Verwendung.

Hardware

Als Temperatursensor dient ein NTC [10k@20°](#), der mit dem Widerstand von 12k einen Spannungsteiler bildet. Ein NTC hat einen negativen Temperaturkoeffizienten, d.h. bei steigender Temperatur wird sein Widerstand geringer. Die Spannung des Spannungsteilers ist über einen Tiefpass an den Analogeingang 1 des ATmega32 angeschlossen, wie das folgende Schaltbild zeigt (in Teil 2 der Artikelserie wurde die entsprechende Hardware entwickelt) :



Bei der Analog/Digitalwandlung der Spannung am Analogeingang entstehen die ADC-Messwerte 0 bis 1023 (10 Bit). Die Temperatur auf der Mitte des gewünschten Messbereichs sollte daher einen ADC-Messwert um die 510 ergeben, was der Fall ist, wenn die beiden Spannungsteiler-Widerstände gleich groß sind, also der Widerstand des NTC ebenfalls 12k beträgt. Wegen des negativen Temperaturkoeffizienten des NTC ist dann seine Temperatur etwa 10° . Der Spannungsteiler-Widerstand von 12k ist daher gut geeignet für Messungen im Bereich von -20° bis $+40^{\circ}$, in dem zum Beispiel die Außentemperatur liegt. Temperaturen an Heizungsanlagen treten im Bereich von 10° bis 80° auf. Dafür sind 6k8 zu empfehlen.

NTCs sind nichtlinear, das heißt zwischen Widerstand bzw. dem ADC-Messwert und der Temperatur besteht *keine* Proportionalität.

LCD-Typ DEZ..

Man wird sich eine digitale Temperaturanzeige für den Hausgebrauch kaum anders vorstellen als in Form einer Dezimalzahl mit einer Stelle nach dem Dezimalpunkt, was nicht heißt, dass die Messung diese Genauigkeit hat. Temperaturen ändern sich meistens langsam. Wir empfinden es als angenehm, wenn sich dennoch die letzte Stelle der Anzeige ab und zu um 1 oder höchstens 2 verändert, woran zu erkennen ist, dass die Messeinrichtung arbeitet. Genau das soll die Ziffer nach dem Dezimalpunkt zeigen. Die Durchschnittsbildung aus mehreren Messungen hält die Ausschläge der Veränderung in den nötigen Grenzen.

Vordergründig müssten daher die Temperaturvariablen vom C-Typ float sein, wodurch auch die Berechnungen und Konvertierungen zu Ressourcen fressenden float-Operationen würden. Obwohl die mt-Anwenderkommunikation mit C-float-Typen umgehen kann, gibt es die LCD-Typen DEZ1, DEZ2 und DEZ3, die C-float-Typen in vielen Fällen überflüssig machen. Dem Typ DEZ.. liegt folgender Gedanke zu Grunde:

Alle Berechnungen erfolgen mit Ganzzahlen. Trotzdem wird das Ergebnis vom C-Typ long bzw. vom LCD-Typ DEZ.. dem Anwender als Dezimalzahl präsentiert, die ein Tausendstel der betreffenden Ganzzahl beträgt. Umgekehrt werden editierte DEZ..-Zahlen mit dem 1000fachen ihres im LCD sichtbaren Wertes als long-Zahl gespeichert.

Damit DEZ.. sinnvoll angewandt werden kann, müssen einige Bedingungen erfüllt sein. Hinter Zahlen vom LCD-Typ DEZ.. sollten sich physikalische Größen wie Spannung, Strom oder Temperatur verbergen, mit denen in technisch üblicher Genauigkeit (max. 4 relevante Stellen) Berechnungen auszuführen sind.

Berechnungen müssen so ausgeführt werden, dass Divisionsreste vernachlässigbar sind, d.h. der Dividend muss groß und der Divisor muss klein sein. Das ist bereits dadurch gegeben, dass die physikalische Größe mit ihrem 1000fachen Wert in die Rechnung eingeht. Und insbesondere auch dadurch, dass Multiplikationen immer *vor* Divisionen auszuführen sind.

Beispiel:

Wenn der ADC-Messwert 367 beträgt bei einer Referenzspannung von 4.9 V, dann gilt:
anliegende Spannung = ADC-Messwert * Referenzspannung / 1023

Die Referenzspannung als Zahl vom LCD-Typ DEZ.. steht mit 4900 im Speicher, folglich ist das Ergebnis auch eine DEZ.. Zahl. Die Rechnung lautet: $367 * 4900 / 1023 = 1757$ ohne den Divisionsrest. Also beträgt die anliegende Spannung +1.757 V. Und so würde sie auch als DEZ3-Zahl im LCD erscheinen. Der genauere Wert mit 6 Nachkommastellen beträgt 1.757869. Technisch ist er ohne Bedeutung.

