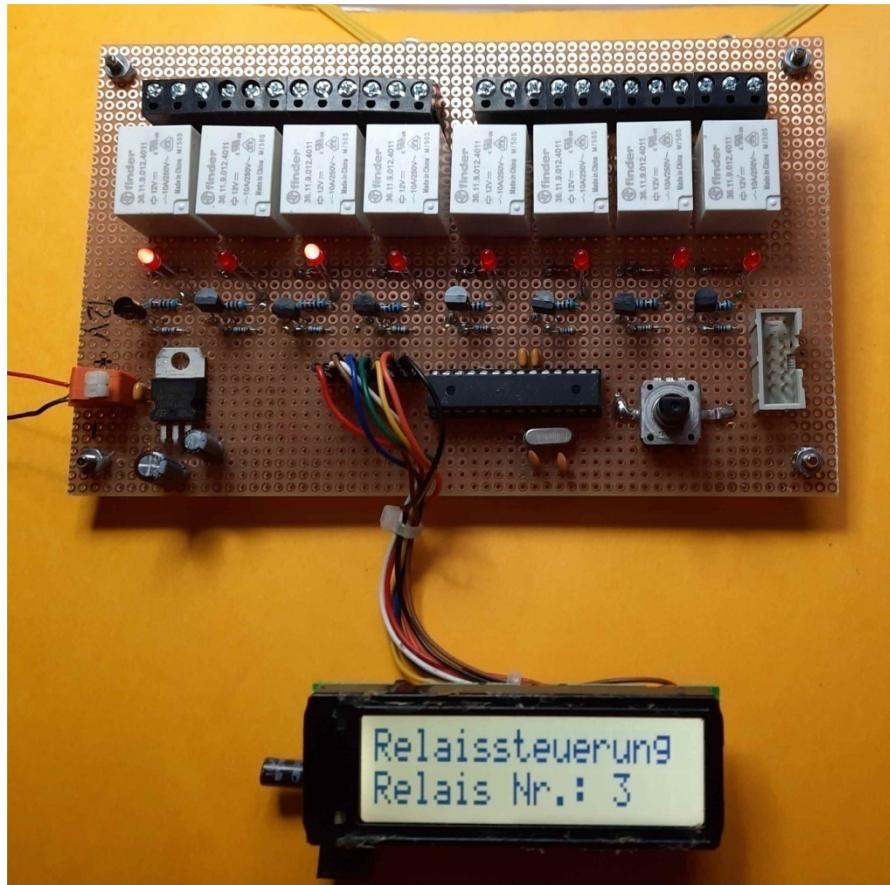


Einführung in die Programmierung von Mikrocontrollern

Joachim Kraatz

Berlin im Dezember 2022



**Eine Erklärung mit Hilfe von einfachen, leicht
verständlichen Projekten. Es wird ein preiswerter
Mikrocontroller der Firma Microchip verwendet**

Inhaltsverzeichnis

Einleitung	1
Entwicklungsumgebung installieren	2
Projekt 1: LED einschalten	3
Einführung in die Programmiersprache C	5
Projekt 2: LED blinkt	7
Logische Algebra	8
Projekt 3: LED mit Taste schalten	9
Kontrollanweisungen	10
Interrupt, Datentypen,Funktionen	11
#define, static, volatile, EXOR	13
Projekt 4: Verwenden eines LCD-Displays	14
Projekt 5: Variable am Display ausgeben	17
Projekt 6: Drehencoder verwenden	19
Zustandsautomat und switch-Anweisung	21
Projekt 7: Relaissteuerung	23
Software Relaissteuerung	24
Hardware Relaissteuerung	28
Projekt 8: Netzfrequenz messen	31
Projekt 9: Temperatur messen	37
Projekt 10: Elektrische Leistung messen	49
Projekt 11: Leistung von Hochfrequenz messen	56
Projekt 12: Wetterstation mit Internet-Anbindung und Solarstromversorgung	65
Schlusswort	121
Verzeichnis der Fotos, Stücklisten, Schaltpläne und Programmlistings	122
Links und Quellenangaben	125

Einleitung – es geht auch ohne den Arduino

Ziel soll sein, mit einfachen Mitteln Hard- und Software zu erstellen, das Produkt soll vielseitig und zugleich preiswert sein.

Was verwenden wir? Einen modernen Baustein, genannt Mikrocontroller. Wir haben uns für einen AVR-Mikrocontroller entschieden, ein kleiner Chip, auf dem sich neben einem Mikroprozessor, Speicher, Ein- und Ausgabe noch einige zusätzliche Hardware befindet, so z.B. Zeitgeber (Timer), Analog-Digitalwandler, Schnittstellen u.a. Zum besseren Verständnis ist die Dokumentation nützlich, welche man im Netz bei Firma Microchip (1) findet – oder noch einfacher, beim Bestellen bei Reichelt (2) dem Link auf die Datenblätter auswählen. Neben flüchtigem Speicher enthalten Mikrocontroller in der Regel auch programmierbaren Flash-Speicher, ähnlich einem Memory-Stick.

Was ist unser Ziel? Wir bauen ein Gerät, welches uns ermöglicht, 8 elektrische Verbraucher unabhängig voneinander ein- und auszuschalten. Das Gerät soll möglichst einfach zu bedienen sein.

Was sind die Voraussetzungen? Etwas handwerkliches Geschick, ein wenig Verständnis für Elektrotechnik und Programmieren. Eine Lötstation und ein PC werden vorausgesetzt, des Weiteren eine Stromversorgung von 5 - 12 Volt, 1 Amp. Um die Programme auf den Chip zu flashen, bedarf es eines Programmiergerätes, gut geeignet ist z.B. das preiswerte DIAMEX USB ISP (bei Firma Reichelt, für etwa 25 € erhältlich). Hilfreich ist es, Schaltpläne lesen zu können. Ein kleines Steckboard ist sehr hilfreich, um Schaltungsteile zu testen. Und last but not least: den Mikrocontroller, ein AT Mega 8 im 28 poligen DIL-Gehäuse, da so leichter im Experiment zu verwenden, (bei Reichelt für etwa 4,00 €).

Die genannten Bauteilepreise können sich stets ändern, auch die Verfügbarkeit von elektronischen Komponenten ist nicht immer gewährleistet. Zuweilen kommt es zu hohen Lieferzeiten!

Damit der Mikrocontroller tätig werden kann, braucht er erst einmal Befehle. Wir müssen eine Software schreiben und uns überlegen, was der Mikrocontroller eigentlich machen soll.

Alle Programmbeispiele können auf Github unter <https://github.com/electronicexpert/AVR-Projects> heruntergeladen werden!

Noch eine Anmerkung:

In dieser Einführung wird aus Gründen der besseren Lesbarkeit das generische Maskulinum verwendet. Weibliche und anderweitige Geschlechteridentitäten werden dabei ausdrücklich mitgemeint, soweit es für die Aussage erforderlich ist.

Bevor wir jetzt loslegen, kümmern wir uns zuerst um die Software. Gute Erfahrungen habe ich mit der Entwicklungsumgebung GEANY (3) gemacht; diese freie Software ist schnell zu erlernen und läuft auch sehr flüssig. Bei mir wurde sie unter LINUX Lubuntu 18.04 (4) eingesetzt, auf einem alten ASUS EEE-Notebook, welches für die neueren Windows-Versionen nicht mehr brauchbar war. Aber auch unter Windows kann man arbeiten, ich habe aber keine Erfahrung damit. ATMEL Studio (Microchip) leistet bei der Software Entwicklung ebenfalls gute Dienste, soviel ich weiß, ist die letzte Version Studio 7, meiner Meinung nach sehr leistungsfähig, aber auch auf schnellen Rechnern recht träge.

Hier gehe ich auf die Installation von GEANY ein:

Vor der Installation erstellen wir uns an geeigneter Stelle ein Verzeichnis, welches am besten den Projektnamen erhält, hier z.B. „Ledtest“. Wenn ich mit mehreren unterschiedlichen Mikrocontrollern arbeite, lege ich natürlich mehrere Unterverzeichnisse an; hier meine Beispieldokumentation:

```
/home/joachim/programmieren/avrs/mega8/ledtest  
    /irgendwas1  
    /irgendwas2  
    /mega1284p/irgendwas1  
        /irgendwas2
```

Ich habe mir angewöhnt, in das leere Verzeichnis gleich das sogenannte „makefile“ hineinzukopieren, dann vergisst man es nicht.

Bestimmt finden wir im Netz weitere geeignete Standardvorlagen für ein Makefile (5).

Mit dem Befehl: *sudo apt-get install geany*

wird die Software in Ubuntu installiert.

Sollte das nicht funktionieren, ist es ratsam, das Ubuntu auf Aktualisierungen zu prüfen, das macht man mit: *sudo apt-get update* und danach *sudo apt-get upgrade*.

Dann startet man GEANY, man findet es in den Applikationen, führt aus: neues Projekt erstellen, und bindet dort (Dateiname und Basisverzeichnis) das neu erstellte Verzeichnis mit ein, nachdem man vorher einen Projektnamen angelegt hat. Das ist nicht ganz eindeutig, man klickt an der Seite das Ordnersymbol an, bis man im richtigen Verzeichnis ist (nicht makefile anklicken!) und dann rechts unten auf Öffnen. Man landet dann wieder in der vorherigen Eingabeaufforderung. Nach 2 maliger Auswahl des richtigen Verzeichnisses wird Erstellen (unten rechts) angeklickt.

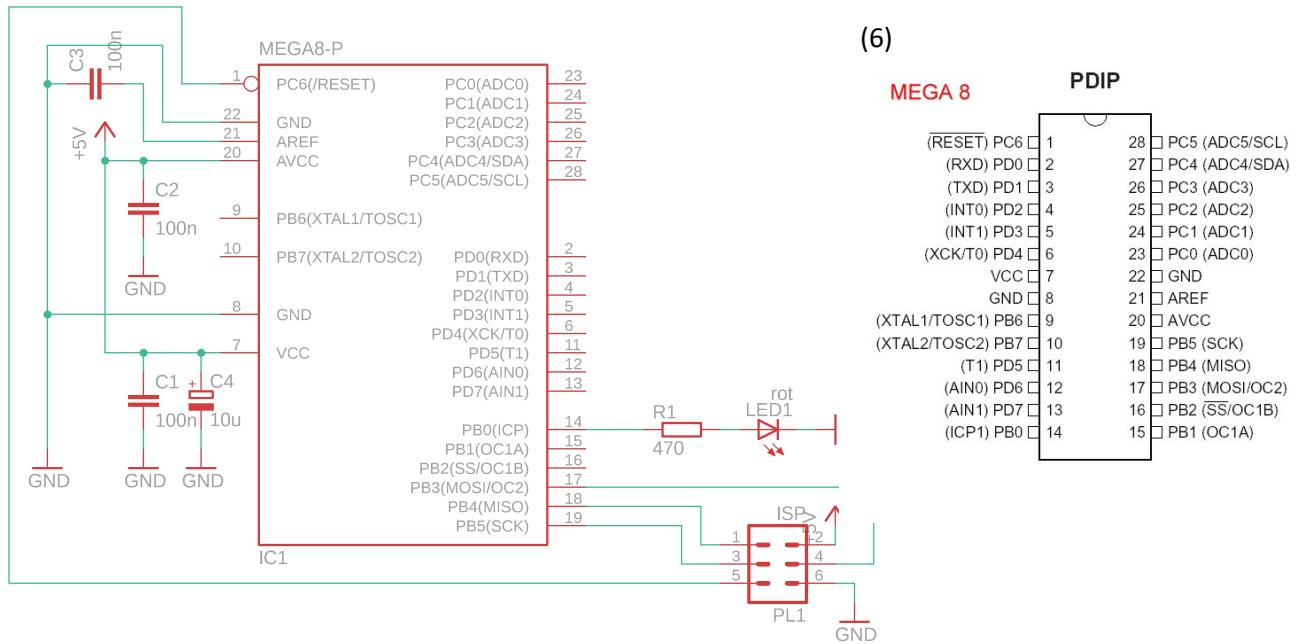
Projekt 1: LED einschalten

Benötigt wird ein Steckboard, etwas Schaltdraht 0,14 mm², gut aus Telefonverlegekabeln zu gewinnen, ein AT Mega 8 (DIL, nicht SMD), ein Elko 10 Mikrofarad(uF) / 25 V, 3 keramische Kondensatoren 0,1 uF, ein Widerstand 470 Ohm, ¼ W, 1 LED 3mm rot, 1 Wannenstecker 6 polig (siehe Bestell-Liste Firma Reichelt).

Stückliste für erstes Experiment: wir rechnen mit etwa 32 Euro (ohne Netzgerät)

Anz	im Plan	Wert	Reichelt Best-Nr	Bemerkungen
1	--	ISP Programmer	DIAMEX USB ISP	
1	-	Steckboard	STECKBOARD 1K2V	
1	-	Netzgerät	PEAKTECH 6080 A	Vorhanden?
1	LED1	LED 3mm rot	LED 3-2000 RT	
1	R1	Widerstand 470 Ohm	METALL 470	
3	C1-3	Kerko 100n	Z5U-2,5 100N	
1	C4	Elko 10 uF	M 10U 25	
1	IC1	Mikrocontroller	ATMEGA 8A-PU	
1	PL1	Wannenstecker	WSL 6G	

Wir beschalten das Ganze so:



Das DIAMEX wird an PL1 angeschlossen, der Pluspol der Stromversorgung an +5V, der Minuspol an GND. Hinweis: Alle GND sind miteinander zu verbinden. +5V geht auch an Pin 2 von PL1. C1 und C2 sind am gleichen Potential angeschlossen, sind jedoch wegen besserer Störunterdrückung möglichst nahe den Anschlussbeinchen (7 bzw. 20) des Mikrocontrollers anzubringen. C3 ist lediglich ein Glättkondensator für die Referenzspannung der im Mikrocontroller enthaltenden Analog zu Digitalwandlern (die wir hier nicht verwenden - aber man macht das so wegen „Good Practice“). Wir müssen unbedingt darauf achten, dass die Spannungsversorgung polrichtig anliegt, im Bereich von 3,3 bis höchstens 5V liegt. Günstig erweist es sich, wenn das Netzgerät eine einstellbare Strombegrenzung hat, diese ist dann auf 0,1 Ampere einzuregeln.

Einführung in die Programmiersprache C

Wenn die Hardware getan ist, setzen wir uns sogleich an die Erstellung der Software. Ich bevorzuge hier die Programmiersprache C (7) . In kurzer Zeit kommt man damit klar, was die „Basics“ angeht. Natürlich ist C sehr komplex, um auch die letzten Leistungsreserven da herauszuziehen, muss man schon viel Zeit opfern um sich das nötige Wissen anzueignen, man sollte dann schon viel können.

Beginnen wir mit dem Einfachen! Jedes C Programm beginnt **immer** mit der Funktion **main**.

Davor können natürlich andere Anweisungen stehen. Wir halten den ersten Programmierversuch jedoch so einfach, wie möglich und verzichten auf unnötigen Ballast. Wenn man sich da mehr interessiert, bieten sich einschlägige Kurse, Literatur, Recherchen im WWW an.

Software Teil 1:

Wir möchten die LED an Port B0 (Pin 14) des Mikrocontrollers aufleuchten lassen. Dazu müssen wir zuerst einmal per Software dem Mikrocontroller mitteilen, dass Pin B0 von Port B ein Ausgang ist, denn nach dem Einschalten des Stromes für den Baustein sind alle Ports auf Eingang geschaltet, das macht man aus Sicherheitsgründen. Desweiteren müssen wir den Port B0 auf logisch Eins setzen, er ist nach dem Einschalten zurückgesetzt. Wir müssen 2 Anweisungen verwenden, wie hier gezeigt:

```
#include <avr/io.h>

int main(void)    // unvermeidbares main - erst einmal ohne weitere Fragen so notieren
{
    DDRB |= (1 << PB0); // das Datenrichtungsregister von Port B wird f. PB0 Ausgang
    PORTB |= (1 << PB0); // Das Port B0 wird auf Eins gesetzt
    while(1);           // man lässt „höflicherweise“ das Programm geordnet terminieren
}
```

#include <> oder #include " " bedeuten Einbinden einer externen Funktionsdeklaration. C bietet nur wenige Schlüsselbegriffe, in sogenannten Libraries sind entsprechende weitere enthalten. In unserem Beispiel verwenden wir die Begriffe DDRB und PORTB, diese sind in avr/io.h deklariert. Vergisst man das Includen,(oder includiert falsch) , entstehen Fehlermeldungen. Der Unterschied von <> zu " " bedeutet in ersten Fall, dass die passende Headerdatei (.h) im Standard-Verzeichnis des Compilers steht, der zweite Fall bedeutet, dass die .h-Datei an einer, von uns angegebener Stelle steht. Übrigens werden die #include-Anweisungen, sowie die Kommentare // , sowie ein paar später zu erklärende Anweisungen vom sogenannten Präprozessor ausgewertet und nicht direkt kompiliert. Die Aufgabe des Compilers ist schließlich die Erzeugung von Assembler-Code, welcher anschließend in den geeigneten Maschinencode des Mikrocontrollers umgewandelt wird. Darum müssen wir uns nicht kümmern, das erfolgt im makefile.

// bedeutet in der Sprache C die Einleitung einer Kommentarzeile, dort kann jeder beliebige Text stehen.

{ und } bedeuten Anfang und Ende eines Blockes. Grob gesagt, müssen beide Klammerarten in gleicher Anzahl vorkommen.