Task TEMP

Die Aufgabe von Task TEMP ist, dem gesamten Programm den zyklisch aktualisierten Messwert einer Temperatur zur Verfügung zu stellen. Task TEMP ist in der Quelldatei temp.c zu finden. Hier der entsprechende Ausschnitt:

```
const long tlin[4] = TINIT ;
const long mlin[4] = MINIT ;

#define MINMW 100
#define MAXMW 1000

unsigned int savmwsun = 0;
long tp = 0;

//////////Task TEMP

void temp2 (void) {
    static unsigned char i = 0;
    static unsigned int mwsun = 0;
    unsigned int a;
    unsigned char m;

    a = adc (TCHAN);

    if ( (a < MINMW || a > MAXMW) && isok (XTP) ) //kommender Fehler
        callstr[XTP] = ERROR;
    else if ( (a >= MINMW && a <= MAXMW) && !isok (XTP) ) { //gehender Fehler
        i = 0;
        mwsun = 0;
        callstr[XTP] = OK1;
    }

    mwsun = mwsun + a; //Analogwerte von MMAX Messungen addieren

    if ( ++i >= MMAX) {
        i = 0;

        if (isok (XTP) ) {
            if ( mwsun < mlin[1]) m = 0; //Bereich auf der Messkurve bestimmen
            else if ( mwsun < mlin[2]) m = 1;
            else m = 2;

            //Innerhalb des Bereiches Temp. linear berechnen
            tp = ((mwsun - mlin[m]) * (tlin[m+1] - tlin[m] ) ) /
                (mlin[m+1] - mlin[m]) + tlin[m];
        }
        savmwsun = mwsun;
        mwsun = 0;
    }
}
```

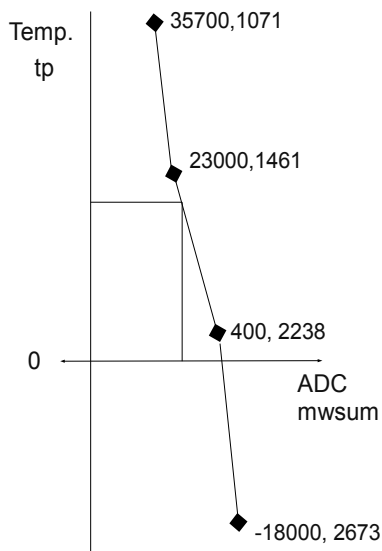
```

    mtdelay (2000, temp2);
    return;
}
mtdelay (1000, temp2);
}

void temp1 (void) {
    callstr[XTP] = OK1;
    mtcoop ( temp2);
}

```

Im Zyklus von einer Sekunde holt sich Taskfunktion temp2() den ADC-Messwert der Temperatur. Bei Unterbrechung der Sensorleitung dürfte der Wert >1000 und bei einem Kurzschluss < 100 sein. In beiden Fällen wird der Fehler XTP (1) gesetzt. Aber auch das Verschwinden des Fehlers wird erkannt, wodurch er sich zurücksetzen lässt. Das heißt, der Anwender muss sich im Fehlerfall nur um die Reparatur der Hardware kümmern, Quittieren ist überflüssig.



Lin. Graph von $tp = f(mwsun)$

In der Variablen mwsun werden die ADC-Messwerte von MMAX (3) Zyklen addiert. Gleich nach dem letzten Aufruf von adc() wird die Temperatur errechnet, sofern kein Fehler vorliegt. Nach zwei Sekunden geht es mit der nächsten Messung weiter.

Um die Temperatur zu berechnen, wird die nebenstehende Funktion benutzt. Sie gibt den Zusammenhang zwischen der Messwertsumme mwsun und der Temperatur tp an und ist durch vier Fixpunkte festgelegt.

Um die Koordinaten der Punkte zu festzulegen, werden der NTC und ein genaues Thermometer den verschiedenen Temperaturen ausgesetzt. Wenn es um die Messung der Außentemperatur geht, können das z.B. die Temperaturen von Kühltruhe (- 18000), schmelzendem Eis (400), Wohnraum (23000) und warmem Wasser(35700) sein.

NTCs in Kugelform umhülle ich zur Hälfte mit Schrumpfschlauch, wodurch sie schadlos (soweit ich feststellen konnte) kurzzeitig in Flüssigkeit getaucht werden können.

Taskfunktion temp2() errechnet nicht nur die Temperatur tp. Sie speichert auch die zugehörige Messwertsumme bis ein neuer Wert entsteht in der Variablen savmwsun, die genauso wie tp in die LCD-Anzeigeliste aufgenommen wird. Weil die Erweiterung der LCD-Anzeigeliste durch zwei Listenpunkte das einzige ist, was Task TEMP mit dem Hauptprogramm zu tun hat, wird es erst später betrachtet.

Die Temperatur, die das Thermometer zeigt, und die zugehörige Messwertsumme, die am LCD zu sehen ist, bilden jeweils die Koordinaten der zu bestimmenden Fixpunkte. Der Programmierer führt sie in Form der Definitionen MINIT und TINIT in das Programm ein (siehe temp.h). TINIT und MINIT initialisieren die Arrays tlin[] und mlin[].

```

#define TINIT    { 35700, 23000,  400, -18000 }    //Temp.punkte der Linearisierung,
//herkömmlich gemessen
#define MINIT    { 1071,   1461, 2238,  2673 }    //korresp. Werte savmwsun
// (Sum von MMAX adc-Aufrufen)

```

Vorausgesetzt, dass die Bestimmung der Punkte bei der Programmierung sorgfältig durchgeführt wurde, gibt es für den Anwender keinen Grund, sie zu wiederholen. Deshalb muss dafür auch keine Software vorgesehen werden.

Da die vier Fixpunkte durch drei Geraden verbunden sind, kann die Temperatur tp mit Hilfe einer Geradengleichung errechnet werden, wenn vorher mit Hilfe von $mwsun$ die zuständige Gerade ermittelt wurde. Von den Geraden sind jeweils zwei Punkte bekannt. Daher ist es zweckmäßig, für die Berechnung die aus der darstellenden Geometrie bekannte Zweipunktformel zu benutzen. Das ist die Funktionsgleichung einer Geraden, die sich durch die Koordinaten zweier Punkte in der Formel wiederfindet. Natürlich wird bei der Berechnung zuerst multipliziert und dann dividiert, so dass der Verwendung des LCD-Typs DEZ1 nichts im Wege steht.

Die aus der Messtechnik bekannten Schritte der Linearisierung, Kalibrierung und Justierung sind dadurch zu *einem* Vorgang zusammengefasst. Sollte der Linearisierungsfehler im Vergleich zu den anderen Fehlerarten groß sein, kann man auch mehr als vier Fixpunkte vorsehen.

Da die Messwertsummen der vier Punkte, genauso wie die der laufenden Messung, aus der Summe von MMAX (3) ADC-Messwerten bestehen, ist eine explizite Durchschnittsbildung nicht erforderlich.

Task TMONI

Abgesehen davon, dass größere örtliche Abstände zwischen Temperatursensor und Anzeige problemlos machbar sind, kann ein MC seine Möglichkeiten erst richtig ausspielen, wenn er Temperaturen nicht nur misst und anzeigt, sondern sie operativ nutzt. Und das macht Task TMONI, die die Temperatur tp laufend daraufhin überprüft, ob sie die einstellbare Temperatur $tlim$ überschreitet. Es ist aber nichts vorgesehen, was in diesem Fall aus- oder einzuschalten wäre. Vielmehr soll der Anwender benachrichtigt werden.

Task TMONI ist genauso wie Task TEMP in der Quelldatei temp.c enthalten:

```
//////////Task TMONI    (Temp. Monitoring)

long tlim = 0;
unsigned int wpause = 100;

long eetlim EEMEM ;
unsigned int eewpause EEMEM ;

void tmoni2 (void) {
    static unsigned int time = 0;
    if ( !isok(XTP) ) callstr[XTMON] = ERROR ;
    switch ( callstr[XTMON] ) {
        case OK1 : if (tp > tlim) callstr[XTMON] = WARNING ;
                    break;
        case WARNING : if ( tp <= tlim - HYST ) callstr[XTMON] = OK1 ;
                        break;
        case OK2 : if ( tp <= tlim - HYST ) { //OK2 entsteht durch Quittieren
                    time = 0;
                    callstr[XTMON] = OK1 ;
                }
                else if ( ++time >= wpause/MCYC ) {
                    time = 0;
                    callstr[XTMON] = WARNING ;
                }
                break;
        case ERROR: if ( isok (XTP) ) callstr[XTMON] = OK1;
                    break;
    }
}
```

```
    mtdelay (MCYC*1000, tmoni2);
}

void tmoni1 (void) {
    callstr[XTMON] = OK1;
    mtcoop (tmoni2);
}
```

Für die Benachrichtigung des Anwenders wird das bereits beschriebene Fehlermanagement benutzt. Task TMONI beansprucht dafür das Byte an Position XTMON (2) von callstr[]. Dieses Fehlerbyte hat eine zentrale Bedeutung für Task TMONI. Das heißt, sein Inhalt bestimmt wesentlich, was jeweils im Zyklus von Taskfunktion tmoni2() zu geschehen hat.

Fehler und ihre Varianten werden im Fehlerbyte am besten mit Großbuchstaben eingetragen. Vordefiniert sind ERROR (E), WARNING (W) und MESSAGE (M). Die Zeichen für Fehlerfreiheit sind festgelegt: OK1 (-) steht für generelle Fehlerfreiheit, und die Symbole OK2 (:) sowie OK3(;) stehen für Varianten davon, die meistens aber eine eingeschränkte Fehlerfreiheit bedeuten.

Eine Variante des Fehlers XTMON, wird als ERROR (E) angezeigt. Diese Fehlervariante erbt Task TMONI von Task TEMP, denn bei einem defekten Temperatursensor (XTP) kann Task TMONI nicht korrekt arbeiten. Der Fehler hat Priorität und überschreibt das Fehlerbyte unabhängig von seinem aktuellen Inhalt. Und er verschwindet von selbst wie sein Pendant in Task TEMP. Dabei wird zunächst für XTMON Fehlerfreiheit (OK1) angenommen (siehe case ERROR:).

Unter case OK1: kann dann die Funktion ihre eigentliche Arbeit tun, nämlich eine Nachricht für den Anwender zu produzieren, falls die Temperatur tp die vorgegebene Temperatur tlim überschreitet. Auf den ersten Blick besteht die Nachricht nur aus einem einzigen Großbuchstaben, nämlich W für WARNING, und der schnell blinkenden LED. Die Warnung wird durch den bereits beschriebenen Mechanismus der zweizeiligen Zusatzinformation deutlicher ausgedrückt.

Wenn die Temperatur tp unter die Temperatur tlim abzüglich der Hysterese HYST fällt, bevor der Anwender die Nachricht wahrgenommen hat, hat sie keine Bedeutung mehr und wird unter case WARNING: auf OK1 zurückgesetzt (mehr zur Hysterese siehe unten).

Dass er die Nachricht erhalten hat, zeigt der Anwender durch Quittieren an (vergl. Watchdog). In das Fehlerbyte wird OK2 (:) geschrieben, was vorläufige Fehlerfreiheit bedeutet. Generelle Fehlerfreiheit mit OK1 gibt es erst, wenn die Temperatur durch irgendeine Maßnahme des Anwenders auch tatsächlich abgesunken ist (siehe case OK2:). Da es sein kann, dass der Anwender nichts unternimmt, gibt es einen Timeout nach der einstellbaren Zeit wpause. Wenn diese abläuft, wird die vorläufige Fehlerfreiheit zurückgenommen und wieder W für Warning gesetzt.