<< bedeutet hier, dass die „1“ um so viele Stellen nach links geschoben wird, wie PB Einsen von rechts nach links stehen hat, im Falle PB0 also um keine Stelle, die „1“ landet somit direkt ganz rechts. Das geht so, weil C immer von 0 aus zählt.

DDRB |= ist eine Zusammenfassung von DDRB = DDRB | (im Klartext: Der neue Inhalt von DDRB ist der momentane Inhalt von DDRB, verodert mit dem Ausdruck auf der rechten Seite). Was hier UND, ODER, NICHT bedeutet, sollte bekannt sein! Man kann auch vereinfacht schreiben: DDRB = 1. Das ist aber „schlechter Stil, denn wir zerstören möglicherweise Informationen, die bereits im Register DDRB enthalten sind und suchen später stundenlang nach Fehlern!“

An anderen Stellen wird der Text vom Compiler auf Plausibilität untersucht, es können dann Fehler auftreten, wenn dort falsche Anweisungen stehen!

Wie schon oben erwähnt, müssen wir ein geeignetes makefile verwenden. Wir laden also die Schablone (5) herunter und passen diese an unsere Bedürfnisse an. Am Besten öffnen wir das makefile im Editor von GEANY. Wir ändern hier: MCU = atmega8, F_CPU = 1000000, und tragen weiter unten bei „Programming Options“ den richtigen Programmierport ein: AVRDUDE_PROGRAMMER = stk500v2 und AVRDUDE_PORT = /dev/ttyACM0). Die beiden letzten Einträge hängen vom verwendeten Programmiergerät ab und können auch anders aussehen!

Nun erstellen wir in GEANY eine neue Datei, schreiben den Code dort hinein und speichern diese unter „main.c“.

Danach sollte mit „Erstellen - Make“ die Übersetzung des Programmcodes ohne Probleme vorstatten gehen, was wir unten in der Statuszeile sehen.

Wenn hier Fehler auftreten, kann das 3 Gründe haben: entweder vertippt! -> Achtung: es gilt Unterscheidung von Groß- und Kleinschreibung. Oder es fehlt das makefile. Dieses ist sehr komplex, aber man kann eigentlich stets eine Schablone davon verwenden und es an die Anwendung anpassen. Wir werden das später tun. Oder das makefile hat Fehler.

Wir gehen mal davon aus, dass alles ok ist.

Jetzt schalten wir die Hardware ein und laden den gerade erstellten Maschinencode auf den Mikrocontroller hoch. Dazu gehen wir in GEANY auf Erstellen – Build. Jetzt sollte die LED aufleuchten.

Projekt 2: Led blinkt

Die LED soll jetzt blinken. Dafür ist sowohl der C Code umzuschreiben, als auch eine weitere .h-Datei zu includieren. Wir überlegen zuerst einmal, wie wir das Problem angehen. Wir benötigen eine Zeitverzögerung für die „An-Zeit“ der LED, sowie eine Zeitverzögerung für die „Aus-Zeit“. Die LED soll ständig blinken, bis der Strom abgeschaltet wird. Sie soll nach dem Einschalten sofort angehen. Das sind die Randbedingungen.

Software Teil 2:

In C formuliert, sieht es so aus:

```
#include <avr/io.h>          // für PORTB und DDRB
#include <util/delay.h>        // für _delay_ms()

main(void)
{
    DDRB |= (1 << PB0); // wir legen die Initialisierung vor die Schleife, einmal ist
ok
    while(1)                // das geordnete Terminieren ist hier
eingeschlossen
    {
        PORTB |= (1 << PB0); // LED ein, wie schon im Beispiel 1
        _delay_ms(500);       // eine halbe Sekunde warten
        PORTB &= ~(1 << PB0); // LED aus, wird gleich ausführlich erklärt
        _delay_ms(500);       // nochmal 500 ms warten
    }                         // ständig wiederholen ...
}
```

Erklärung Programmanweisungen und logische Algebra

Diesen Programmcode sollte man bereits gut verstehen, die eigenartige Anweisung in der vorletzten Zeile muss aber erklärt werden, aber auch die Anweisung 2 Zeilen darüber, die wir in Beispiel 1 stillschweigend so akzeptiert hatten, beginnen wir also damit. Wir haben verstanden, dass die Anweisung $\text{PORTB} |= (1 << \text{PB0})$ bedeutet: $\text{PORTB} = \text{PORTB} \text{ ODER } 1$. Jetzt sollten wir ein wenig Boolesche Algebra (8) verstehen und uns mit Bits gut auskennen. In der Recheneinheit des Mikrocontrollers werden Daten im Format 8 Bit verarbeitet, das erkennt man schon ganz gut daran, das ein Port, z.B. Port B von PB0 bis PB7 reicht.

Zur Wiederholung:

(I) ODER bedeutet:			(&) UND bedeutet:			(~) Invertieren bedeutet:	
Bit1	Bit2	Ergebnis	Bit1	Bit2	Ergebnis	Bit	Ergebnis
0	0	0	0	0	0	0	1
1	0	1	1	0	0	1	0
0	1	1	0	1	0		
1	1	1	1	1	1		

Wenn wir nun ein Bit eines Ports mit 1 verodern, kommt immer 1 heraus, egal was vorher dort drin stand (siehe 1. Logik-Tabelle) Das Schieben bestimmt hier die Position des zu manipulierenden Bits innerhalb der 8 Bits im Port.

Merke: *ODER* schaltet Bit ein!

Etwas verzwickter wird der 2. Ausdruck: $\text{PORTB} \&= \sim(1 << \text{PB0})$. Aufgelöst bedeutet er: $\text{PORTB} = \text{PORTB} \text{ UND } 11111110$. Wir denken einen Schritt weiter: da wir 8 bit haben, würde eine Null so dargestellt werden: 00000000. Wenn wir diese invertieren, erhalten wir 11111111. Zur Notation: da wir bei der Verarbeitung von Zahlenwerten verschiedene Repräsentationen zulassen, verwenden wir Suffixe, um Verwechslungen zu vermeiden. Eine ganz normale Zahl mit Basis 10, z.B. 255 würde dann mit 255d bezeichnet. Die Bitfolge 11111111 dann mit 11111111b. Daneben gibt es noch Zahlen mit Basis 16, z.B. 2EA4, diese werden als 2EA4x markiert, es werden sonst unvermeidliche Verwechslungen auftreten, man denke sich, die Zahl wäre 2000 und hätte die Basis 16.

Zu unserem Problem zurück: Wenn PORTB nun 1 ist (s.o.) ist seine Inversion ja 11111110. Wir wissen bereits, dass bei einem *UND* immer das Ergebnis Null wird, egal welche Eingangsvariable Null aufweist, es müssen für die Ausgabe einer Eins beide Eingangsvariable Eins sein.

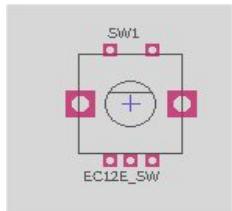
Merke: *UND* schaltet Bit aus.

Damit sollte auch die zweite Anweisung in der vorletzten Zeile verstanden sein. Wir können das Programm somit testen.

Projekt 3: Led mit Taste schalten

Wir benötigen ein zusätzliches Bauteil, es ist bei Reichelt unter der Bezeichnung STEC12E08 erhältlich und kostet um 2,50 €.

Nun werden wir die LED mit Tastendruck einschalten. Hierfür verbinden wir die beiden Pins wie im Bild gezeigt mit dem Mikrocontroller.



SW1 (oben) eine Seite mit PB1 , die andere Seite mit GND verbinden.

Die Software wird nun so ergänzt, dass diese den Zustand der Taste erkennen kann. Da die Taste gegen GND schaltet, also sozusagen beim Drücken einen Kurzschluss zur Masse bildet, müssen wir dafür sorgen, dass der nun fließende Strom in vernünftigen Grenzen bleibt. Normalerweise wird das sichergestellt, indem das Port PB1 über einen Widerstand mit 5 V verbunden wird. Das nennt man Pull-Up. Dann steht am Port die Spannung von 5 V an. Solange die Taste unbetätigt bleibt. Beim Druck auf selbige wird die Spannung am Port 0 V. Das entsprechende Bit, welches dem Port zugeordnet ist, spiegelt diesen Zustand, es kann in der Software abgefragt werden. Wir haben bisher DDR und PORT kennengelernt, nun kommt noch PIN hinzu. Diese Variable gibt den aktuellen Zustand des Port wieder.

Wenn wir einen Eingang nutzen, müssen wir das DDR Register nicht explizit auf 0 setzen, es ist schon nach dem Start des Mikrocontroller in diesem Zustand. Vereinfachend können wir den Widerstand weglassen und den Pull-Up per Software herstellen, dazu müssen wir lediglich nach PORTB eine Eins für das entsprechende Port schreiben, das schaltet, anders als im vorherigen Beispiel den Port nicht auf Eins, denn dieser ist ja jetzt ein Eingang (es wird „innerlich“ auf Eins gesetzt).

Kontrollanweisungen

Die Abfrage geht so: Wenn das Bit von PORT B1 gesetzt ist, führe die Aktion aus. Hierfür brauchen wir eine neue C-Anweisung, nämlich das if. Davor hatten wir bereits eine andere Anweisung genutzt, nämlich das while. Die beiden sind Kontrollanweisungen, es gibt davon einige. Das while bleibt solange in einem Block, bis sein Argument (in der Klammer) Null wird. Daher wird der Block permanent ausgeführt. Solange im Argument eine 1 steht. Das if wird nur ausgeführt, wenn in der Klammer eine 1 steht, sonst wird es übersprungen. Folgt dem if ein else, so wird dieses als Alternative ausgeführt, wenn in der Klammer eine 0 steht.

Software Teil 3:

Der Code ist jetzt recht einfach:

```
if(PINB & ((1 << PB1) == 0))
{
    PORTB |= (1 << PB0);
}
else
{
    PORTB &= ~(1 << PB0);
}
```

Genaue Erklärung: der Ausdruck innerhalb der ersten Klammer um if muss wahr, also 1 sein, somit wird er erste Block ausgeführt und die LED damit eingeschaltet. In der zweiten Klammer steht die Verundung des Ausdruckes $1 << PB1$, ist er wahr (1) dann würde der 1. Block ausgeführt. Aber das wäre unrichtig, da wir die Taste ja noch nicht gedrückt haben und das Port PB1 bereits jetzt 1 ist. Wir müssen die Logik daher umkehren und den Ausdruck für if wahr machen, wenn die Und-Verknüpfung Null ergibt. Die 3 Klammerung um das $1 << PB1$ kann weggelassen werden, erhöht aber die Lesbarkeit des Ausdrucks. Die Klammern um den zu verundenden Ausdruck sind aber Pflicht!

Mit diesem Wissen sollte man in der Lage sein, Software Teil 3 selbstständig zu erstellen. Das #include von delay.h braucht man hier nicht. Falls Fehler beim Übersetzen des Programms auftreten, empfehle ich nochmal alles durcharbeiten und ggf. im WWW nachzuschauen.

Interrupt, Datentypen und Funktionen

Nun werden wir viele neue Fakten kennenlernen, um uns fit für die Relaissteuerung zu machen. Hier zuerst einmal die Ergänzung der Software.

Neu hinzu kommt die Abfrage des Drehencoders und die Ausgabe auf dem 2-zeiligen LCD Display. Hierfür – wie schon erwähnt – gibt es fertige Software-Module (9) und (10). Die Abfrage des Encoders verlangt recht viel Aufwand – die Reaktion auf das Drehen sollte verzögerungsfrei erfolgen. Daher kommen wir nicht umher, die Funktionalität eines Interrupt auszunutzen. Diese ermöglicht, ein laufendes Programm zu unterbrechen, sobald externe Ereignisse auftreten, z.B. Änderungen von Zuständen an Eingangsports oder auch Ablauf von Zeitgebern. Auch gewisse andere Ereignisse der im Mikrocontroller befindlichen Hardware können Interrupts auslösen.

Was die Software für die Auswertung des Drehencoders beinhaltet, ist auf der Internetseite von [Microcontroller.net](#) (9) schön erklärt: es werden hier 3 Module benötigt: die Initialisierung(`encode_init`), der Timer-Interrupt(`ISR(TIMER0_OVF_vect)`) und die Auswertung(`encode_read4`). Die Initialisierung muss im Hauptprogramm (`main`) aufgerufen werden, sie setzt die Variablen `old`, `new` und `enc_delta`. Der Timer-Interrupt fragt zyklisch alle 155 us die Kontakte A und B des Encoders ab. Man kann experimentell dieses Intervall verändern, es hängt davon ab, welche zusätzliche Rechenleistung der Mikrocontroller zu erfüllen hat. Der hier genannte Wert war ein guter Kompromiss, was flüssige Reaktion auf Drehen des Encoders und Rechenleistung betrifft. Letztendlich kehrt das letzte Modul mit dem am Encoder eingestellten Wert zurück, wenn es im Hauptprogramm aufgerufen wird. Entweder mit 1 (rechts gedreht), -1 (links gedreht) oder Null (nicht gedreht). Die Taste am Encoder ist davon unabhängig, sie wird, wie im vorangegangenen Beispiel einfach nur abgefragt.

Um das Ganze jetzt nicht unnötig in die Länge zu ziehen, werde ich hier nur einige wenige Erklärungen zum Programmcode geben.

Datentypen:

„char“ beinhaltet 8 bit mit Vorzeichen (-128 bis 127)
„int“ 16 bit mit Vorzeichen (-32768 bis 32767)
„unsigned int“ auch 16 bit, aber ohne Vorzeichen (0 – 65535).

Es gibt noch viele andere Datentypen und Konstrukte, so z.B. Arrays, das sind Anordnungen von gleichen Datentypen, welche durch Indizes angesprochen werden können, Beispiel: `char argv[]`. Diese Variable besteht aus mehreren Elementen vom Typ `char`. Das Sternchen vor weiner Variablen bedeutet Zeiger, das bedeutet, dass hier nicht der Wert einer Variablen repräsentiert wird, sondern deren Speicheradresse. Das ist jedoch mehr Werkzeug für Fortgeschrittene, es wird hier nicht weiter erklärt.

Funktionen:

Wir kennen bereits main als zentrale Funktion in C. Wir schreiben bekanntlich main(void) {....}. Das ist das absolute Minimum für einen Funktionsaufruf. Es bedeutet: springe in die Funktion main, mache da irgendwas und komme ohne ein Ergebnis zurück. Hört sich erst einmal sinnlos an, hat aber schon einen Zweck, wenn z.B. Werte direkt an die Hardware transportiert - und im Weiteren nicht verarbeitet werden. Das Wort „void“ bedeutet „nichts“ und muss an der genannten Stelle angegeben werden!

Wenn wir aber von der Funktion einen Rückgabewert erwarten, können wir das bei der Deklaration angeben. Hoppla, wieder ein neuer Begriff. In C sollte eine Funktion vor ihrer ersten Verwendung immer deklariert werden, manche Compiler verlangen das, andere nicht. Es hilft aber sehr, Fehler zu vermeiden. Eine Deklaration besteht in der Angabe des Funktionskopfes (mit abschließendem Semikolon – und Definition aller Parameter mit deren Namen inklusive deren Datentyp). Eine Definition bedeutet die Angabe des Funktionskopfes, jetzt aber nur mit Namen – aber ohne die Datentypen der Parameter und anschließenden geschweiften Klammern zur Angabe eines neuen Blockes. Bei der Deklaration muss auch der Rückgabewert mit seinem Datentyp angegeben werden, er bekommt aber keinen Namen.

Die Deklaration von main (ist irgendwo in einer Library abgelegt) kann lauten:

void main(void); aber auch int main(void); In Fall 2 muss main dann auch einen Rückgabewert vom Typ int liefern, welcher mit der Anweisung return an z.B. das Betriebssystem übergeben wird.

Sie kann aber auch lauten: int main(int argc char *argv[]); Das heißt, dass an „main“ mehrere Werte übergeben werden, wir erwarten einen int Wert als Rückgabe. Die beiden Werte für die Übergabe sind nicht zwingend 2 Werte, denn argc ist ein Zähler für die zu übergebenden Argumente in *argv[], hört sich kompliziert an, lässt sich aber gut erklären, wenn man weiß, wie ein Programmaufruf vom Betriebssystem für ein C-Programm aussieht. Wenn man das Programm jetzt nicht auf einem Mikrocontroller ausführt, sondern unter Windows oder Linux, was wir aber hier nicht machen werden, ist es gut, wenn wir beim Übersetzen einen gut erkennbaren Namen für das Programm auswählen, das machen wir im makefile. Würden wir das nicht machen, würde als Ergebnis eine Datei herauskommen, die a.out heißt, damit kann niemand etwas ausführen. Würden wir sie z.B. Schaltprogramm nennen, wäre das günstiger. Wenn wir jetzt im Betriebssystem Schaltprogramm aufrufen, stünde in argc eine Null, in *argv[0] der Programmname, nämlich Schaltprogramm. Würden wir jetzt das Programm dahingehend erweitern, indem wir nach dem Programmname Schaltprogramm noch eine Relaisnummer mitgeben und das Ganze aus dem Betriebssystem starten mit „Schaltprogramm 1“, dann stünde in argc eine 1, in *argv[0] der Programmname und in *argv[1] die 1. Dazu muss man beachten, dass in C die Indizierung immer mit der Zahl 0 beginnt. Ein C-Programm kann somit zur Laufzeit einen Parameter verwenden, welcher beim Programmaufruf mitgegeben wird, hier die 1. Wir wissen, sobald argc nicht mehr 0 ist, wurde ein Parameter mit übergeben, dieser steht dann in *argv[1]. Das kann man auf viele Parameter erweitern. Wir brauchen das alles hier nicht, die Beispiele dienten nur zur Erklärung der verschiedenen Funktionsaufrufe.