Was hat es mit der Hysterese auf sich ?

Es ist nicht sinnvoll, einfach bei *Unterschreiten* der Grenztemperatur den Fehler zurückzunehmen. Die Temperatur tp verändert sich nur langsam, und es ist möglich, dass sie eine Weile aus messtechnischen Gründen um die Grenztemperatur pendelt. Weil die entsprechenden Reaktionen nicht gewollt sind, gibt es HYST. Erst wenn tp die Temperatur (tlim-HYST) unterschreitet, entsteht Fehlerfreiheit. Der Vorgang hat die Merkmale einer Hysterese, weil zwischen tlim und (tlim-HYST) die Warnung besteht oder auch nicht, abhängig davon, ob die Temperatur tp den Bereich fallend oder steigend durchläuft.

Hauptprogramm

Was ist nun durch Task TEMP und Task TMONI im Hauptprogramm dazugekommen?

Es sind zunächst die folgenden Listenpunkte der LCD-Anzeigeliste:

```
{ "Temperatur      ", (void*) & tp,      DEZ1,      NN,      NN},
{ "Grenztemperatur", (void*) & tlim,     DEZ1,      vtlim,   NN},
{ "Alarmpause     ", (void*) & wpause,   UINT,     vwpause, NN},
{ "adc sum        ", (void*) & savmwsom, UINT,     NN,      NN},
{ "TOutp..V PortB", (void*) & PORTB,   BITS,     NN,      NN},
```

Die Grenztemperatur tlim und die Alarmpause wpause besitzen die folgenden val-Funktionen, die sie jeweils im RAM *und* im EEPROM speichern. Man könnte das auch mit einer Gültigkeitsprüfung verbinden, was aber nicht sein muss.

```
////////Grenztemperatur
void savtlim (void) {
    wree ( EETLIM );
    RLCD
}

void vtlim (void) {
    tlim = num;
    mtwait (EESEM, savtlim );
}

////////Alarmpause
void savwpause (void) {
    wree ( EEWPAUSE );
    RLCD
}

void vwpause (void) {
    if (num < 10000) {
        wpause = num;
        mtwait (EESEM, savwpause );
    }
    else RLCD
}
```

Zum Speichern in den EEPROM sollte immer auf die Semaphore EESEM gewartet werden, auch wenn das in Einzelfällen nicht nötig ist. Man spart es sich das abzuklären, was auch daneben gehen kann. Das gilt genauso für's Lesen, was hier aber nicht vorkommt.

Nicht zu vergessen die Definitionen EETLIM und EEWPAUSE, die in die Funktion eeinit() eingefügt werden müssen:

```
#define EETLIM          (void*)&eetlim,   (void*)&tlim,   sizeof(tlim)
#define EEWPAUSE      (void*)&eewpause, (void*)&wpause, sizeof(wpause)
```

Da zwei Fehler hinzugekommen sind, sind deren Infotexte in callinfo einzutragen (hier nicht dargestellt), und die sel-Funktion scallstr() ist mit der Quittierung des Fehlers XTMON durch die Auf-Taste zu erweitern. Das war's!

```
switch (curpos) {
    //Quittierungen
    case XWD + 16 : callstr[ XWD ] = OK1;
    break;
    case XTMON + 16: callstr[ XTMON ] = OK2;
    break;
}
```

Menü, Array und mehr

Beim Hauptprogramm main4.c kommen gegenüber main3.c keine neuen Tasks hinzu. Es beschäftigt sich mehr mit sich selbst und wird komplexer. Mit einem Hauptmenü kommt Übersichtlichkeit in die LCD-Anzeigeliste, und es wird ein Array für Uhrzeitdaten vorgestellt, allerdings ohne diese Daten zu nutzen. Außerdem gibt es noch zwei Umrechnungen, bei denen das Kalenderdatum eine Rolle spielt.

Hauptmenü

Die bisherige LCD-Anzeigeliste ist zwar noch zu überblicken. Damit sie aber größer werden kann, ohne Übersichtlichkeit zu verlieren, wird der Listenpunkt *Hauptmenue* eingeführt. Um ihn als Menü nutzen zu können, müssen Bereiche gebildet werden, in denen jeweils thematisch zusammen gehörende Listenpunkte zusammengefasst sind.

Es beginnt mit NONAV und HIDDEN. Diese beiden Bereiche sind nicht durch einfache Navigation zu erreichen (Durchgehen der ganzen Liste mit der Auf- und der Ab-Taste).

Während in NONAV überhaupt keine Navigation möglich ist, kann man HIDDEN über das Hauptmenü erreichen. Unter HIDDEN sind die Listenpunkte für Experten zu finden. Es ist kein Problem, dafür eine Zugangskontrolle mit einer PIN einzubauen.

Das Hauptmenü steht unter MENU. Es könnte auch aus zwei oder noch mehr Listenpunkten bestehen.

Selbstverständlich bekommen die Applikationen Temperatur und Schaltuhr jeweils einen eigenen Bereich, nämlich TMP und TIM. Und im Bereich ERR steht die Fehleranzeige.