Define

Was wir bis jetzt nicht kennen, ist `#define`, `int8_t`, `static`, `volatile`, `^=`, `cli()`, `sei()`, ISR, TIMER0

„`#define`“ ordnet einen Variablenwert einem Zahlenwert (oder auch einem Makro) zu, hier niemals ein Semikolon am Ende setzen, bei fast allen anderen C Anweisungen ist es aber Pflicht.

Bsp.: `#define VORWAERTS 157`

157 kann man sich schlecht merken, im Programm kann man diese Zahl aber durch VORWAERTS ersetzen. Wie man hier auch sehen kann, sind Umlaute in der Sprache C nicht gewünscht.

„`int8_t`“ ist das Gleiche, wie `char`, wird gerne verwendet, da mit anderen Compilern kompatibel.

„`static`“ kann bedeuten, dass eine Variable in einer Funktion erhalten bleibt, normalerweise werden alle Variablen in einer Funktion bei der Rückkehr nach deren Aufruf gelöscht, daher die Übergabe.

„`volatile`“ gibt an, dass sich eine Variable jederzeit ohne vorherigen Zugriff ändern kann, ist notwendig, um Aktionen des Interrupt an das Hauptprogramm weiterzugeben.

„`^=`“ eine weitere logische Verknüpfung, das Exklusiv-ODER.

(`^`)EXKLUSIV- ODER bedeutet:

Bit1	Bit2	Ergebnis
0	0	0
1	0	1
0	1	1
1	1	0

Es hat stark Ähnlichkeit mit dem ODER, entspricht auch dem Oder in unserer Sprache. Es ist gut geeignet, um den Wert eines Bits umzukehren.

„`cli()`“ und „`sei()`“ sind Schalter für die Ausführung von Interrupts. Manchmal möchte man im Programmablauf nicht von Interrupts gestört werden, dann schaltet man diese komplett mit `cli()` aus. Wenn man sie wieder braucht, mit `sei()` ein. Auf jeden Fall ist im Programm ein `sei()` notwendig, wenn Interrupts verwendet werden. Daneben muss für jeden Interrupt ein besonderes Bit zu setzen, damit dieser ausgeführt werden kann. Für den Timer 0, welchen wir hier verwenden, ist dieses das TOIE0-Bit im Register TIMSK. Die Namen der Register und der zuständigen Bits können wir in der ATMEL-Beschreibung im Kapitel Timer nachlesen. Wird das TOIE0 gesetzt, dann wird immer dann ein Interrupt ausgeführt, wenn Timer 0 überläuft, er zählt permanent von 0 bis 255 hoch und wird vom Controller Takt bedient; ein Verteiler kann dazwischengeschaltet werden, in unserem Fall Faktor 8.

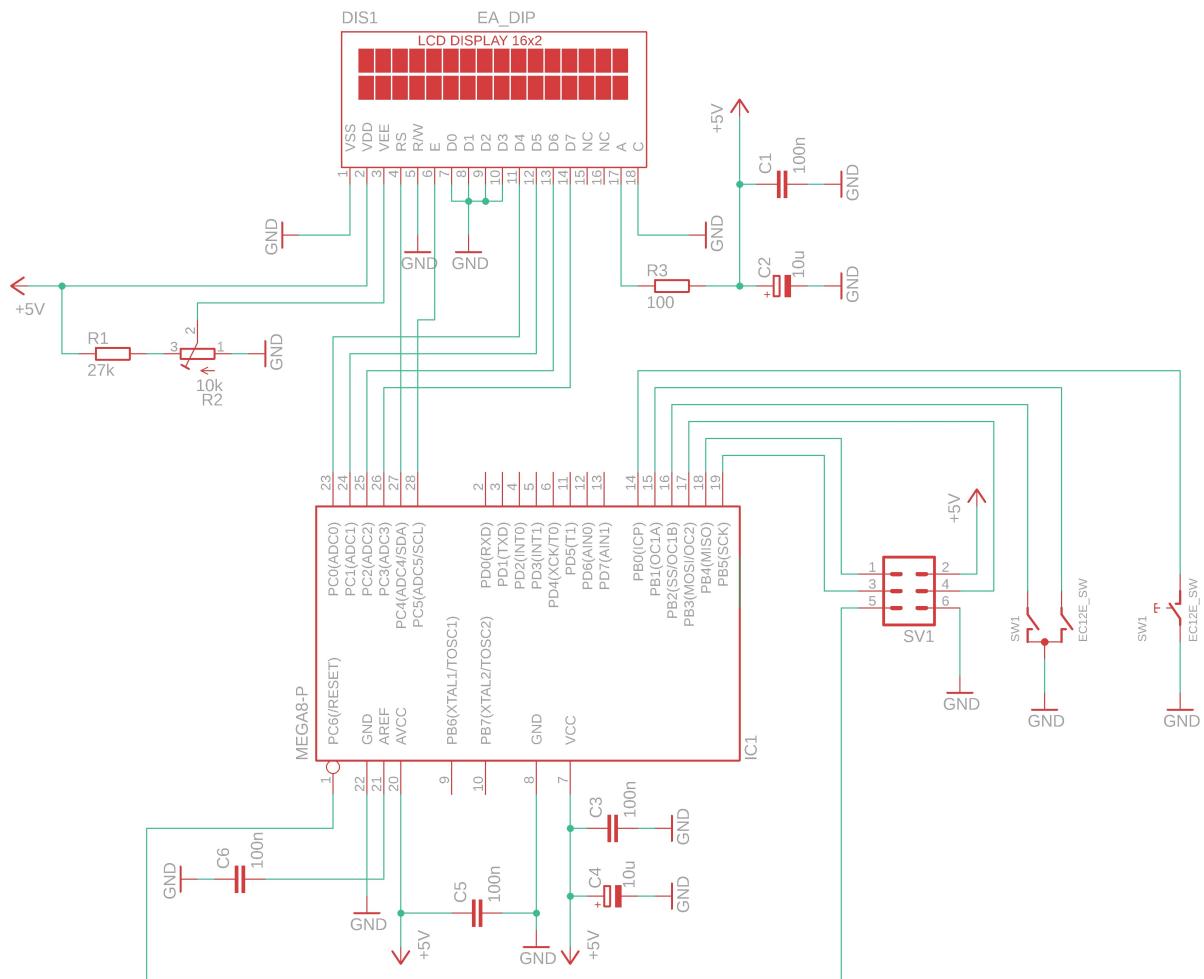
Projekt 4: LCD Display verwenden

Es ist jetzt an der Zeit, einmal wieder etwas Praktisches zu unternehmen. Wir lassen die viele Theorie, welche wir eben durchgegangen sind, erst mal sacken, wir beschäftigen uns mit dem LCD-Display (11). Hierfür benötigen wir eine neue Schaltung und mehr Bauteile. Hier die ergänzte Stückliste:

Ergänzte Stückliste für zweites Experiment: wir rechnen mit etwa 23 Euro

Anz	im Plan	Wert	Reichert Best-Nr	Bemerkungen
1	DIS1	LCD Display 2 Zeilen	LCD 162BL DIP	
1	R2	Trimm-Poti 10k	64Y-10K	
1	R3	Widerstand 100 Ohm	METALL 100	
1	R1	Widerstand 27 k	METALL 27K	
4	C1,3,5,6	Kerko 100n	Z5U-2,5 100N	
2	C2,4	Elko 10 uF	M 10U 25	

Hardware, welche nun schon umfangreicher wird. An dieser Stelle der Schaltplan:



Die Bauteile passen alle auf das Steckbrett, günstig ist es, das Display mit Drähten an die übrige Schaltung anzuschließen.

Wir sollten nach dem Aufbau gleich mal 5 V an die Schaltung anlegen, die Strombegrenzung, wenn vorhanden, bitte auf 100 mA einstellen. Das LCD Display sollte aufleuchten und ggf. in Zeile 1 schwarze Blöckchen zeigen. Mit Hilfe von R2 kann deren Kontrast verändert werden. Klappt das alles, machen wir uns sogleich an die Software.

Software Teil 4:

Auf (9) gehen wir in die beiden Beispielsdateien (lcd-routine.c und lcd-routine.h) und kopieren diese in unser Projektverzeichnis von GEANY. Ganz einfach, wir haben ja noch vielleicht das Projekt von Experiment 1 offen. Dort erstellen wir jeweils eine neue Datei (Datei-> Neu) und fügen den Inhalt der entsprechenden Datei dort ein, speichern die neue Datei unter dem Namen lcd-routine.c bzw lcd-routine.h ab. Jetzt müssen wir im makefile ein paar Änderungen machen bzw. kontrollieren:

```
#Processor Frequency sollte stehen: F_CPU 1000000.  
#List C Source Files here: ergänzen lcd-routines.c  
#Place -D or -U Options here: dort soll stehen: -DF CPU=$(F_CPU)UL
```

In einer anderen Version eines makefiles hatte ich in den Zeilen

```
#Compiler Options: CFLAGS += -DF_CPU=$(F_CPU)  
# Assembler Options: ASFLAGS += -DF_CPU=$(F_CPU)  
zu ändern, es stand dort DF_OSC=$(F_OSC).
```

Alle Einträge in das makefile müssen mit großer Sorgfalt vorgenommen werden, ansonsten treten viele unerklärbare Fehler auf! Sicherheitshalber immer eine Kopie eines intakten makefiles bereithalten.

Nun müssen wir auch noch die Datei lcd-routine.h anpassen:

```
#ifndef F_CPU  
#define F_CPU 1000000 (8000000 steht da normalerweise)  
#endif  
// LCD DB4-DB7 <-> PORTC Bit PC0-PC3  
#define LCD_PORT PORTC  
#define LCD_DDR DDRC  
#define LCD_DB PC0  
  
// LCD RS <-> PORTC Bit PC4  
#define LCD_RS PC4  
// LCD EN <-> PORTC Bit PC5  
#define LCD_EN PC5
```

Oben sind es nur Kommentare, aber es ist ein guter Stil, auch diese aktuell zu halten.

Wir nehmen nun das Programm aus „Software Teil 3“ und ergänzen dieses. Oben kommt die Anweisung: #include „lcd-routine.h“

In den main-Block schreiben wir unter die erste geschweifte Klammer die Anweisung:

```
lcd_init();
```

Das Ganze wird übersetzt und in das Hardware-System geladen. Wenn alles funktioniert hat, sind jetzt zumindest die schwarzen Balken in der 1. Zeile verschwunden, je nach Einstellung von R2. Ist dieses auf vollen Kontrast gedreht, können jedoch 2 Zeilen mit schwarzen Blöcken auftreten.

Jetzt können wir bereits mit dem Display spielen. Wir geben jetzt in der ersten Zeile einen Text aus, in der zweiten einen anderen. Dafür benutzen wir die Anweisungen aus `Lcd-routine`: `lcd_setcursor(spalte,zeile);` und `lcd_string(„Text ..“);`

In unserem Fall: `lcd_setcursor(0,1); lcd_string(„Hello, World“);
lcd_setcursor(0,2); lcd_string(„C macht Freude“);`

2 Anmerkungen: in C dürfen wir alles in eine Zeile schreiben, hauptsächlich, wir setzen am Ende einer Anweisung das Semikolon. Sieht aber nicht schön aus! Besser ist es, wenn wir jeden neuen Block etwa 4 – 8 Spalten einrücken, sodass die Programmstruktur erkennbar wird. `Lcd_setcursor` beginnt die Spalte bei Null, die Zeile aber bei 1. Das Display fasst nur 16 Zeichen, alle darüber verschwinden irgendwo. Man kann aber den Text verschieben (für Fortgeschrittene!). Die oben stehenden Anweisungen setzen wir vor die `while-Schleife`, übersetzen und sehen den Text auf dem Display.

Jetzt können wir uns um den Drehencoder kümmern. Aber bevor wir ihn einlesen, machen wir uns mit der Ausgabe von Variablen auf dem Display bekannt. Bisher konnten wir nur Text ausgeben. Wenn wir aber den Drehgeber einlesen werden, wollen wir auf dem LCD Display eine Art Zähler darstellen, welcher entweder aufwärts oder abwärts zählt – je nach Drehrichtung des Encoders. Wie wir eine Variable ausgeben können, lernen wir nun. Wir gehen davon aus, dass wir eine Variable vom Typ `integer` haben, welche sich jede Sekunde um den Wert Eins erhöht. Dazu setzen wir unser bisheriges Wissen ein und verwenden die Funktion `_delay_ms`. Wir schreiben eine Endlos- Schleife:

Projekt 5: Variable auf LCD Display ausgeben

```
main()
{
    int zaehler = 0;
    while(1)
    {
        _delay_ms(1000);
        zaehler++;
    }
}
```

Der Zähler wird jede Sekunde um 1 hochzählen, nach etwa 9 Stunden wird er negativ werden, nur haben wir nichts davon, weil wir das nicht sehen! Daher werden wir jetzt den Zähler auf das Display bringen. Dafür brauchen wir eine neue Variable, eine neue Funktion und eine neue Include-Datei, nämlich stdio.h

Die neue Funktion ist `int sprintf(char * str, constant char * format, ...)`. Sie ist leicht zu erklären: `char * str` ist ein Puffer mit Elementen vom Typ `char`, den wir gleich anlegen, `constant char * format` ist eine Formatierungsanweisung, dazu mehr im WWW. Schließlich folgen ein oder mehrere Variable, die so umgewandelt werden, dass sie im Puffer als Zeichenkette auftauchen werden. Dazu dient die Formatierungsanweisung. Übrigens kann dort auch Text stehen, der mit ausgegeben wird. Wir werden gleich ein Beispiel sehen.
Zuerst legen wir eine neue Variable an, es ist ein Array mit 16 Elementen vom Typ `char`:
`char zpu[16]; // diese bitte in „main“ oben platzieren.`

Software Teil 5:

Jetzt ergänzen wir unser kleines Testprogramm wie folgt:

```
#include <stdio.h>

main()
{
    int zaehler = 0;
    char zpu[16];
    while(1)
    {
        _delay_ms(1000);
        zaehler++;
        sprintf(zpu,"Zaehler: %5d",zaehler);
        lcd_string(zpu);
    }
}
```

Wir testen das und verstehen die Anweisungen. In der Formatierungsanweisung steht links der Text, der im Display mit ausgegeben wird, dann kommt das Prozentzeichen für die erste Variable (wir haben hier nur eine), die Stellenzahl, welche erwartet wird (5 Stellen) und die Zahlendarstellung, hier dezimal. Schließlich folgt die Variable. Darunter wird der gesamte Puffer an das Display gesendet. Wir müssen aufpassen, dass die Zeile nicht länger als 16 Zeichen wird, denn es würden 2 Unannehmlichkeiten resultieren: erstens würde der Text beschnitten, zweitens würde der Puffer zpu überlaufen, das kann zu schwer zu findenden Fehlern führen. Leider prüft der Compiler in C nicht, ob ein Überlauf vorliegt, andere Programmiersprachen sind da penibler.

Zaehler: 12345 sind 14 Zeichen (mit dem Leerzeichen), das reicht.

Aufpassen, wenn die Zahl negativ wird, dann kommt auch noch das Minus-Zeichen hinzu!

Ein weiterer Tip: wenn wir kürzere Zeilen ausgeben, bleibt immer der Rest der längeren, vorherigen Zeile im Display stehen. Entweder löschen wir das komplette Display mit lcd_clear(), was aber den Nachteil mit sich bringt, dass beide Zeilen gelöscht werden und das Display dabei lästig flackert. Besser ist es, die betreffende Zeile mit 16 Leerzeichen zu beschreiben.

Wenn man viel programmiert hat, wird man viele Tricks kennenlernen. Ich verweise gerne darauf, sich so oft wie möglich Programmcode von anderen Programmierern anzusehen, da bekommt man sehr gute Anregungen. Beispielsweise könnte man die Zeile im Beispiel auch so schreiben:

```
sprintf(zpu,"Zaehler: %5d",zaehler++);
```

die Zeile darüber könnte somit entfallen. Die Zeile würde ausgegeben werden, erst dann würde der Zähler erhöht werden.

Würde man aber schreiben:

```
sprintf(zpu, "Zaehler: %5d", ++zaehler);
```

dann würde zuerst der Zähler erhöht und dann die Anzeige ausgegeben werden.

Projekt 6: Drehencoder verwenden

Nun ist der Drehencoder an der Reihe. Wir öffnen (9) und schauen uns den Beispielcode von Peter Dannegger an. Wir können direkt encode_init, die ISR und encode read übernehmen. Die Phasen A und B liegen bei uns allerdings auf Port B PB1 und PB2, das muss somit geändert werden!

Software Teil 6:

Ich habe hier das Programm von Herrn Dannegger leicht abgeändert und zeige es hier:

Ganz oben landet die Port-Initialisierung und 2 Variable:

```
#define PHASE_A  PINB & ((1 << PB2))
#define PHASE_B  PINB & ((1 << PB1))
#define TASTE (PINB & (1 << PB0))

volatile int8_t enc_delta;
static int8_t last;
```

bei den Phasen habe ich eine Vereinfachung gemacht und die UND-Verknüpfung gleich in die Definition mit hineingenommen.

Wir dürfen auch nicht vergessen, die Headerdatei /avr/interrupt.h zu includieren. Falls wir viele Headerdateien verwenden, bietet es sich an, diese in einer eigens erstellten Headerdatei namens main.h abzuspeichern und nur diese in main einzubinden.