Bis auf *DemoProgAbsturz* sind die Listenpunkte unter DEMO neu. Was da demonstriert wird, wird im Anschluss beschrieben.

```
//LCD-List
const struct lcdliststruct  lcdlist[] PROGMEM = {
//NONAV
  {"
  {"Time Array      ", (void*) & tcopy,    TIME,    vtcopy,    stcopy },

//HIDDEN
  {"adc sum        ", (void*) & savmwsun, UINT,    NN,        NN},
  {"TOutp..V PortB ", (void*) & PORTB,   BITS,    NN,        NN},
  {"clock          ", (void*) & clock,   ULONG,   NN,        NN },
  {"idle           ", (void*) & idle ,    UINT,    NN,        NN },

//MENU
  {"Hauptmenue     ", (void*) "TMP TIM DEMO E X", STRING, NN, smenu},

//ERR (E)
  {"012345678 ERRORS", (void*) callstr,   STRING, NN,    scallstr },

//TMP
  {"Temperatur     ", (void*) & tp,      DEZ1,    NN,        NN},
  {"Grenztemperatur ", (void*) & tlim,   DEZ1,    vtlim,    NN},
  {"Alarmpause     ", (void*) & wpause, UINT,    vwpause,  NN},

//TIM
  {"Uhrzeit        ", (void*) & tm,      TIME,    vtm,      NN},
  {"Timer ON von   ", (void*) & tmfrom, TIME,    vtmfrom,  NN},
  {"Timer ON bis   ", (void*) & tmto,   TIME,    vtmtto,   NN},

//DEMO
  {"Dat>>WochTag   ", (void*) & date,    DATE,    vdate,    NN},
  {"Wochentag      ", (void*) & ptxt,    PSTRING, NN,        NN},
  {"Dat>>WochNr    ", (void*) & wdate,   DATE,    vwdate,   NN},
  {"Time Array Index", (void*) & tpos,    UCHAR,   vtpos,    NN },
  {"DemoProgAbsturz", (void*) "do it" ,  STRING,  NN,        sdo},
};
```

Benutzt wird das Hauptmenü wie folgt: Man geht in den Menüpunkt *Hauptmenue* (nach dem Programmstart ist man schon dort) und wählt mit dem Cursor einen Bereich. E steht für ERR und X steht für HIDDEN. Bei diesem einfachen Beispiel bedeuten X und HIDDEN dasselbe. Der Unterschied muss dennoch sein, weil HIDDEN dafür vorgesehen ist, mehr als einen Bereich aufzunehmen.

Einen Bereich zu wählen bedeutet, dass der erste Listenpunkt des Bereichs in die Anzeige kommt und die einfache Navigation durch Manipulation von minpos und maxpos auf diesen Bereich eingeschränkt wird. Das erledigt die zum Hauptmenü gehörende self-Funktion smenu().

```
void smenu (void ) {
  switch (curspos % 16 ) {
    case 15:
      minpos = NONAV;
      maxpos = minpos + HIDDEN -1;
      break;
    case 13: minpos = NONAV+HIDDEN+MENU;
      maxpos = minpos + ERR -1;
      break;
    case 0: case 1: case 2:
      minpos = NONAV+HIDDEN+MENU+ERR;
      maxpos = minpos+TMP-1;
      break;
    case 4: case 5: case 6:
      minpos = NONAV+HIDDEN+MENU+ERR+TMP;
      maxpos = minpos+TIM-1 ;
      break;
    case 8: case 9: case 10: case 11:
      minpos = NONAV+HIDDEN+MENU+ERR+TMP+TIM;
      maxpos = minpos + DEMO-1 ;
      break;
  }
  nextpos (minpos);
  RLCD
}
```

Hinter der Definition eines Bereichsnamens verbirgt sich die Anzahl der Listenpunkte des Bereichs. Wo ein Bereich beginnt und endet, muss nicht definiert werden, weil die Grenzen errechnet werden können, wie in smenu() zu sehen ist.

```
//Anzahl der Einträge in den Rubriken der Anzeigeliste
#define NONAV      2
#define HIDDEN    4
#define MENU      1
#define ERR       1
#define TMP       3
#define TIM       3
#define DEMO      5
```

Mit einer einfachen Änderung der Bereichsgröße kann man so im Laufe des Programmierfortschritts den betreffenden Bereich vergrößern oder verkleinern. Mehr muss nicht gemacht werden.

Die Bedingung dafür ist, dass die Reihenfolge der Bereiche nicht verändert wird, was keine bedeutsame Einschränkung ist.

Array

Das Array `timarr[]` speichert eine Liste von Uhrzeiten, die im LCD nicht nur angezeigt, sondern auch editiert werden. Weil die Daten einen Reset überstehen sollen, werden sie im EEPROM aufbewahrt. Deshalb ist `timarr[]` mit dem Zusatz `EEMEM` definiert.