Das ist vor main zu kopieren:

```
void encode_init( void )
{
    int8_t new;

    new = 0;
    if( PHASE_A )
        new = 3;
    if( PHASE_B )
        new ^= 1;                      // convert gray to binary
    last = new;                      // power on state
    enc_delta = 0;
}
```

```

ISR( TIMER0_OVF_vect )           // 1ms for manual movement
{
    int8_t new, diff;
    new = 0;
    if( PHASE_A )
        new = 3;
    if( PHASE_B )
        new ^= 1;           // convert gray to binary
    diff = last - new;          // difference last - new
    if( diff & 1 ){            // bit 0 = value (1)
        last = new;            // store new as next last
        enc_delta += (diff & 2) - 1; // bit 1 = direction (+/-)

    }

    TCNT0 = 0x64;             // reload Timer0 Counter with 0x64
}

int8_t encode_read4( void )      // read four step encoders
{
    int8_t val;

    cli();
    val = enc_delta;
    enc_delta = val & 3;
    sei();
    return val >> 2;
}

```

In die main-Schleife kommt die folgende Initialisierung hinein:

```

int8_t val = 0;

PORTB |= (1 << PB0) | (1 << PB1) | (1 << PB2); // Pullup fuer PB0 bis PB3
aktivieren(Phasen + Taste)

// Timer 0

TCCR0  = (1<<CS00);           // TCCR0 = 0x01 = Eingangsteiler : 1;
TIMSK  = (1<<TOIE0);

```

Dann wird main ergänzt, sodass der Drehencoder gelesen werden kann:

```

encode_init();

sei(); // Enable Interrupt

```

in der while-Schleife kann man jetzt ganz bequem den Drehgeber auslesen:

```

val += encode_read4();

```

Dem Leser bleibt die Aufgabe, den Drehgeber auszulesen und dessen Zahlenwert auf dem LCD Display anzuzeigen.

Zustandsautomat und Switch-Anweisung

Bevor wir jetzt die komplette Relaissteuerung programmieren, beschäftigen wir uns mit der Theorie der sogenannten Zustandsautomaten, das jedoch nur ansatzweise.

Wenn man umfangreiche Programme schreibt, kommt man recht schnell mit den Kontrollanweisungen durcheinander. Ein besserer Weg ist gegeben, wenn man den Programmablauf in Schritten kontrolliert steuert, das wird als Zustandsautomat oder State Machine bezeichnet.

Um das noch besser verstehen zu können, lernen wir eine neue Anweisung kennen, die Switch-Anweisung. Man kann diese als längere Kette von verschachtelten if-else ansehen, selbstverständlich kann man das Switch damit ersetzen, das ist dann aber leider recht unübersichtlich.

Software Teil 7:

Die Anweisung besteht aus mehreren Teilen:

```
switch(Argument)
{
    case A:
        do something;
        break;
    case B:
        do something;
        break;
    case C:
        do something;
        break;
    .
    .
    .
    default:
        do something;
        break;
}
```

Argument darf nur eine Zahl sein, daher auch A bis C. Das case wird ausgeführt, wenn das Argument gleich dem Wert von case ist.

Das break ist sehr wichtig, fehlt es, würde gleich die nächste Anweisung ausgeführt.

Es können beliebig viele cases verwendet werden.

Das default wird immer dann ausgeführt, wenn es keine passenden cases gibt.

Ein einfacher Zustandsautomat kann man entweder auf dem Papier als Zeichnung hinterlegen – oder das mit Worten tun.

Nach dem Einschalten des Stroms wird auf dem Display eine Einschaltmeldung gezeigt (Zustand 1) Wird eine Taste gedrückt, dann springt das Programm in eine Folgeroutine (Zustand 2), in dieser eine neue Meldung auf dem Display erscheint und dort der Drehencoder gelesen werden kann. Drückt man auf den Knopf des Encoders, springt das

Programm wieder in den Zustand 1 zurück. Wir wollen das später benutzen und nach dem Einschalten zuerst mit dem Drehencoder ein Relais auswählen (1 bis 8), dann per Knopfdruck in eine neue Routine springen (Zustand 2), in dieser erneut mit Betätigung des Drehencoders 2 Schaltvarianten für das gewählte Relais ausgewählt werden können (Ein/Aus), sowie 2 weitere Varianten für alle Relais (Alle ein/ Alle aus). Per Knopfdruck wird die Auswahl gültig und es erfolgt ein Rücksprung nach Zustand 1.

In main definieren wir:

```
#define AUSWAHL_REL 0
#define AUSWAHL_EAZ 1
int state = 0;
```

Wir betten das Switch daher in eine Endlosschleife ein:

```
while(1)
{
    switch(state)
    {
        case A:
            Programmanweisungen
            .....
            state = AUSWAHL_EAZ;
            break;
        case B:
            Programmanweisungen
            .....
            state = AUSWAHL_REL;
            break;
        default:
            break;
    }
}
```

Projekt 7: Relaissteuerung

Das „Grundgerüst“: 8 Relais werden über die IO-Ports eines Mikrocontrollers gesteuert, das bedeutet, dass deren Spulen wahlweise mit Hilfe von einem zusätzlichen Transistor angesteuert werden; das ist notwendig, weil die Relais eine Spulenspannung von 12 Volt aufweisen und etwa 40 mA benötigen. Diese Leistung kann der Mikrocontroller nicht unmittelbar bereitstellen. Auf die weitere Beschaltung der Relais gehe ich später noch ein.

Die Relais werden erst durch Aktionen „von außen“ zum Schalten veranlasst. Diese Aktionen können unterschiedlich sein, z.B. Druck auf eine Taste, Betätigen eines Schalters, Ansprechen einer Datenadresse usw. Die ersten benannten Situationen könnten auch ganz ohne Mikrocontroller bewältigt werden. Ganz bewusst habe ich mich entschieden, keine 8 Schalter im Projekt einzusetzen, vielmehr beschränke ich mich auf ein einziges Bedienelement! Zur Visualisierung der Schaltzustände wird ein 2-zeiliges LCD-Display eingesetzt. Das Bedienelement ist ein sogenannter Drehencoder, der zusätzlich einen Taster beinhaltet. So lässt sich die Bedienung einfach halten, sie läuft wie folgt: durch Drehen nach rechts werden die Relaisnummern von 1 bis 8 hoch-, durch Drehen nah links heruntergezählt. Das Zählen erfolgt in einem Ring, das bedeutet, wenn die Grenze von 8 überschritten wird, beginnt die Zählung wieder bei 1; wird die 1 unterschritten, erfolgt die Weiterzählung wieder bei 8. Ist nun das passende Relais ausgewählt, dann erfolgt durch Druck auf die Taste (Achse) des Drehencoders der Sprung in eine weitere Auswahlroutine, hier kann für das vorher gewählte Relais der Schaltzustand ausgewählt werden, dieser kann Ein (E) oder Aus(A) sein. 2 weitere Zustände dienen hier der „Komfortsteigerung“: unabhängig vom gewählten Relais betreffen diese alle 8 Relais. Mit Gemeinsam (C) werden alle Relais gleichzeitig aktiviert, mit Reset (R) alle abgeschaltet. Durch Druck auf die Taste wird die getroffene Auswahl unmittelbar an die entsprechenden Relais weitergegeben. Auch diese Auswahl erfolgt in einem Ring.

Mit diesem Wissen können wir das endgültige Programm verfassen.

Ich verwende hier fertige Libraries von (9) und gehe daher nur am Rand auf die Funktionalität des Displays ein. Genauso verfare ich mit den Routinen für den Drehgeber (10).

Software Teil 8:

Es gibt viele Möglichkeiten, das Programm zu formulieren. Eine kommentierte Version stelle ich hier vor:

```
// nicht vergessen, „lcd_routines.c“ in das makefile mit aufzunehmen!

// Steuerung 8 - Kanal-Relaisplatine
// JK 17.11.21

#include "main.h" // hier sind folgende Header-Dateien enthalten:
                  <avr/io.h>, <stdio.h>, <avr/interrupt.h>,
"lcd_routines.h"

// Portbelegung:

// PB3 MOSI Programmer
// PB4 MISO Programmer
// PB5 SCK  Programmer
// PC6 RESET Programmer

// PD0 - PD7 Relaisport
// PB0 Taster Drehencoder
// PB1 Phase A Drehencoder
// PB2 Phase B Drehencoder

// PC0 - PC3 LCD Data
// PC4   LCD  RS
// PC5   LCD  EN

#define PHASE_A PINB & ((1 << PB2)) // elegante Art, einzelne Bits zu definieren
#define PHASE_B PINB & ((1 << PB1))
#define TASTE (PINB & (1 << PB0))

volatile int8_t enc_delta; // Variable, k. außerhalb des Hauptprogramms geändert werden
static int8_t last; // Variable für Interruptroutine
```

```

void encode_init( void )           // Initialisierung des Drehencoders
{
    int8_t new;

    new = 0;
    if( PHASE_A )
        new = 3;
    if( PHASE_B )
        new ^= 1;
    last = new;
    enc_delta = 0;
}

ISR( TIMER0_OVF_vect )           // durch Timer 0 ausgelöster Interrupt
{                                // für das Auslesen des
Drehgebers
    int8_t new, diff;
    new = 0;
    if( PHASE_A )
        new = 3;
    if( PHASE_B )
        new ^= 1;
    diff = last - new;
    if( diff & 1 )
    {
        last = new;
        enc_delta += (diff & 2) - 1;
    }
}

TCNT0 = 0x64;                   // Nachladewert für Timer 0
}

int8_t encode_read4( void )       // Wert von Drehencoder lesen
{
    int8_t val;

    cli();
    val = enc_delta;
    enc_delta = val & 3;
    sei();
    return val >> 2;
}

```

```

int main (void)
{
    #define AUSWAHL_REL 0           // Zustand 1
    #define AUSWAHL_EAZ 1          // Zustand 2

    char ea[5] = {' ', 'E', 'A', 'C', 'R'}; // Array für Schalt-Art der Relais
                                              // (EIN,AUS,ALLE,KEIN)
                                              // beachten, dass Array eins größer wird, da [0] n.V. wird

    uint8_t switch_werte[9] = {0x00,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80};
                                              // hier werden die Bitfolgen für die zu schaltenden Rel. definiert

    int rel_gewaehlt = 1; // da wir mit 1 und nicht 0 beginnen, wird mit 1 vorbelegt
    int8_t val = 1;
    char zpu[16];           //Array für Display-Text
    int state = 0;          // Zustandsvariable

// Relaisport als Ausgang schalten (alles in 1 Zeile schreiben)

DDRD |= (1 << PD7) | (1 << PD6) | (1 << PD5) | (1 << PD4) | (1 << PD3) |
(1 << PD2) | (1 << PD1) | (1 << PD0);

// Pullup fuer PB0 bis PB3

PORTB |= (1 << PB0) | (1 << PB1) | (1 << PB2);

// Timer 0 initialisieren

TCCR0  = (1<<CS00);           // TCCR0 = 0x01 = Eingangsteiler : 1;
TIMSK  = (1<<TOIE0);          // Interrupt bei Timer 0 Überlauf

sei();   // Interrupt frei geben

lcd_init(); // Display initialisieren

// Text in Zeile 1 ausgeben

lcd_clear();           // Display löschen
lcd_setcursor(0,1);   // Zeile1 wählen
lcd_string("Relaissteuerung"); // Text ausgeben

while(1) // Hauptschleife
{
    switch(state) // Auswahlschleife
    {
        case AUSWAHL_REL:           // Zustand 1
            if(TASTE) // solange Taste nicht gedrueckt wurde
            {
                val += encode_read4(); // Lesen des Drehencoders
                if ( val > 8 )

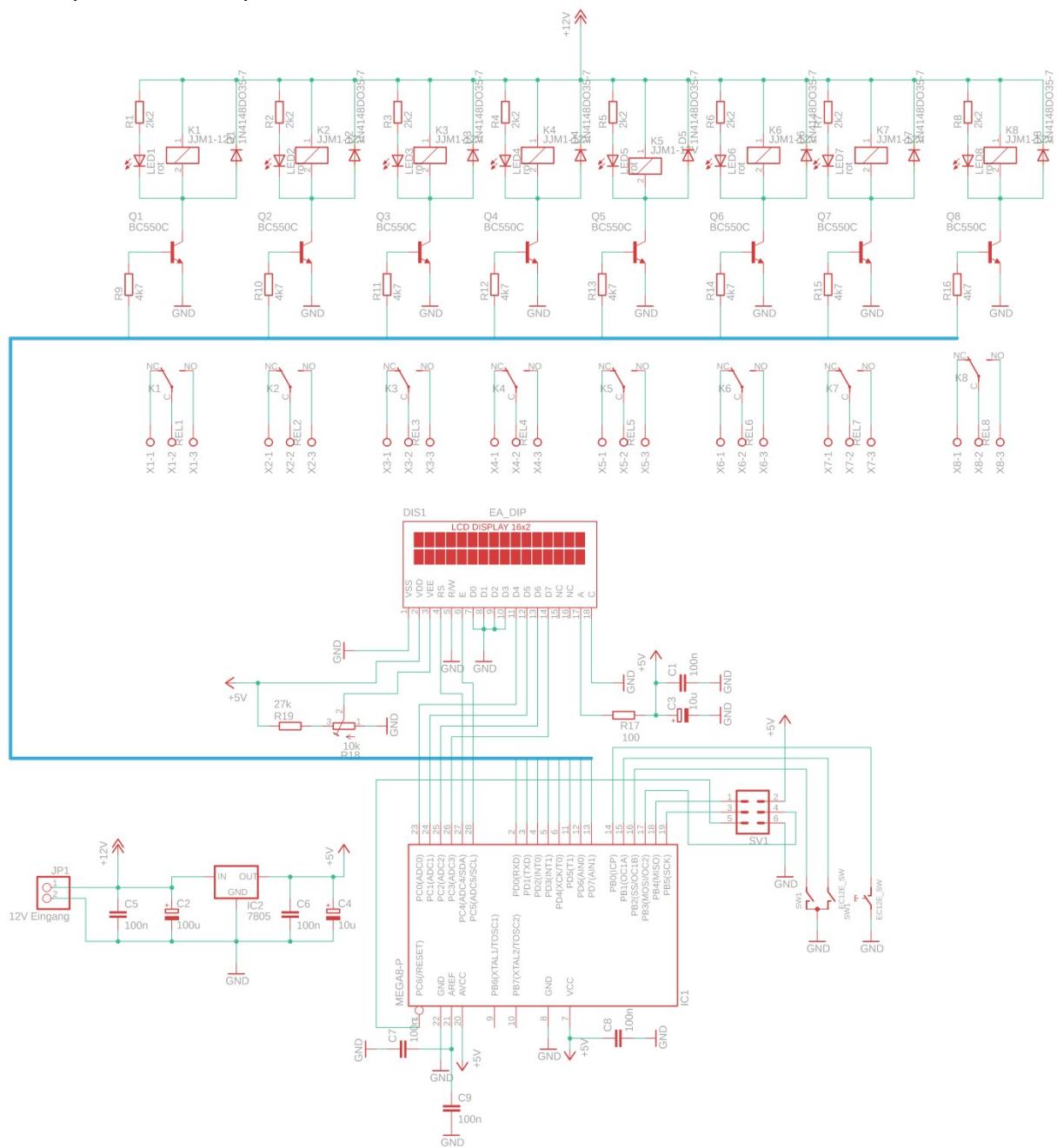
```

```

        val = 1;                                // auf 8 begrenzen
    if (val < 1 )
        val = 8;                                // mit 1 beginnen
    rel_gewaehlt = val;// engest.Wert merken, val wird überschr.
    sprintf(zpu,"Relais Nr.:%2d    ",rel_gewaehlt); // Textzusatz.
    lcd_setcursor(0,2);                      // Zeile 2 wählen
    lcd_string(zpu);                        // hier gewähltes Relais schreiben
}
else
{
    while(TASTE == 0);      // warten, bis Taste losgelassen wurde
    val = 1;                // wichtig, somit fängt Auswahl immer mit E an
    lcd_setcursor(0,2);      // Zeile 2 wählen
    sprintf(zpu,"Relais Nr.:%2d",rel_gewaehlt); // Textzusatz.
    lcd_string(zpu);        // aktuell gewähltes Relais schreiben
    state = AUSWAHL_EAZ;    // neuen Zustand vorgeben
}
break;                                // Ende Zustand 1
case AUSWAHL_EAZ:                     // Zustand 2
if(TASTE)                            // solange Taste nicht gedrueckt wurde
{
    val += encode_read4();            // Lesen des Drehencoders
    if ( val > 4 )
        val = 1;                    // auf 4 begrenzen
    if (val < 1 )
        val = 4;                    // mit 1 beginnen
    sprintf(zpu,"switch:%c    ",ea[val]); // Text zus. (Idx ea-Tab).
    lcd_setcursor(0,2);            // Zeile 2 wählen
    lcd_string(zpu);              // hier E,A,C oder R schreiben
}
else
{
    while(TASTE == 0);      // warten, bis Taste losgelassen wurde
    if(val == 1)
        PORTD |= switch_werte[rel_gewaehlt];
    else if(val == 2)
        PORTD &= ~switch_werte[rel_gewaehlt];
    else if(val == 3)
        PORTD = 0xFF;
    else if(val == 4)
        PORTD = 0;
    state = AUSWAHL_REL;
}
break;                                // Ende Zustand 2
default:
break;
}                                    // Ende Auswahlschleife
}                                    // Ende Hauptschleife (wird nie erreicht)
return 0;                            // Ende Programm
}

```

Schaltplan der Relaisplatine:



Stückliste für Relaisplatine, inklusive bereits verwendeter Bauteile, wir rechnen mit etwa 67 Euro:

Anz	im Schaltplan	Wert	Reichelt Best-Nr	Bemerkungen
1	IC1	Mikrocontroller	ATMEGA 8A-PU	
8	K1-K8	Relais	FIN 41.31.9 12V	Pro Stück
8	LED1	LED 3mm rot	LED 3-2000 RT	" "
8	R1-R8	Widerstand 2k2	VIS C2201FC100	" "
8	D1-D8	Diode 1N4148	1N4148	" "
1	PL1	Wannenstecker	WSL 6G	
1	IC2	Spannungsregler 5V	µA 7805	
1	C2	Elko 100µF/ 25V	NHG-A 100U 25	
2	C3,C4	Elko 10µF/ 50V	NHG-A 10U 50	Pro Stück
6	C1, C5-9	Kerko 100n	Z5U-2,5 100N	" "
8	Q1-Q8	Transistor NPN	BC 550C	" "
1	DIS1	LCD Display 2 Zeilen	LCD 162BL DIP	
1	R18	Trimm-Poti 10k	64Y-10K	
1	SW1	Drehencoder mit Ta.	STEC12E08	
1	R17	Widerstand 100 Ohm	METALL 100	
1	R19	Widerstand 27k	METALL 27K	
8	R9-R16	Widerstand 4,7k	VIS C4701FC100	Pro Stück
8	K1-K8	Leiterplattenklemme	CTB0509-3	" "
1		Leiterplatte		div. Firmen

Fangen wir unten links an: IC2 ist ein Spannungsregler, er regelt die Eingangsspannung von 12V auf stabile 5V. 12V werden für die Relais gebraucht, 5V für den Mikrocontroller und das Display. Hier sowohl für die Logik als auch für die Hintergrundbeleuchtung. C2,4,5,6 dienen der Unterdrückung von Schwingungen, sie sind nahe IC2 anzuhören. Ganz besonders muss auf eine zuverlässige Masseverbindung des mittleren Pins von IC2 geachtet werden, dieser bestimmt den Wert der Ausgangsspannung. Wird er unterbrochen, sieht es für Display und Prozessor schlecht aus! IC1 ist unser Mikrocontroller, die ISP Verbindung über den 6 poligen Stecker wird mit dem Programmiergerät verbunden. Rechts vom Mikrocontroller befindet sich der Drehencoder, er liefert 2 Rechtecksignale beim Drehen, welche 90 Grad phasenverschoben sind und deren Taktflanken vom Mikrocontroller gezählt werden. Durch Auswertung der Phasenverschiebung erkennt der Controller die Drehrichtung. Sollte diese in die falsche Richtung führen, sind die beiden Anschlüsse an PB1 und PB2 zu tauschen. Ganz rechts ist der Taster zu erkennen, er ist Bestandteil des Drehencoders, führt auf PB0. Sowohl Taster, als auch der Drehgeber, schalten auf Masse. Dazu muss im Mikrocontroller ein dort befindlicher Widerstand auf 5V gelegt werden, das geschieht durch Software und nennt sich „Pull-Up“. An Port C (oben links) ist das zweizeilige Display über insgesamt 6 Leitungen angeschlossen. R17 ist der Vorwiderstand für die LEDs der Hintergrundbeleuchtung, R18 ist ein Trimmsteller, es dient zur Einstellung des Display-Kontrasts.

Der Teil im oberen Bereich ist schnell erklärt. 8 Relais werden von 8 NPN-Transistoren angesteuert, zuvor wurde das bereits erklärt. Die 8 Si-Dioden dienen der Unterdrückung der Rückschlagspannung beim Abschalten der Relais, fehlen diese, werden die Transistoren und u.U. weitere Bauteile unmittelbar defekt. Parallel zu den Relaisspulen liegen LEDs, sie zeigen den Schaltzustand der Relais an. Die Vorwiderstände der LEDs sind an 12 V angepasst. Relais werden durch das komplette Port D geschaltet. Die Kontakte aller Relais sind an Klemmleisten herausgeführt. Es bietet sich an, das Ganze auf einer geätzten Leiterplatte anzubauen, alleine schon deshalb, um ggf. die VDE-Vorschriften zu erfüllen, sollten die Relais mehr als 50V AC oder 120V DC schalten.

Wichtige Anmerkung: Sollte mit den Relais Netzspannung (230 V AC) geschaltet werden, müssen beim Aufbau die Vorschriften nach VDE (12) unbedingt beachtet werden. Das betrifft vor allem die Einhaltung von Isolations-Abständen und Absicherung vor Überstrom.

Im Netz sind preiswerte Firmen zu finden, welche die Arbeit machen, man hat somit nicht den Stress mit Chemikalien. Zum Erstellen der Leiterplatte sind ebenfalls gute, freie Tools im WWW zu finden, z.B. EAGLE (13), leider wird damit der Platz für die Relais knapp, denn die kostenfreie Version unterstützt nur Platinen bis 80 mal 100 mm². Das Steckbrett hat hier nur die Aufgabe, Schaltungsteile zu testen.

Wenn wir das alles bis hierhin verstanden haben, können wir uns mit weiteren, interessanten Themen beschäftigen.

Wir wollen hier im Beispiel einmal die Netzfrequenz messen und diese sowohl in Hertz, als auch in Millisekunden darstellen. Wir vermuten bereits: Verwenden von Timer.

Ein anderes Beispiel zeigt uns, wie wir recht einfach die Umgebungstemperatur messen können.

Projekt 8: Netzfrequenz messen

Um eine Frequenz zu messen, können wir mit einem Mikrocontroller 2 Verfahren ausnutzen: Zählen der Perioden über 1 Sekunde oder Akkumulieren eines Zählers über 1 Sekunde. Wir schauen uns das mal genauer an.

Wenn wir Verfahren 1 nehmen, dann stünde in unserem Ergebnis eine 50, wenn wir die Netzfrequenz messen. Dieses ist eine recht kleine Zahl, sie ist auch ohne Komma und nebenbei mit einer Mess-Unsicherheit von +- 1 behaftet, sicher kein genaues Messverfahren. Die Genauigkeit nimmt jedoch zu, wenn wir z.B. 10 s messen und dann das Ergebnis durch 10 teilen. Leider stört dann die lange Messdauer. Wenn wir aber höhere Frequenzen messen, also mehrere Kilohertz, dann wird dieses Verfahren wieder interessant.

Verfahren 2 verwendet einen Counter, der von der Taktfrequenz des Mikrocontrollers hochgezählt wird. Wir gehen davon aus, dass dieser Counter nicht bis zum Überlauf hochzählt. Daher beachten wir: 50 Hz entsprechen 20 Millisekunden. Ein 16 bit Timer, z.B. Timer/Counter 1 – würde, mit 1 MHz getaktet, nach 65 ms überlaufen. Wir könnten somit bis zu 15,2 Hz messen. Diese untere Grenze hängt vom Zählertakt ab. Bei 50 Hz würde das Ergebnis 20000 sein. Das ist streng genommen die Periodendauer des gemessenen Wertes. Wir müssen ihn nun in Frequenz umrechnen, das ist ganz einfach: wir wissen ja, das 20000 20 ms entspricht, daher bilden wir den Kehrwert davon und kommen auf 50 Hz. Durch die jetzt höhere Auflösung können wir jetzt auch Nachkommastellen ermitteln. Selbstverständlich können wir auch die Zählerüberläufe mitzählen, das würde aber das Verfahren unnötig kompliziert machen. Ähnlich Verfahren 1 gibt es auch hier eine Genauigkeitsschranke. Wenn wir sehr hohe Frequenzen messen wollen, wird die Auflösung des Zählers immer geringer, denn ein Zählertakt ist 1 Mikrosekunde. Wenn wir 1 MHz messen, würde gerade 1 Takt gezählt werden.

Es ist sehr günstig, beide Verfahren zu kombinieren, also Verfahren 2 bei niedrigen Frequenzen, Verfahren 1 bei höheren.

Jetzt zur Praxis: wir hatten ja bereits erwähnt, Timer/Counter 1 zu verwenden. Für das Weitere empfehle ich, die Dokumentation für den AT Mega 8 bereitzuhalten, damit wir die Funktionen der Timer besser verstehen können. Wir wollen möglichst genau bleiben, daher versuchen wir alles, dass wir keine Messinformationen verlieren können (das passiert sehr leicht, wenn die Software Zeit benötigt). Wir bedienen uns daher des sogenannten Input-Capture Verfahrens, dieses speichert den momentanen Stand des Counters in ein Register ab, sobald an einem speziellen Eingang ein Ereignis auftritt. Gleichzeitig kann auch ein Interrupt ausgelöst werden, dieser liest das Ergebnis und führt uns dann beispielsweise in eine Berechnungsroutine. Das A und O der Programmierung von Mikrocontrollern ist die genaue Kenntnis der Register dieser. Ich gebe hier mal die wichtigsten für unsere Aufgabe an:

Timer/Counter 1 Control Register A – TCCR1A
Timer/Counter 1 Control Register B – TCCR1B
Timer/Counter 1 – TCNT1H and TCNT1L
Input Capture Register 1 – ICR1H and ICR1L
Timer/Counter Interrupt Mask Register – TIMSK

Der Timer/Counter 1 kennt 16 verschiedene Betriebsarten; für unser Beispiel reicht es, den Modus Normal zu verwenden, dieser ist nach dem Reset bereits vorgegeben. Daher müssen wir im TCCR1A nichts verändern.

Von TCCR1B verwenden wir nur das niederwertigste Bit, CS10 und die beiden höherwertigen Bits ICES1 und ICNC1. Mit CS10 wird der Counter erst einmal gestartet gestartet; ICES1 bestimmt, mit welcher Flanke das Input Capture ausgelöst wird. 0 bedeutet fallende Flanke, 1 steigende Flanke. Wir werden unsere Hardware später so auslegen, dass eine steigende Flanke den Input Capture auslöst. ICNC1 ist für Störunterdrückung. Wird es auf 1 gesetzt, dann wird diese aktiviert. Dabei wird das Input Capture um 4 Taktperioden verzögert, was uns hier aber nicht weiter stört.

TCNT1H und TCNT1L enthalten den momentanen Zählerstand, das interessiert uns nicht.

ICR1H und ICR1L enthalten nach Auslösung des Input Capture den momentanen Zählerstand. Dieser wird zur Berechnung der Periodendauer des gemessenen Signals herangezogen.

TIMSK: Bit 5 (TICIE1) schaltet den Input Capture-Interrupt ein, wir schreiben später eine Interrupt-Routine, welche die Auswertung vornimmt.

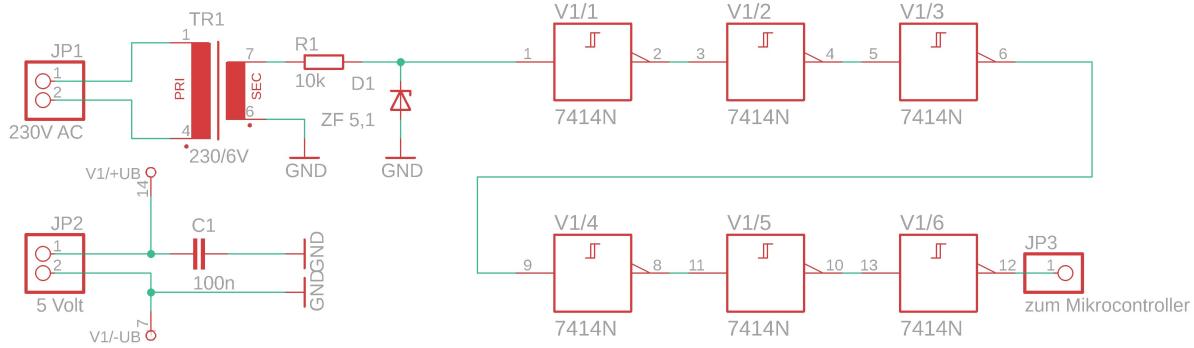
Mit diesen Fakten können wir uns an die Arbeit machen.

Hardware für die Messung der Netzfrequenz:

Da wir auf Sicherheit großen Wert legen, werden wir niemals die Netzspannung direkt mit unseren Schaltungen verbinden!! Ein Transformator hat sich sehr gut bewährt, die Netztrennung vorzunehmen. Ein Kleintrafo UI 30/5,5 206 von Reichelt für knappe 10 Euro kann das bereits. Wir müssen dann noch die Wechselspannung in ein Signal umformen, welches der Mikrocontroller versteht. Dafür verwenden wir einen Widerstand, einen Kondensator, eine Zener-Diode (14) und einen 6 fach-Inverter mit Schmidt-Trigger-Verhalten. Das bedeutet, dass dieser Baustein bei einer definierten Spannungsschwelle einschaltet, jedoch dann bei einer niedrigeren Spannungsschwelle wieder ausschaltet. Genau das brauchen wir, um aus der sinusförmigen Netzspannung ein Rechtecksignal zu erzeugen, welches der Mikrocontroller auswerten kann. Der gewählte Baustein hat 6 Funktionseinheiten, es bietet sich an, diese in Reihe zu schalten.

Stückliste für Hardware Messung Netzfrequenz, wir rechnen mit etwa 10 Euro:

Anz	im Plan	Wert	Reichelt Best-Nr	Bemerkungen
1		Kleintransformator	UI 30/5,5 206	
1		Inverter Schmidt-Trig.	74HC14	
1		Widerstand 10k	METALL 10k	
1		Kerko 100n	Z5U-2,5 100N	
1		Zenerdiode	ZF 5,1	



Wir können diese Schaltung auf dem Steckbrett aufbauen – besonders vorsichtig müssen wir mit dem Anschluss der Netzspannung vorgehen- am Besten den Transformator in ein separates Kästchen einbauen und mittels Netzstecker an eine Steckdose anschließen. Auch darauf achten, dass die 6 Volt-Anschlüsse (Trafo hat 2 davon, man kann beliebigen davon nehmen) nicht kurzgeschlossen werden. Wer es besonders brandsicher mag, fügt sowohl in die Zuleitung als auch in die 6V-Seite des Transformatoren Feinsicherungen ein, ich empfehle auf der 230V-Seite 32 mA und auf der 6V Seite 100 mA.

Das Steckbrett wird mit dem Display und dem Mikrocontroller ergänzt. Wir schließen das Display so an, wie wir es in den vorangegangenen Schaltungen getan haben. Der im Schaltplan gezeigte Ausgang JP3 wird mit Port B0, das ist Pin 14 des Mikrocontrollers, verbunden.

Nun beginnen wir mit der Programmierung.

Software Teil 9:

An dieser Stelle werde ich meinen Programmvorstellung platzieren; ich werde die einzelnen Schritte kommentieren:

```
// Messung der Netzfrequenz

#include "main.h"
#include <math.h>           // das können wir auch noch in main.h platzieren
#include <util/delay.h> // das können wir auch noch in main.h platzieren
```

// Portbelegung:

```
// PB3 MOSI Programmer
// PB4 MISO Programmer
// PB5 SCK  Programmer
// PC6 RESET Programmer

// PB0  Input Capture

// PC0 - PC3  LCD Data
// PC4    LCD   RS
// PC5    LCD   EN
```

```

static volatile uint16_t oldwert = 0; // Variable fuer Periodendauermesung
static volatile uint16_t newwert = 0;
static volatile uint16_t period = 0;
static volatile uint8_t done = 0; // optionales Flag fuer Ende des Interrupt,
// ist eigentlich nicht notwendig

ISR(TIMER1_CAPT_vect ) // Interrupt nach Input Capture-Ereignis
{
    newwert = ICR1;
    period = newwert - oldwert; // Ermitteln der Periodendauer
    oldwert = newwert;
//    done = 1;
}

int main (void)
{
    char zpu[16];
    float frequenz; // Gleitkommazahl, da Frequenz mit Nachkomma

// Timer 1

    TCCR1B |= (1 << ICES1) | (1 << ICNC1) | (1 << CS10); // Input Cap mit Filter
                                                               // Timer 1 Starten
    TIMSK |= (1 << TICIE1); // Interrupt bei Input Cap

    sei(); // Interrupts freigeben

    lcd_init(); // LCD initialisieren

    lcd_clear(); // LCD löschen
    lcd_setcursor(0,1); // Zeile 1 Text ausgeben
    lcd_string("Netzfrequenz"); // Text ausgeben

    while(1)
    {
//    done = 0; // nicht notwendig
//    while(done == 0); // nicht notwendig
        frequenz = 1000000.0 / (float) period; //aus Periodendauer Frequenz errechnen
        sprintf(zpu,"Wert: %2.3lf ",frequenz); // in LCD Puffer schreiben
        lcd_setcursor(0,2); // Zeile 2 auswählen
        lcd_string(zpu); // Ergebnis hineinschreiben
        _delay_ms(100); // 0,1 s warten
    }
    return 0;
}

```

Erklärungen:

Bei der Berechnung der Frequenz muss das Ergebnis vom Typ float (Gleitkomma) sein. Dafür müssen alle, an der Berechnung teilhabenden Variablen und konstanten ebenfalls float sein, ansonsten geht der Compiler davon aus, dass das Ergebnis eine Ganzzahl sein wird – und wir verlieren Genauigkeit! Um das sicherzustellen, müssen wir den Ganzzahl-Wert period in einen float Wert überführen, man nennt das Casten. Dafür steht das geklammerte (float) vor dem Ausdruck. Für die Konstante mit dem Wert 1000000 genügt es, mit einer Nachkommastelle dem Compiler mitzuteilen, dass es sich auch hier um eine Gleitkommazahl handelt.