`tpos` enthält den Index auf das aktuelle Element des Arrays. Nur das wird im RAM gespiegelt, und zwar in `tcopy`. Zum Datentransfer zwischen `tcopy` und `timarr[]` dienen die Funktionen `wree (EETIMARR)` und `rdee (EETIMARR)`, wobei `EETIMARR` für die etwas komplizierte Parameterliste steht.

```
#define MAXTIM 5
unsigned char tpos = 0;
unsigned long timarr [MAXTIM] EEMEM ;
unsigned long tcopy = 0;
```

Zugriffe auf das Array, sowohl zum Schreiben als auch zum Lesen, erfolgen mit Hilfe von `tcopy`. Diese Date ist es auch, die im LCD angezeigt und editiert wird. Sie ist als Listenpunkt *Time Array*, der vom LCD-Typ `TIME` ist, im Bereich `NONAV` enthalten und darf durch einfache Navigation nicht erreichbar sein, weil die sich nicht um Zusatzangaben kümmert. Die Position von *Time Array* ist in `TIMEARRAYPOS` definiert.

```
{"Time Array      ", (void*) & tcopy,    TIME,    vtcopy,    stcopy },
```

Die Verwendung des Arrays beginnt damit, dass im Listenpunkt *Time Array Index* aus dem Bereich `DEMO` der Index auf das Element des Arrays eingegeben wird, mit dem man sich befassen möchte.

```
{"Time Array Index", (void*) & tpos ,    UCHAR,    vtpos,    NN },
```

Nach der Eingabe wird die `val`-Funktion `vtpos()` aufgerufen, die letztlich den Listenpunkt *Time Array* mit korrekten Zusätzen zur Anzeige bringt.

In `oldpos` wird die aktuelle Position, also die von *Time Array Index*, gespeichert, um nach Verlassen von *Time Array* wieder zu *Time Array Index* zurückkehren zu können. Dann wird die Anzeige von `tcopy` vorbereitet. `nextpos()` setzt die Listenposition von `tcopy`.

Weil wegen der Funktion `rdee()`, die das aktuelle Element des Arrays nach `tcopy` lädt, auf die Semaphore `EESEM` gewartet werden muss, gibt es die Funktion `gettcopy()`. Sie sorgt mit `nextins()` auch dafür, dass der aktuelle Index `tpos` als Zusatz in der ersten Zeile der Anzeige erscheint und bestimmt mit `nextcurs()` die Position des Cursors. Erst jetzt ist alles bereit.

```
void gettcopy (void) {          //tcopy aus EE laden
    nextins (tpos, UCHAR);
    nextcurs (1);
    rdee (EETIMARR);
    RLCD
}
```

```
void vtpos (void) {            //Eingabe des Index
    if (num < MAXTIM) {
        tpos = num;
        oldpos = getpos();
    }
}
```

```

nextpos (TIMEARRAYPOS);
mtwait (EESEM, gettcopy);
}
else RLCD
}

```

Und so sieht die Anzeige dann aus:

```

Time Array      2
<≥00:00:00

```

In der ersten Zeile rechts wird der jeweilige Index angezeigt und in der zweiten Zeile die Date, der ein Wählfeld aus den Zeichen <> vorangestellt ist. Zu diesem Wählfeld kommt es, weil im Listenpunkt eine sel-Funktion eingetragen ist und die Date kein String ist.

Das Wählfeld bietet vier Wahlmöglichkeiten, die mnemonisch als Himmelsrichtung definiert sind:

```

NW  NO
  <>
SW  SO

```

Zum Beispiel wird NW gewählt, wenn der Cursor auf dem Zeichen < steht und die Auf-Taste betätigt wird.

Was sich hinter den Himmelsrichtungen verbirgt, bestimmt die sel-Funktion stcopy().

```

void stcopy (void ) {
switch (curspos) {
case SO :  if (++tpos == MAXTIM) tpos = 0;
           break;
case NO :  if (tpos) tpos--; else tpos = MAXTIM - 1;
           break;
case SW :
case NW :  nextpos (oldpos);
           RLCD
           return;
}
mtwait(EESEM, gettcopy); //bei SO u. NO, EEMEM auslesen
}

```

SO zählt den Index umlaufend aufwärts und NO umlaufend abwärts. Und die oben schon verwendete Funktion gettcopy() bereitet die Neuanzeige vor, sobald die Semaphore EESEM zu haben ist.

SW und NW bringen wieder den Listenpunkt *Time Array Index* in die Anzeige. Dann kann die Arbeit am Array beendet werden, oder man gibt einen weiter entfernten Index ein, der mit Durchblättern nur schwer zu erreichen ist.

Im Listenpunkt *Time Array* ist auch die val-Funktion stcopy() eingetragen. Dadurch wird die Uhrzeit in der Anzeige editierbar.

```

void savtcopy (void) {
wree ( EETIMARR ); //tcopy in EE speichern
RLCD
}

void vtcopy (void) {
tcopy = num;
mtwait (EESEM, savtcopy);
}

```

Es gibt keine Gültigkeitsprüfung, denn jede Uhrzeit ist zulässig, auch der Eintrag für *keine Uhrzeit* (99:xx:xx oder 99:99:99 bzw. NOTIME). Die einzige Aufgabe von `vtcopy()` ist, die Eingabe zu speichern, und zwar sowohl in `tcopy` als auch im EEPROM.

Noch einen letzten Punkt gibt es: Im Fall, dass das Array im EEPROM beim Programmstart keine gültigen Daten enthält, muss es initialisiert werden, wie alle anderen EEPROM-Daten auch (siehe oben). Dafür wird die Funktion `inittimee()` in `eeinit()` eingebaut.

```
void inittimee (void) {          //EE-Init beim Start, Aufruf s.o.
    tcopy = NOTIME;
    for (tpos = 0; tpos < MAXTIM; tpos++) initee (EETIMARR);
    tpos = 0;
}
```

Das normale Initialisieren von RAM-Daten nach Programmstart ist beim Array überflüssig.