Eine Anmerkung zum Format-String in der Anweisung:

```
 sprintf(zpu,"Wert: %2.3lf    ",frequenz);
```

Er sagt aus, dass frequenz ein float-Wert ist, welcher mit 2 Vor- und 3 Nachkommastellen ausgegeben werden soll.

Aber das ist noch lange nicht Alles! Wir müssen die Header-Datei „math.h“ einbinden und im makefile eintragen, dass beim printf Gleitkommazahlen verwendet werden. Wir modifizieren das Makefile daher an der Stelle:

```
LDFLAGS+= $(PRINTF_LIB) ...
```

In

```
LDFLAGS+= $(PRINTF_LIB_FLOAT) ...
```

Somit sollte dieses Programm nach der Übersetzung die Netzfrequenz ziemlich genau auf dem LCD-Display anzeigen. Ich sage an dieser Stelle „ziemlich genau“, denn das Timing des Mikrocontrollers hängt von einem internen RC-Oszillator ab, welcher auf Betriebsspannungs- und Temperaturänderungen reagiert. Um da noch ein wenig besser zu werden, empfehle ich noch die folgende Schaltungsänderung.

Optional:

Stückliste Option, wir rechnen mit etwa 1 Euro::

Anz	im Plan	Wert	Reichelt Best-Nr	Bemerkungen
1		Schwingquarz 8 MHz	8,0000-HC49U-S	
2		Kerkos 33 pF	KERKO 33P	Pro Stück

Wir legen den Quarz zwischen die Anschlüsse Pin 9 und Pin 10 des Mikrocontrollers, dann schalten wir je einen Kerko von Pin 9 nach Masse, sowie einen Kerko von Pin 10 nach Masse.

Jetzt müssen wir noch in der Konfiguration einige Änderungen vornehmen. Zuerst beschaffen wir uns die Software „Burnomat“ (16) und installieren diese. Bitte darauf achten, das gefixte Debian-Paket zu benutzen! Hierfür den Link benutzen. Wir starten den Burn-O-Mat (zu finden unter Entwicklung) – ja, man muss lange warten, bis das Fenster sich öffnet.

Zuerst tragen wir unter AVR Type den AT Mega 8 ein (einfach in der Drop Down Liste auswählen), danach den Button „FUSES“ betätigen. Links „Read Fuses“ drücken, das muss erfolgreich sein! Ansonsten Kabelverbindung und Stromversorgung prüfen! Fenster mit „OK“ verlassen.

In der 2. Reihe der Register im oberen Bereich steht rechts der Punkt „Oscillator/Clock Options“. Bitte hier auswählen. Eintragen: „nominal Frequency: 8 MHz“, dann „External Crystal or Ceramic Resonator“ wählen, anschließend die folgende Checkmarks setzen: „Type: Crystal“; „Frequency Range: 3 – 8 MHz“ – alles Andere so lassen und mit „Write Fuses“ bestätigen.

Wir beachten, dass ab diesem Zeitpunkt der Mikrocontroller nur noch mit dem angeschlossenen Quarz läuft!

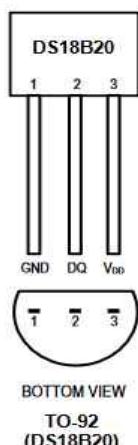
In unserer Software müssen wir auch noch den Teilerfaktor für Timer 1 ändern, da kommt der geschickte Leser aber selbst darauf, welches Register man dazu ändern muss!

Last, not least, müssen wir im Makefile die Variable F_CPU von 1000000 auf 8000000 erhöhen.

Projekt 9: Temperatur messen

Um mit einem Mikrocontroller Temperaturen zu messen, benötigen wir ein weiteres Bauteil. Dieses ist von der Firma Dallas (15) und nennt sich DS 18B20. Es hat nur 3 Anschlüsse und ist bei Reichelt für etwa 8 Euro erhältlich. Eigentlich ziemlich teuer, aber trotz seiner geringen Größe recht leistungsfähig. Für die Datenübertragung wird nur 1 Anschluss benötigt, die verbleibenden 2 Anschlüsse dienen der Spannungsversorgung. Ja, und es gibt einen Trick, der es erlaubt, einen Pin der Spannungsversorgung einzusparen. Selbstverständlich können die Datenausgänge mehrerer Sensoren als Bus parallelgeschaltet werden, denn jedes Bauteil hat eine eindeutige Bausteinadresse. Ich verweise hier auf die Internetseite von Peter Dannegger (17) mit Beispielen, auf die ich mich beziehe. Leider sind die dort veröffentlichten Beispiele schon älteren Datums, sie lassen sich nicht mit Geany compilieren. Ich habe mir die Mühe gemacht, die Dateien entsprechend anzupassen, nachdem ich daran gescheitert bin, neuere Programmversionen, die im Netz kursieren, auf einem Mega 8 zum Laufen zu bringen. Ich vermute, dass diese mit dem Timing nicht klarkommen. Die Software von Herrn Dannegger bedient sich eines eigenen Moduls (timebase.c), welches ein recht exaktes Timing erzeugt und – zumindest mit einer Taktfrequenz von 8 MHz funktioniert.

Der Sensor wird mit Pin 3 an 5V, mit Pin 2 an Port B0 und mit Pin 1 an Masse angeschlossen. Der Datenausgang Pin 2 wird über einen Widerstand von 4,7 k mit 5V verbunden. Bitte nicht falsch anschließen, das Bauteil wird sonst zerstört!



Wie bereits erwähnt, habe ich den Code überarbeitet, alles Unnötige herausgenommen, die LCD Routinen integriert und den Code optisch ein wenig geordnet. Wir verwenden in diesem Projekt 6 C-Files und 6 H-Files, sowie das makefile.

main.c und main.h: hier nur zyklische Messung und Ergebnis-Ausgabe

1wire.c und 1wire.h: Kommunikation mit dem Sensor auf Bit-Ebene

delay.c und delay.h: Verzögerung im Mikrosekunden-Bereich

timebase.c und timebase.h: Erzeugung eines genauen Zeitrasters von 1 s mit Timer 1

tempmeas.c und tempmeas.h: Messung der Temperatur und formatierte Ausgabe

lcd_routines.c und lcd_routines.h: diese sind uns bereits bekannt.

Dieses Projekt ist nun ein wenig größer, was auch bedeutet, dass im Makefile alle 6 C Quellen aufgenommen werden müssen. Da wir nicht direkt mit float-Werten arbeiten, können die Änderungen, welche wir für das printf im vorhergehenden Beispiel gemacht haben, wieder entfernt werden. Auch brauchen wir kein math.h mehr einzubinden. Das alles führt zu einem recht kompakten Binärkode, der gut in den AT Mega 8 passt. Wie bereits erwähnt, kann ich nicht garantieren, dass dieses Beispiel mit 1 MHz laufen wird. Daher sind mit (16) die Fuse Bits für externen Quarz anzupassen.

Hier ein Auszug aus dem verwendeten Makefile:

```
...
# MCU name
MCU = atmega8

# Main Oscillator Frequency
# This is only used to define F_OSC in all assembler and c-sources.
# F_OSC = 8000000
F_CPU = 8000000
....
# Target file name (without extension).
TARGET = main

# List C source files here. (C dependencies are automatically generated.)
SRC = $(TARGET).c lcd_routines.c 1wire.c tempmeas.c timebase.c delay.c
...
CFLAGS += -DF_CPU=$(F_CPU)
...
ASFLAGS += -DF_CPU=$(F_CPU)
...
#Additional libraries.

# Minimalistic printf version
PRINTF_LIB_MIN = -Wl,-u,vfprintf -lprintf_min

# Floating point printf version (requires MATH_LIB = -lm below)
PRINTF_LIB_FLOAT = -Wl,-u,vfprintf -lprintf_flt

PRINTF_LIB =

# Minimalistic scanf version
SCANF_LIB_MIN = -Wl,-u,vfscanf -lscanf_min

# Floating point + %[ scanf version (requires MATH_LIB = -lm below)
SCANF_LIB_FLOAT = -Wl,-u,vfscanf -lscanf_flt

SCANF_LIB =

MATH_LIB = -lm
```

Eigentlich spielt es keine Rolle, was hier bei F_CPU eingetragen wird, denn das System von Herrn Dannegger benutzt eine eigene Konstante (XTAL), sie ist in main.h eingetragen. Auch die Portpins für den Sensor; wir nehmen hier Port B0, sind ebenfalls dort hinterlegt. Auf keinen Fall diese in timebase.c überschreiben!

Software Teil 10:

Nun zu den Dateien im Detail:

main.c

```
#include "main.h"

int main( void )
{
    lcd_init();
    init_timer();
    sei();
    lcd_setcursor(0,1);
    lcd_string("Temperatur:");

    second = 0;

    for(;;)
    {
        if( second == 1 )           // Hauptschleife
        {
            start_meas();
            second = 2;
        }
        if( second == 3 )
        {
            read_meas();
            second = 0;
        }
    }
}
```

Nach der Initialisierung des Displays und des Timers wird der Interrupt eingeschaltet und ein Text in der 1. Zeile des LCD Displays ausgegeben. Dieser bleibt permanent angezeigt. Nun wird der Sekundentimer gesetzt und die Endlosschleife gestartet. In der 1 Sekunde wird die Temperaturmessung gestartet, in der 3. Sekunde der Messwert abgefragt. Das alles wiederholt sich permanent.

main.h

```
/************************************************************************/
/*          1-Wire Example                                         */
/*          Author: Peter Dannegger                                */
/*                      danni@specs.de                               */
/*          */                                                 */
/*#ifndef _main_h_                                              */
#define _main_h_
#include <avr/io.h>
#include <avr/interrupt.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include "lcd_routines.h"

#define XTAL      8000000UL

#define uchar unsigned char
#define uint unsigned int
#define bit uchar
#define idata
#define code

#define W1_PIN      PB0
#define W1_IN PINB
#define W1_OUT     PORTB
#define W1_DDR     DDRB

#include "1wire.h"
#include "delay.h"
#include "tempmeas.h"
#include "timebase.h"
//#include "uart.h"

#endif

char zpu[16];
```

Hier wurde bereits das Wichtigste erwähnt. Mit #ifndef wird geprüft, ob eine .h-Datei bereits vorliegt, sie wird dann nicht mehrfach eingebunden, was zu schwer zu findenden Fehlern führt!

Schließlich wird die Variable „zpu“ für den Displaytext global deklariert, das ist Geschmackssache.

1wire.c

```
/*****************************************************************************  
/*  
/*          Access Dallas 1-Wire Devices  
/*  
/*          Author: Peter Dannegger  
/*          danni@specs.de  
/*  
/******  
#include "main.h"  
  
// hier nicht ändern, ist bereits in main.h erfolgt  
  
#ifndef W1_PIN  
#define W1_PIN      PD6  
#define W1_IN PIND  
#define W1_OUT     PORTD  
#define W1_DDR      DDRD  
#endif  
  
  
bit w1_reset(void)  
{  
    bit err;  
  
    W1_OUT &= ~(1<<W1_PIN);  
    W1_DDR |= 1<<W1_PIN;  
    DELAY(DELAY_US(480));           // 480 us  
    cli();  
    W1_DDR &= ~(1<<W1_PIN);  
    DELAY(DELAY_US(66));  
    err = W1_IN & (1<<W1_PIN);      // kein Sensor entdeckt  
    sei();  
    DELAY(DELAY_US(480 - 66));  
    if((W1_IN & (1<<W1_PIN)) == 0) // Kurzschluss  
        err = 1;  
    return err;  
}  
  
uchar w1_bit_io( bit b )  
{  
    cli();  
    W1_DDR |= 1<<W1_PIN;  
    DELAY(DELAY_US(1));  
    if(b)  
        W1_DDR &= ~(1<<W1_PIN);  
    DELAY(DELAY_US(15 - 1));  
    if((W1_IN & (1<<W1_PIN)) == 0)  
        b = 0;  
    DELAY(DELAY_US(60 - 15));  
    W1_DDR &= ~(1<<W1_PIN);  
    sei();  
    return b;  
}  
  
uint w1_byte_wr( uchar b )  
{  
    uchar i = 8, j;  
    do{  
        j = w1_bit_io( b & 1 );  
    }
```

```

        b >>= 1;
        if( j )
            b |= 0x80;
    }while( --i );
    return b;
}

uint w1_byte_rd( void )
{
    return w1_byte_wr( 0xFF );
}

uchar w1_rom_search( uchar diff, uchar idata *id )
{
    uchar i, j, next_diff;
    bit b;

    if( w1_reset() )
        return PRESENCE_ERR; // Fehler, kein Sensor gefunden
    w1_byte_wr( SEARCH_ROM ); // ROM Suchkommando
    next_diff = LAST_DEVICE; // unveraendert bei letztem Sensor
    i = 8 * 8; // 8 Bytes
    do{
        j = 8; // 8 Bits
        do
        {
            b = w1_bit_io( 1 ); // lese Bit
            if( w1_bit_io( 1 ) ) // lese komplementaeres Bit
            {
                if( b ) // 11
                    return DATA_ERR; // Datenfehler
            }
            else
            {
                if( !b ) // 00 = 2 Sensoren
                {
                    if( diff > i || ((*id & 1) && diff != i) )
                    {
                        b = 1; // jetzt 1
                        next_diff = i; // naechster Durchgang 0
                    }
                }
                w1_bit_io( b ); // schreibe Bit
                *id >>= 1;
                if( b ) // speichere Bit
                    *id |= 0x80;
                i--;
            }
        }
        while( --j );
        id++; // naechstes Byte
    }
    while( i );
    return next_diff; // Suchen weiterfuehren
}

void w1_command( uchar command, uchar idata *id )
{
    uchar i;
    w1_reset();
    if( id )

```

```

{
    w1_byte_wr( MATCH_ROM );           // fuer einen einzelnen Sensor
    i = 8;
    do
    {
        w1_byte_wr( *id );
        id++;
    }
    while( --i );
}
else
{
    w1_byte_wr( SKIP_ROM );          // fuer alle Sensoren
}
w1_byte_wr( command );
}

```

Hier finden sich die Routinen für die Steuerung des Temperatursensors auf Bit-Ebene. Der Ablauf muss in einem definierten Zeitraster erfolgen. Neben Reset, Read und Write wird an dieser Stelle auch die Verwaltung mehrerer Sensoren bewerkstelligt.

1wire.h

```
#ifndef _1wire_h_
#define _1wire_h_
#define MATCH_ROM    0x55
#define SKIP_ROM     0xCC
#define      SEARCH_ROM   0xF0

#define CONVERT_T    0x44          // DS1820 Kommandos
#define READ         0xBE
#define WRITE        0x4E
#define EE_WRITE     0x48
#define EE_RECALL    0xB8

#define      SEARCH_FIRST 0xFF      // starte neue Suche
#define      PRESENCE_ERR 0xFF
#define      DATA_ERR     0xFE
#define LAST_DEVICE   0x00          // das letzte Teil gefunden
//                      0x01 ... 0x40: setze Suche fort

bit w1_reset(void);

uint w1_byte_wr( uchar b );
uint w1_byte_rd( void );

uchar w1_rom_search( uchar diff, uchar idata *id );

void w1_command( uchar command, uchar idata *id );
#endif
```

Hier werden die für den Temperatursensor spezifischen Kommandos hinterlegt, sowie die Funktionsdeklarationen für 1wire.c.

delay.c

```
#include "main.h"

//           Achtung !!!
// Aufpassen, dass während des delay kein Interrupt die Timer-Register veraendert
// oder den TCNT1H neu schreibt

void delay( int d )           // d = 0 ... 32000
{
    d += TCNT1;               // nicht atomic !
    while( (TCNT1 - d) & 0x8000 ); // nicht atomic !
}
```

Hier wird das genaue Zeitraster für die Mikrosekunden generiert. Nicht-Atomic bedeutet, dass die Anweisungen aus mehreren getrennten Zugriffen bestehen, diese dürfen keinesfalls durch Interrupts „zerrissen“ werden.

delay.h

```
#ifndef _delay_h_
#define _delay_h_

#define DELAY_US(x) ((uint)( (x) * 1e-6 * XTAL ))
#define DELAY(x)     delay(x)

void delay( int d );

#endif
```

Hier erfolgt die Definition der Berechnungsroutine für Mikrosekunden

timebase.c

```
*****  
/*                                         */  
/*                                         */  
/*          Precise 1 Second Timebase      */  
/*                                         */  
/*          Author: Peter Dannegger        */  
/*          danni@specs.de                 */  
/*                                         */  
*****  
#include "main.h"  
  
#define DEBOUNCE     256L           // Entprell-Takt (256Hz = 4msec)  
  
uchar prescaler;  
uchar volatile second;           // Sekundenzaehler  
  
ISR(TIMER1_COMPA_vect)  
{  
    uchar tcnt1h = TCNT1H;  
  
    OCR1A += XTAL / DEBOUNCE;       // neuer Vergleichwert  
  
    if( ++prescaler == (uchar)DEBOUNCE ){  
        prescaler = 0;  
        second++;                  // genau 1 s vergangen  
        #if XTAL % DEBOUNCE         // behandle Rest  
            OCR1A += XTAL % DEBOUNCE; // vergleiche einmal pro s  
        #endif  
    }  
    TCNT1H = tcnt1h;               // speichern fuer delay() !  
}  
  
void init_timer( void )  
{  
    TCCR1B = 1<<CS10;           // Teiler = 1  
    OCR1A = 0;  
    TCNT1 = -1;  
    second = 0;  
    prescaler = 0;  
  
    TIMSK = 1<<OCIE1A;  
}
```