Kalenderdatum und Wochennummer

Ein Kalenderdatum ist bei der `mt`-Anwenderkommunikation eine Zahl vom C-Typ *unsigned long*. Präsentiert wird sie dem Anwender im Datumformat `dd:mm:yy`, wofür der LCD-Typ `DATE` verantwortlich ist. Die Zahl enthält die Anzahl von Sekunden, die seit 01.01.2000 0 Uhr bis zum einem beliebigen späteren Datum ebenfalls 0 Uhr vergangen sind. Sie ist daher ein Vielfaches von 86400. Man kann sie auch Datumszahl nennen.

Eine solche Datumszahl ist `wdate` im Listenpunkt `Dat>>WochNr`.

```
{"Dat>>WochNr", (void*) & wdate, DATE, vwdate, NN},
```

Bei der Schaffung der LCD-Typs `DATE` habe ich bewusst „die Sünde“ des vergangenen Jahrhunderts begangen, nämlich die Jahreszahl zweistellig darzustellen. Denn es vereinfacht die Sache sehr, weil man sich mit der Jahreszahl dann nur im gegenwärtigen Jahrhundert bewegen kann. Und es ist sichergestellt, dass der C-Typ `unsigned long` für die Maximalzahl der Sekunden (also für 100 Jahre) ausreicht. Einen Jahrhundertwechsel wird `mt`-Multitasking wohl nicht erleben.

Weil in `Dat>>WochNr` die `val`-Funktion `vwdate()` eingetragen ist, ist `wdate` editierbar. Nach der Eingabe wird das Datum geprüft und gespeichert. Die Eingabe von *kein Datum* (99.xx.xx oder 99.99.99 bzw. `NODATE`) ist nicht erlaubt, weil die Aufgabe besteht, die zum betreffenden Datum gehörende Wochennummer zu errechnen und sie als Zusatz rechts oben anzuzeigen.

```
void vwdate (void) {
    if (isdate (num) ) {
        wdate = num;
        nextins (weeknum(num), UCHAR);
        nextcurs (0);
    }
    RLCD
}
```

Die Funktion `vwdate()` hat keine große Mühe mit dieser Aufgabe, da die Funktion `weeknum()` aus `convert.h` die Wochennummer zurück gibt. (Die Kalenderdaten 01. und

02.01.00 liefern kein korrektes Ergebnis, weil sie zur letzten Woche des Jahres 1999 gehören.) Kaum bekannt sind die Regeln zur Errechnung der Wochennummer beim Jahreswechsel:

Wenn das alte Jahr an der Woche des Jahreswechsels einen Anteil von mehr als 3 Tagen hat, dann ist diese Woche die 52. oder 53. des alten Jahres. Im anderen Fall ist sie die 1. Woche des neuen Jahres. Der Montag ist der 1. Tag der Woche.

Den Listenpunkt *Dat>>WochNr* mit einfacher Navigation zu erreichen, ist, im Gegensatz zu oben, zulässig. Mehr noch, die Anzeige darf, wenn sie erscheint, gar keine Zusatzangabe haben, da diese erst nach der Eingabe eines Datums als Rechenergebnis entsteht.

Kalenderdatum Wochentag

Für das nächste und letzte Beispiel sind in der LCD-Anzeigeliste zwei Listenpunkte vorgesehen:

```
{"Dat>>WochTag      ", (void*) & date,    DATE,    vdate,    NN},
{"Wochentag         ", (void*) & ptxt,    PSTRING, NN,      NN},
```

Das Datum in der Variablen *date* wird editierbar angezeigt. Nach Abschluss der Eingabe besteht die Aufgabe, den zum Datum gehörenden Wochentag zu errechnen und in Textform anzuzeigen.

Der Wochentag ist, genauso wie die Wochennummer oben, ein Rechenergebnis, das jeweils als Zusatz rechts oben angezeigt werden könnte. Darauf wird verzichtet, um mit dem Listenpunkt *Wochentag* den LCD-Typ *PSTRING* demonstrieren zu können. Dieser Listenpunkt erscheint nicht von selbst. Der Anwender muss er die *Ab*-Taste betätigen, um das Ergebnis zu sehen, denn nach der Eingabe des Datums soll er noch betrachten können, was er eingegeben hat.

```
char wd [] [11] =
{ "Samstag",
  "Sonntag",
  "Montag",
  "Dienstag",
  "Mittwoch",
  "Donnerstag",
  "Freitag", };

char * ptxt = wd [0];

unsigned long date = 0;
void vdate (void ) {
  if (isdate (num) ) {
    date = num;
    ptxt = wd [ num / 86400 % 7 ];
  }
  RLCD;
}
```

Das eingegebene Datum wird von der *val*-Funktion *vdate()* geprüft und gespeichert. Weil mit dem Datum gerechnet werden soll, ist die Eingabe von *kein Datum* nicht erlaubt.

Die Division durch 86400 ergibt die Anzahl der Tage seit 01.01.2000 und die Modulo-Division durch 7 die Nummer des Wochentages. Der Samstag hat die Nummer 0, der

Sonntag 1 usw. Diese Nummernzuordnung kommt dadurch zustande, dass der 01.01.2000 ein Samstag war.

Die Nummer des Wochentages wird als Index beim Zugriff auf das Text-Array `wd[]` verwendet. Das Ergebnis in `ptxt` ist ein Zeiger auf das betreffende Element des Arrays, also auf den Text für den Wochentag.

Einen solchen Zeiger braucht der LCD-Typ `PSTRING`. Der Listenpunkt dieses Typs, hier *Wochentag*, zeigt den Text an, dessen Zeiger jeweils in `ptxt` gespeichert ist.

Das bedeutet allgemein, dass mit `PSTRING` in der zweiten Zeile des LCD ein neuer Text einfach dadurch angezeigt werden kann, dass der betreffenden Zeigervariablen ein neuer Zeiger zugewiesen wird.

Was bisher nicht vorgekommen ist

Während die LCD-Typen für vorzeichenlose Ganzzahlen `ULONG`, `UINT` und `UCHAR` zum Teil mehrmals vorgekommen sind, hat sich für die korrespondierenden LCD-Typen `LONG`, `INT` und `SCHAR` kein Anwendungsfall ergeben.

Während bei den vorzeichenlosen Ganzzahlen die Stellenzahl durch Editieren des Leerzeichens vergrößert werden kann, geschieht das bei den Typen mit Vorzeichen durch Editieren von '+' oder '-'.

LCD-Typ `FLOAT...`

Manche Probleme können nicht mit den LCD-Typen `DEZ..` (Pseudo-Dezimalzahlen) gelöst werden, so dass man Variablen des C-Typs `float` braucht, die auch verarbeitet, angezeigt und editiert werden müssen. Es gibt daher den LCD-Typ `FLOAT..`, der für die Aus- und Eingabe von Zahlentexten die Funktionen `sprintf()` und `sscanf()` der Standardbibliothek `stdio.h` verwendet.

Der Gebrauch von `float`-Variablen wirkt sich deutlich auf die beschränkten Ressourcen eines MC aus. Das sieht wohl auch der Compiler so, denn im `makefile` muss die Verwendung der Funktionen `printf()` und `scanf()` angemeldet werden. Bei der `mt`-Anwenderkommunikation ist es nicht anders: Wenn in die LCD-Anzeigeliste der Typ `FLOAT..` aufgenommen werden soll, muss in `convert.h` das Symbol `FL` definiert werden.

Das Format der Zahlenein- und -ausgabe muss in einem Format-String beschrieben werden. In `convert.h` können drei Format-Strings konfiguriert werden, die den drei LCD-Typen `FLOATA`, `FLOATB` und `FLOATC` zugeordnet sind. Die Hilfen dazu sind in `convert.h` zu finden.

Zu beachten ist, dass die LCD-Typen `FLOAT..` den `val`-Funktionen die editierte Zahl in der globalen Variablen `fnum` zur Verfügung stellen.

Es besteht die Gefahr, dass die Funktion `sprintf()` über das Ende des Textpuffers hinaus schreibt, wenn eine große `float`-Zahl durch Berechnung entsteht und sie dezimal dargestellt wird. Daher muss in diesen Fällen zur Sicherheit die Exponentialdarstellung gewählt werden.

LCD-Typ CHARNUM

Dieser Typ ist für einen speziellen Zweck geschaffen worden. Er präsentiert den C-Typ *unsigned long* als Zahl mit konfigurierbarer Zahlenbasis und konfigurierbaren Ziffern. Die Konfiguration gilt für alle Punkte der LCD-Liste, die diesen Typ verwenden.

Mit diesem Typ ist es ohne weiteres möglich, eine Zahl im Dualsystem, im Oktalsystem, im Zehnersystem oder im (Hexa-)Sedezimalsystem darzustellen, um die gebräuchlichen zu nennen. Aber dafür ist der Typ CHARNUM nicht gedacht.

Als Zahl spielt der Typ CHARNUM nur insofern eine Rolle, als dass er in 4 Byte, also mit minimalem Speicherplatzbedarf, gespeichert werden kann. Eingesetzt wird er, um mehrere Einstellungen vorzunehmen zu können, ohne den Listenpunkt zu wechseln. Es gibt so viele Einstellungen wie es Stellen gibt und so viele Einstellmöglichkeiten wie es Ziffern gibt, vergleichbar mit Einstellungen durch „Daumenräder“.

Beispiel: Für jeden Tag einer Woche gibt es 5 Möglichkeiten etwas einzustellen. Für den Typ CHARNUM sind dann mindestens 5 mnemonische Zeichen als Ziffern und die Zahlenbasis 5 zu konfigurieren.

Task LCD behandelt den Typ CHARNUM wie jeden anderen vorzeichenlosen Zahlentyp, bis auf eine Besonderheit: Man kann in `convert.h` `CHARNUMFIELD > 0` definieren, um eine Anzeige mit entsprechender Feldbreite zu erhalten, bei der es dann kein Einfügen von führenden Nullen gibt.

CHARNUMFIELD muss für das vorgenannte Beispiel 7 betragen.

Sofortanzeige

Ein Reaktionstester, wie in Teil 3, ist mit der Aktualisierung der Anzeige von Task LCD im Zyklus von ca. 500 ms nicht zu machen, da „Push“ möglichst exakt mit Beginn der Zeitmessung angezeigt werden muss.

Task LCD bietet für diesen Fall an, die Semaphore LCDSEM zu definieren. Mit `signal (LCDSEM)` kann dann eine Task den aktuellen Zyklus abbrechen und die sofortige Aktualisierung der Anzeige erzwingen.

Zum Schluss

Ich möchte mit Teil 4 die Artikelserie beenden. Das schließt nicht aus, dass weitere Artikel entstehen, die zwar Kenntnisse des Systems voraussetzen, aber sonst mit der Serie nicht weiter verbunden sind.

Die DCF77 Uhr und die Portierung auch der Anwenderkommunikation auf den R8C/13 von Renesas werden weitere Themen von Artikeln sein.