Das funktioniert über einen Output Compare-Interrupt und gleichzeitig auch für das delay, beides bedient sich des Timers 1.

timebase.h

```
#ifndef _timebase_h_
#define _timebase_h_

extern uchar volatile second;

void init_timer( void );

#endif
```

Deklaration der externen Variable second und der Funktion init_timer

tempmeas.c

```
#include "main.h"

void start_meas( void ){
    if( W1_IN & 1<< W1_PIN ){
        w1_command( CONVERT_T, NULL );
        W1_OUT |= 1<< W1_PIN;
        W1_DDR |= 1<< W1_PIN; // Parasitaere Versorgung an
    }else{
        lcd_setcursor(0,2);
        lcd_string("Kurzschluss");
    }
}

void read_meas( void )
{
    unsigned char id[8], diff;
    unsigned int temp;

    for( diff = SEARCH_FIRST; diff != LAST_DEVICE; )
    {
        diff = w1_rom_search( diff, id );
        if( diff == PRESENCE_ERR )
        {
            lcd_setcursor(0,2);
            lcd_string("kein Sensor");
            break;
        }
        if( diff == DATA_ERR )
        {
            lcd_setcursor(0,2);
            lcd_string("Bus-Fehler");
            break;
        }
        if( id[0] == 0x28 || id[0] == 0x10 ) // Temperatur-Sensor
        {
            w1_byte_wr( READ ); // Lesekommando
            temp = w1_byte_rd(); // LO byte
            temp |= (uint)w1_byte_rd() << 8; // HI byte
            if( id[0] == 0x10 ) // 9 -> 12 bit
```

```

        temp <= 3;
sprintf( zpu, "%4d.%01d ", temp >> 4, (temp << 12) / 6553 ); // 0.1°C
lcd_setcursor(0,2);
lcd_string(zpu);
lcd_data(0xDF);           // das Grad-Zeichen
lcd_data(0x43);           // das grosse C
}
}
}

```

Hiermit wird eine Messung gestartet, aber auch der Messwert formatiert gelesen und in der 2. Zeile des LCD_Displays zusammen mit dem Grad-Zeichen als Zahl mit 1 Nachkommastelle dargestellt. Auch werden in Zeile 2 Fehlermeldungen angezeigt.

tempmeas.h

```

void start_meas( void );
void read_meas( void );

```

hier sind lediglich die beiden Funktionen deklariert.

lcd_routines.c und **lcd_routines.h** wurden bereits zuvor erklärt.

Man kann sich jetzt die Mühe machen, den Code abzutippen. Ich empfehle aber, den Code von Herrn Dannegger herunterzuladen und diesen anzupassen. Das geht schneller!

Wenn wir dieses Projekt fertiggestellt haben, können wir davon ausgehen, dass wir über umfangreiche Programmierkenntnisse verfügen.

Noch eine Anmerkung zum Format-String in der sprintf-Anweisung von tempmeas.c:

```

sprintf( zpu, "%4d.%01d ", temp >> 4, (temp << 12) / 6553 ); // 0.1°C

```

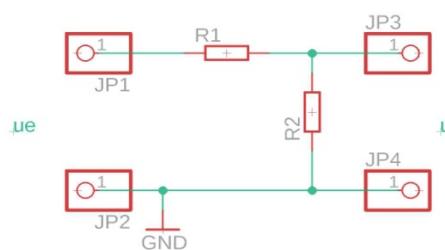
da wir ja nicht mit float-Werten arbeiten, müssen wir uns die Kommandarstellung selbst zusammenbasteln. Genau das macht diese, kompliziert aussehende Anweisung. Man kann gerne darüber nachdenken, wie das bewerkstelligt wird!

Projekt 10: Elektrische Leistung messen

In diesem Teil verwenden wir den Analog-Digitalwandler, welcher Teil des Mikrocontrollers ist. Es gibt 6 Eingangskanäle, diese werden über einen Multiplexer ausgewählt und können dann in einen Zahlenwert umgewandelt werden. Bitte beachten: Eingang 4 und 5 bieten nur 8 bit Auflösung, die anderen aber 10 bit. Die analogen Eingänge werden von Port C bereitgestellt. Da wir bislang das LCD Display an Port C betrieben haben, müssen wir nun dieses auf einen anderen Port umstellen, ich empfehle Port D.

Um Leistung messen zu können, werden wir einen gewissen Anteil analoger Schaltungstechnik verwenden und auch etwas Mathematik; diese ist für den Mikrocontroller aber kein großes Problem. Bekanntlich ist die elektrische Leistung das Produkt von Strom und Spannung. Wir begrenzen uns bei der Aufgabe auf Gleichstrom und Gleichspannung, denn dann haben wir keine Verschiebung von Strom und Spannung zu befürchten und es wird bei der Berechnung der Leistung wesentlich einfacher. Des Weiteren begrenzen wir die zulässige Spannung auf 15 Volt und den zulässigen Strom auf 1 Ampere. Somit können wir eine Maximalleistung von 15 Watt messen.

Wie geht das nun physikalisch? Spannung messen ist kein großes Problem. Wir können diese theoretisch direkt dem AD-Wandler zuführen, müssen jedoch beachten, dass der Wertebereich der Spannung eingehalten wird. Jeder Analog-Digitalwandler hat eine Spannungsreferenz, die vorgibt, wann der Zahlenwert der Eingangsspannung 100 % beträgt. Bei 10 bit bedeutet 100 % ein Zahlenwert von 1024 (2 hoch 10). Da unser Mikrocontroller nur positive Spannungswerte messen kann, entspricht der Wert 1024 genau der Referenzspannung. Anmerkung: da mit 0 bei der Zählung begonnen wird, ist der Endwert selbstverständlich 1023. Die Referenzspannung kann entweder die Betriebsspannung des Mikrocontrollers sein (5 Volt), eine extern anliegende Spannung in einem definierten Bereich (siehe Datenblatt) oder eine recht genaue interne Referenzspannung von 2,56 Volt. Wir werden uns in unserem Beispiel auf diese Spannung beziehen. Da wir jetzt wissen, dass 2,56 Volt 100 % sind, können wir leicht feststellen, was die Auflösung unserer Anordnung bietet: 2,56 V / 1024 sind 2,5 Millivolt. Das ist für unsere Anwendung bereits recht genau. Wir müssen nur darauf achten, dass wir den ganzen Bereich der Mess-Spanne gut ausnutzen, sodass wir die Genauigkeit beibehalten. Wenn wir 15 Volt messen (was ja höher als die Referenzspannung ist), müssen wir diese Spannung soweit verringern, dass 15 Volt genau 2,56 Volt betragen. Das macht man mit einem Spannungsteiler (18). Da diese Referenzspannung jedoch eine gewisse Toleranz aufweist, das Datenblatt nennt hier 2,3 bis 2,7 V, tun wir gut daran, diese zu messen (an Pin 21 gegen Masse) und den gemessenen Wert in unserer Software als Konstante zu verwenden.



$$\text{Es gilt: } \frac{u_a}{u_e} = \frac{R_2}{R_1 + R_2} \quad (1)$$

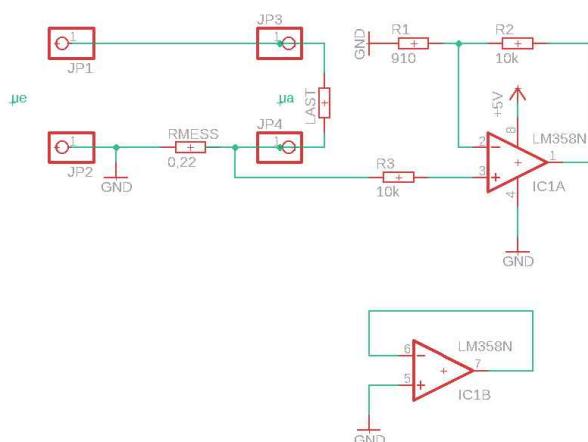
Wir rechnen das Teilverhältnis ua/ue aus, das beträgt 0,171. Wählen wir nun einen festen Widerstandswert, z.B. $R1 = 10000$ Ohm, dann können wir nach Umstellen der Gleichung (1) nach (2) auch $R2$ berechnen, in diesem Fall 2058 Ohm.

$$R2 = \frac{ua * R1}{ue - ua} \quad (2)$$

Leider sind solche „krummen“ Widerstandswerte schwer zu beschaffen, daher machen wir einen Kompromiss und nehmen einen kleinen Fehler in Kauf und verwenden für $R2$ zwei Widerstände mit 1000 Ohm, welche wir in Reihe schalten. Anstatt 2,56 V erhalten wir bei 15 Volt nun 2,5 V, das sind 60 mV weniger, insgesamt 24 Auflösungsstufen. Wer es supergenau möchte, nimmt für $R2$ einen Festwiderstand und einen variablen Widerstand (Potentiometer) zum Abgleich. Zum Schluss sei noch erwähnt, dass diese Schaltung nur funktioniert, wenn der AD-Wandler keine nennenswerte Last darstellt, er würde sonst zu $R2$ parallel geschaltet sein und einen Fehler hervorrufen. Das ist hier aber nicht zu befürchten.

Aber gerade an dieser Stelle gibt es jetzt doch Probleme. Beim Abgleichen ist mir aufgefallen, dass die Spannung „einen Buckel“ macht und mit den erwarteten Berechnungswerten überhaupt nicht korreliert. Ich habe das zuerst auf den Eingang des AD-Wandlers geschoben und diesen daher durch eine Schaltungsmaßnahme vom Spannungsteiler entkoppelt. Aber das Problem blieb bestehen. Ich stellte fest, dass schon die Spannung aus meinem hier eingesetzten Kalibrator am Eingang des Spannungsteiler viel zu gering war. Ich vermutete richtig, dass mein Kalibrator einen hochohmigen Ausgang aufweist. Sicherheitshalber habe ich die Teilerwiderstände um den Faktor 100 vergrößert ($R1 = 1$ Megohm, $R2 = 200$ Kilohm). Damit funktioniert alles bestens; die Genauigkeit habe ich noch verbessern können, indem ich direkt am Pin 21 (Aref) gemessen habe und das im Programm verwende.

Für die Strommessung müssen wir noch einen Schritt weitergehen. Direkt können wir diesen nicht messen, aber wir können nach dem Ohmschen Gesetz vorgehen, und den Spannungsabfall an einem Widerstand (Messwiderstand) feststellen und diesen dann messen. Der Spannungsabfall darf jedoch nicht zu groß werden, da sonst die angeschlossene Last zu wenig Spannung erhält.



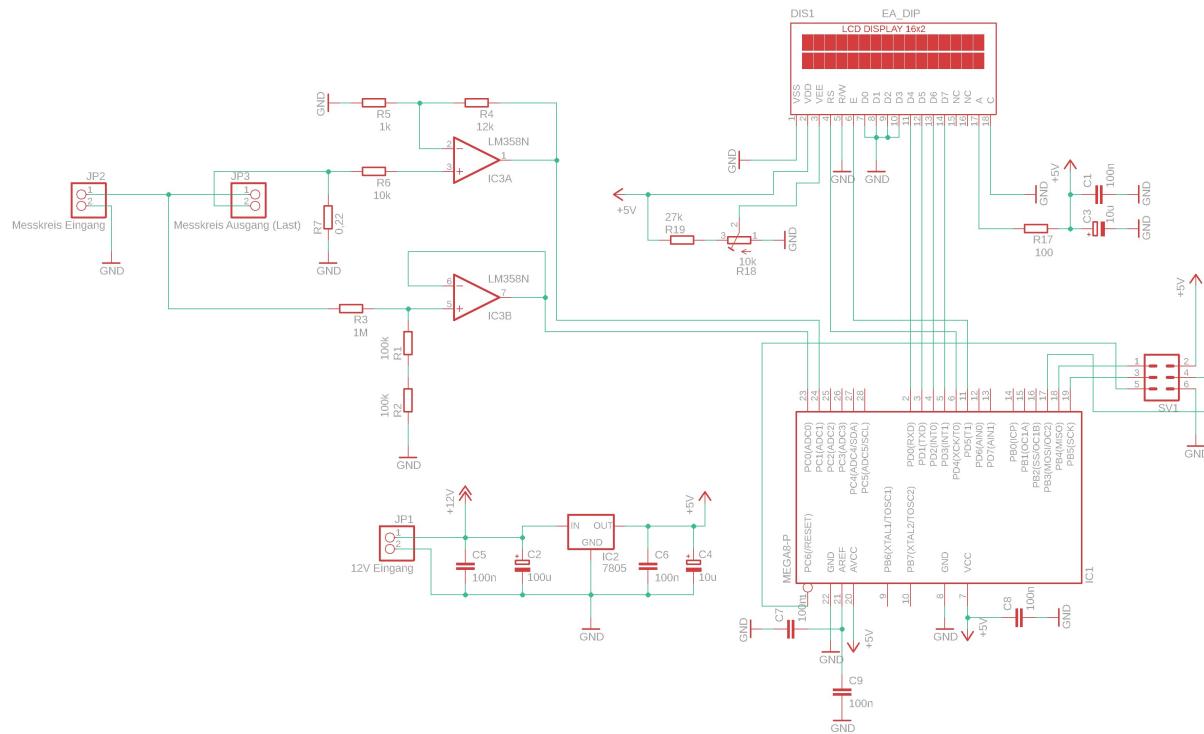
Diese Schaltung nennt sich „nicht-invertierender Verstärker“. IC1 ist ein Operationsverstärker (19), eine integrierte Analogschaltung mit 2 Funktionseinheiten in einem Gehäuse. Der nicht benutzte Teil 2 dieser Einheit wird definitiv auf 0 Volt gelegt, um Störungen auszuschließen.

Nehmen wir einen praktikablen Widerstandswert von 0,22 Ohm, dann fallen dort bei 1 Ampere gerade mal 0,22 Volt ab, was bei 15 Volt gewiss akzeptabel ist, ebenso auch bei 5 Volt. Leider müssen wir jetzt aber die Spannung verstärken, sodass wir auf 2,56 Volt kommen, sobald am Messwiderstand 0,22 V anliegen. Der Verstärkungsfaktor wäre somit 11,64.

Für den nicht-invertierenden Operationsverstärker ist der Verstärkungsfaktor: $g = 1 + \frac{R_2}{R_1}$

Für die im Schaltplan aufgeführten Werte wäre dieser 11,99, also zu hoch. Man kann aber die Widerstandswerte feintunen (durch Reihen – Parallelschaltung oder Potentiometer, wenn man absolute Genauigkeit erreichen möchte. Gleich vorweg erwähnt: der hier verwendete Operationsverstärker hat leider auch bereits einen recht großen Offsetfehler, das ist zu berücksichtigen. Praktikabel ist es, und so habe ich es auch gemacht: R1 ist 1k, R2 ist 12k. R3 ist nur ein Schutzwiderstand, er soll bei Unterbrechung des Messwiderstands den Strom in den Operationsverstärker begrenzen.

Hier zeige ich noch einmal den kompletten Schaltplan des Leistungsmessgerätes:



Hinweis: Die Masseverbindungen (GND) im Messkreis müssen mit richtig dickem Draht gestaltet werden, ansonsten wird ein großer Messfehler durch zusätzlichen Spannungsabfall entstehen!

Stückliste für Leistungsmessgerät, komplett inklusive bereits verwendeter Bauteile, wir rechnen mit etwa 28 Euro:

Anz	im Schaltplan	Wert	Reichert Best-Nr	Bemerkungen
1	IC1	Mikrocontroller	ATMEGA 8A-PU	
2	R1, R2	Widerstand 100 k	METALL 100k	Pro Stück
1	R3	Widerstand 1 Megohm	METALL 1M	
1	R4	Widerstand 12 k	METALL 12K	
1	R5	Widerstand 1k	METALL 1K	
1	R6	Widerstand 10k	METALL 10k	
1	R7	Widerstand 0,22 Ohm	2W DRAHT 0,22	
1	C2	Elko 100µF/ 25V	NHG-A 100U 25	
2	C3,C4	Elko 10µF/ 50V	NHG-A 10U 50	Pro Stück
6	C1, C5-9	Kerko 100n	Z5U-2,5 100N	" "
1	PL1	Wannenstecker	WSL 6G	
1	DIS1	LCD Display 2 Zeilen	LCD 162BL DIP	
1	R18	Trimm-Pot 10k	64Y-10K	
1	IC2	Spannungsregler 5V	µA 7805	
1	R17	Widerstand 100 Ohm	METALL 100	
1	R19	Widerstand 27k	METALL 27K	
1	IC3	Operationsverstärker	LM 358 DIP	
3	JP1 – JP3	Leiterplattenklemme	CTB0509-3	Pro Stück
1		Leiterplatte		div. Firmen

Wir kommen nun zu der Software.

Die Erfassung der Analogwerte Spannung und Strom erfolgt über ADC0 und ADC1. Das wird elegant über einen Interrupt gemacht, dieser tastet beide Kanäle 100 mal nacheinander ab und speichert das Ergebnis in einem Array. Wenn die Routine fertig ist, signalisiert sie dem Hauptprogramm dieses. Das wechselseitige Einlesen von ADC0 und ADC1 wird durch eine State machine vorgenommen. Wir müssen dabei beachten, dass die Einstellung eines Kanals erst für die nächste Abtastung gültig wird, daher sind in der Interruptroutine die Kanaleinstellungen versetzt.

In der Hauptroutine wird zuerst der AD-Wandler konfiguriert, er soll die interne Referenzspannung verwenden, frei laufen, das mit einer Abtastrate von 208 Mikrosekunden (das reicht völlig für unsere Aufgabe), er soll beim Ende einer Wandlung einen Interrupt auslösen. Wenn das vollzogen ist, muss einmalig eine Wandlung gestartet werden, ab da läuft der AD Wandler selbstständig.

Wir initialisieren weitere Hardware und geben in der Zeile 1 des LCD Displays einen Text aus. Dann setzen wir die State machine zurück und schalten den Interrupt ein.

Schließlich gelangen wir in eine Endlosschleife. Zuerst setzen wir die Variablen, welche in der Schleife aufsummiert werden, auf Null, des Weiteren löschen wir das Flag, welches das Ende der vollständigen Erfassung von 100 Samples der beiden Analogkanäle anzeigt. Dann warten wir auf die 100 Samples und bilden anschließend den Mittelwert über diese, getrennt für Spannung und Strom. Wir wissen, dass bisher nur Rohwerte vorliegen, diese müssen in nachfolgenden Schritten in reale Werte umgerechnet werden. Schließlich werden diese in der zweiten Zeile des LCD Displays ausgegeben – weiterhin deren Produkt als Leistung.

Software Teil 11:

```
// Leistungsmessung

#include "main.h"
#include <math.h>
#include <util/delay.h>
// Portbelegung:

// PB3 MOSI Programmer
// PB4 MISO Programmer
// PB5 SCK Programmer
// PC6 RESET Programmer

// PB0 Input Capture

// PD0 - PD3 LCD Data
// PD4 LCD RS
// PD5 LCD EN

#define CH0          0      // State Machine fuer Kanalzuordnung
#define CH1          1      // Groesse AD Sample Tabelle
#define TABSIZE     100

uint8_t static volatile chan_status;           // 2 Zustaende
uint8_t static volatile ad_index;             // Tabellenindex
uint8_t static volatile samples_ready;        // 100 Werte geschrieben
uint16_t static volatile ad_samples[TABSIZE][2]; // Tabelle fuer AD Werte

ISR(ADC_vect) // Einlesen von 2 Kanaelen
{
    switch(chan_status)
    {
        case CH0:                      // Einlesen Kanal 1 (Spannung)
            ADMUX &= ~(1 << MUX0);
            ad_samples[ad_index][CH0] = ADCW;
            chan_status = CH1;
            break;
        case CH1:                      // Einlesen Kanal 2 (Strom)
            ADMUX |= (1 << MUX0);
            ad_samples[ad_index][CH1] = ADCW;
            chan_status = CH0;
            ad_index++;                // Tabellenindex erhoehen
            break;
        default:
            break;
    }
    if(ad_index > TABSIZE - 1)           // alle 100 Werte gelesen
    {
        ad_index = 0;
        samples_ready = 1;
    }
}
```

```

int main (void)
{
    int i;
    char zpu[16];
    unsigned long volt_summe;           // Summe fuer Mittelwert
    unsigned int volt_average;          // Mittelwert Spannung
    unsigned long amp_summe;           // Summe fuer Mittelwert
    unsigned int amp_average;           // Mittelwert Strom
    float vref = 2.497;                // am Chip nachgemessen
    float spannung;                   // realer Spannungswert
    float strom;                      // realer Spannungswert
    float leistung;                   // realer Spannungswert

    // AD Wandler konfigurieren

    void ad_init(void)
    {
        ADMUX |= (1<<REFS1)|(1<<REFS0); // ADC Ref auf AREF geschaltet
        ADCSRA |= (1<<ADEN)|(1 << ADFR)|(1 << ADIE)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0); // ADC eingeschaltet,
        // Freilaufend, Interrupt, Teiler / 128
        // Eingangstakt 8E6 / 128 / 13 (13 Zyklen fuer Wandlung)      // T = 208 µs

        ADCSRA |= (1<<ADSC);           // eine ADC-Wandlung starten
    }
    lcd_init();
    ad_init();

    lcd_clear();
    lcd_setcursor(0,1);             // Zeile 1 Text ausgeben
    lcd_string("Leistungsmessung");

    chan_status = CH0;              // State initialisieren

    sei();

    while(1)
    {
        volt_summe = 0;             // vor Aufsummieren auf 0 setzen
        amp_summe = 0;               // vor Aufsummieren auf 0 setzen
        samples_ready = 0;           // Ready-Flag loeschen

        while(samples_ready == 0); // warte auf 100 Samples

        // Summe ueber Iteration (fuer Mittelwert)
        for(i = 0; i < TABSIZE; i++)
        {
            volt_summe = volt_summe + ad_samples[i][0];
            amp_summe = amp_summe + ad_samples[i][1];
        }
        // Mittelwert ueber Summe

        volt_average = volt_summe / TABSIZE;
        amp_average= amp_summe / TABSIZE;
    }
}

```

```

// diese Werte beruhen auf Spannungsteiler G = 0,166 (1M/200k)
// und Stromverstaerker mit G = 13 (12k/1k)
// reale Spannung, Strom, Leistung berechnen
// auskommentierte Zeilen dienen nur der Verdeutlichung
// spannung = (float) volt_average * vref / 1024.0;
    spannung = (float) 15.0 * volt_average / 1024.0;
// strom = (float) amp_average * vref / 1024.0;
    strom = (float) 1.0 * amp_average / 1024.0;
// leistung = (15.0/vref)*spannung * (1.0/vref)*strom;
    leistung = spannung * strom;

//      sprintf(zpu,"%4.1fV %2.1fA %4.1fW",(15.0/vref)*spannung,(1.0/vref)*strom,leistung);
    sprintf(zpu,"%4.1fV %2.1fA %4.1fW",spannung,strom,leistung);
    lcd_setcursor(0,2);
    lcd_string(zpu);
    _delay_ms(100);
}
return 0;
}

```

Wir müssen selbstverständlich wieder im makefile das printf für float konfigurieren, wie im Beispiel Frequenzmessung.

Projekt 11: Leistung von Hochfrequenz messen

Wenn wir Funkamateure sind, möchten wir gerne wissen, welche Leistung in einer Verstärkerstufe erzeugt wird. Leider können wir diese nicht wie im vorangegangenen Beispiel ermitteln, denn wir haben es zum einen mit Wechselstrom und Wechselspannung, zum anderen mit einer hohen Frequenz zu tun. Wir sprechen hier von Bereichen ab 1 Megahertz bis etwa 600 Megahertz. Unter diesen Bedingungen wird die Hardware bereits sehr kompliziert, denn bereits kleine Kapazitätswerte und Induktivitäten wirken sich extrem stark auf die zu messenden Größen aus. Daneben handelt es sich bei den Leistungen um eine sehr großen Messbereich, beginnend von Picowatt bis hin zu Milliwatt, also 4 Größenordnungen. In der HF Technik hat sich ein Standard bewährt: man legt Schaltungen so aus, dass deren Ein- und Ausgangsimpedanz stets 50 Ohm beträgt. So ist es nicht verwunderlich, wenn wir die Leistung an einem Widerstand von 50 Ohm ermitteln. Da wir über mehrere Größenordnungen messen wollen, bietet es sich an, die Messgröße in Dezibel auszudrücken.

Hier ein wenig Theorie:

Wenn wir es mit Spannung oder Strom zu tun haben, dann beträgt das Übertragungsmaß in dB:

$$L = 20 * \log(U_2/U_1).$$

Beispiel: $U_1 = 1 \text{ V}$, $U_2 = 10\text{V}$, dann ist

$$L = 20 * \log(10) = 20 * 1 = 20 \text{ dB}$$

Hätten wir 1V und 100 V, dann wären es 40 dB.

Man erkennt, dass jede Erhöhung um 20 dB eine Multiplikation mit 10 bedeutet.

0 dB ist stets Faktor 1, negative dB-Zahlen sind Brüche, so z.B. -20 dB = 0,1 usw.

Das alles ist nur für Spannung, Strom, Schalldruck gültig, also für Größen, welche keine Produkte aus anderen Größen darstellen.

Bei Leistung sieht die Berechnung daher anders aus: (das ergibt sich aus der Natur des Logarithmus)

$$L = 10 * \log(P2/P1)$$

Wir erkennen, dass jede Erhöhung um 10 dB eine Multiplikation mit 10 bedeutet.

Eine große Erleichterung ist es, dass Dezibel einfach addiert bzw. subtrahiert werden können. Verstärkungsfaktoren können das nicht, da sie Produkte sind.

Das Dezibel ist ein reines Zahlenverhältnis und daher dimensionslos, in der Technik gibt es jedoch einige Bezugswerte, nämlich: Milliwatt eff. (dBm), Volt eff. (dBV), 0,775V eff. (dBu). Alle beziehen sich auf 0 dB. Zur Erklärung von eff. muss ich ein wenig ausholen:

Bisher hatten wir uns nur mit Gleichstrom und Gleichspannung beschäftigt, jedoch ist Hochfrequenz, wie auch Audio und auch die Netzspannung eine Wechselspannung und daraus resultierend auch Wechselstrom. Es ist nicht schwer zu erkennen, dass deren Spannungswerte nicht 1:1 mit Gleichspannung verglichen werden dürfen, denn anders als dort, können die Momentanwerte der Wechselspannung auch Null werden und sich auch umpolen. Ganz besonders spielt die Kurvenform der Wechselspannung eine Rolle, sowie auch der möglicherweise überlagerte Wert einer Gleichspannung. Um dort nicht zuviele Abhängigkeiten berücksichtigen zu müssen, hat man in der Technik den Kompromiß gewählt und sich auf einen reinen Sinus bezogen. Mit etwas höherer Mathematik kommt man nun dahin, dass 1 Volt eff. einer sinusförmigen Spannung von 1,414 Volt Spitzenwert entspricht, welcher nach Durchtritt durch die Nulllinie einen negativen Spitzenwert von -1,414 Volt annimmt, bevor er wieder durch den Nulldurchgang tritt. 1,414 ist übrigens die Wurzel aus 2.

Die Größe dBm bezieht sich somit auf 1 Milliwatt eff (im englischsprachigen Raum auch mit RMS bezeichnet). 10 dBm sind demzufolge 10 dBm, ein gängiger Amateur-Kurzwellensender macht locker 50 dBm (100 Watt). An einer Antenne hat man es mit -70 dBm zu tun, das kann jeder mal nachrechnen, welche Leistung das ist.

Da wir uns in Elektrotechnik gut auskennen und bereits das Ohmsche Gesetz kennen, sind wir in der Lage nun auch die Spannung an 50 Ohm auszurechnen, wenn die Leistung gegeben ist. $P = U * I$ und $U = R*I$, daraus abgeleitet: $U = \sqrt{P*R}$.

Das alles gilt aber nur, wenn Strom und Spannung in Phase sind, andernfalls haben wir einen nicht zu vernachlässigenden Messfehler. Um auch hier genau werden zu können, müssen wir, ähnlich dem vorangegangenen Beispiel sowohl Spannung als auch Strom in zeitlich hochauflösender Abtastung getrennt ermitteln und mathematisch für jeden Sample separat berechnen. Das würde jedoch den Aufwand unserer Aufgaben erheblich übersteigen.

Zum Glück haben sich Ingenieure bereits Gedanken gemacht, wie man auf einfache und preiswerte Art die Leistung von Hochfrequenz messen kann. Die Firma Analog Devices (20) hat einen interessanten Halbleiter entworfen, dieser hat die Bezeichnung AD 8307. Es handelt sich um einen logarithmischen Verstärker. Dieser erfasst die Spannung an einem Widerstand von 50 Ohm und liefert eine Gleichspannung, welche der Effektivleistung der am Widerstand anliegenden HF Leistung entspricht. Zusammen mit der notwendigen Beschaltung findet sich dieses Bauteil auf einer Platine, welche für 14,90 € bei (21) zu beziehen ist. Unsere Aufgabe ist es nun, aus den ausgegebenen Spannungswerten dieser Platine eine LED-Kette anzusteuern, welche die Angaben in dBm repräsentiert. Wir wollen hier von -80 dBm bis +20 dBm messen, sind uns aber bewusst, dass der Messbereich der Anordnung nur von -75 dBm bis + 15 dBm reicht. Daher gehen wir an den Enden des Messbereichs Kompromisse ein.

Die Spannung, welche der AD 8307 ausgibt, gehorcht folgender Abhängigkeit:

Die Spannung am Ausgang des AD 8307 wächst um 25 mV pro dBm. Bei -74 dBm beträgt sie 0,25V, bei +16 dBm bereits 2,5V.

In 2 Schritten ermitteln wir, wieviel dB die gemessene Spannung ist: z.B. 2,364 V sind $2,364V - 0,25V / 0,025V = 84,56 \text{ dBm}$. Davon müssen wir im 2. Schritt die -74 dBm abziehen, die ja 0,25V sind: $84,56 \text{ dBm} - 74 \text{ dBm} = 10,56 \text{ dBm}$ (das sind 11,38 mW eff und an 50 Ohm 0,754 Veff).

Um aus den dBm die Ausgangsspannungen des AD 8307 zu ermitteln, müssen wir diesen Schritt gehen:

$$U = (74 \text{ dBm} - \text{einzusetzende dBm}) * 0,025V$$

Beispiel: für -50 dBm:

$$U = 74 \text{ dBm} - 50 \text{ dBm} = 24 \text{ dBm} * 0,025V = 0,6V.$$

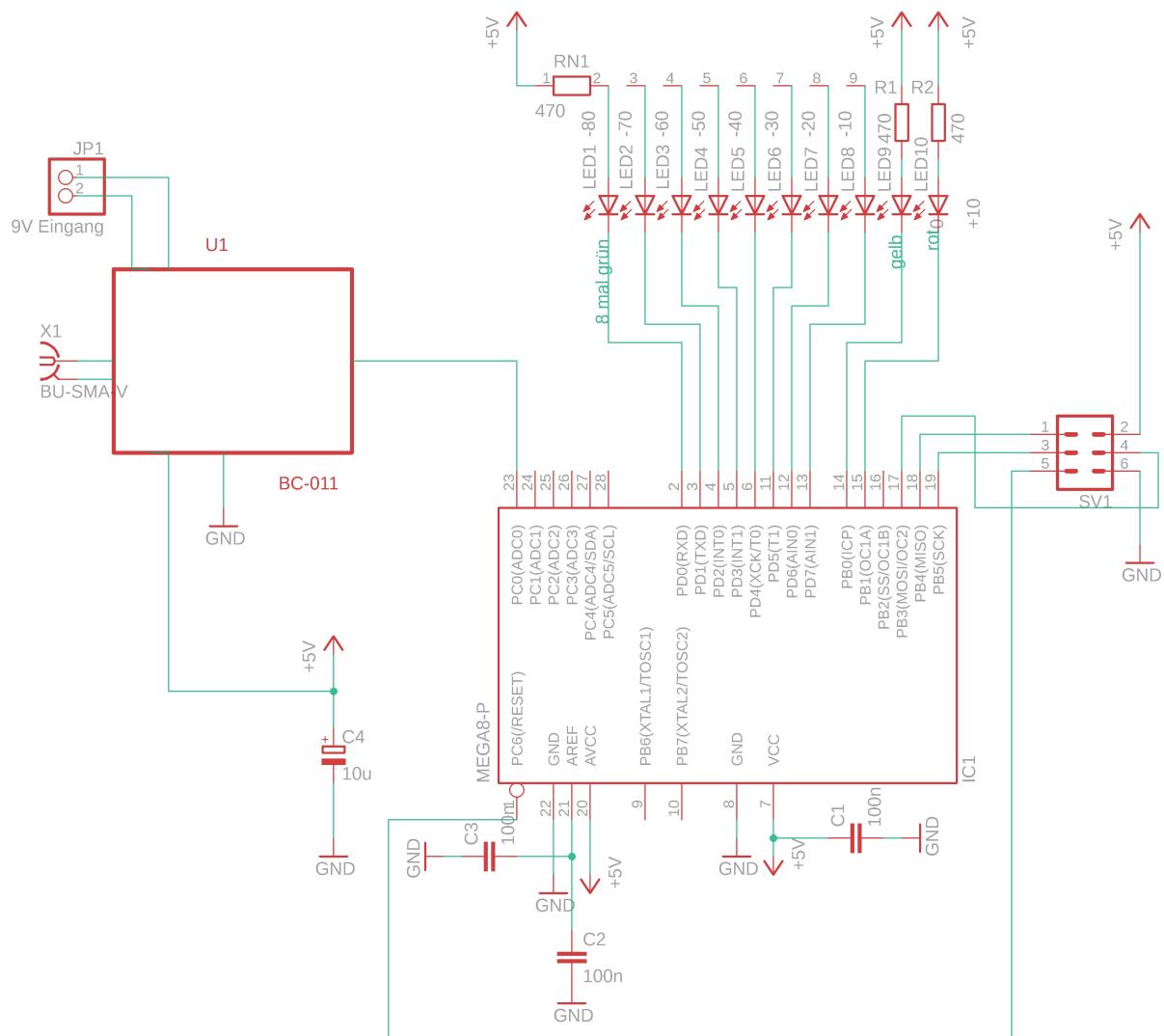
Wir können uns somit eine Tabelle schaffen, in der Spannungswerte gegenüber dB-Werten abgelegt werden. Unser Ziel sei es, nicht auf das Komma genau diese Information auszugeben (man kann schließlich mit einem genauen Voltmeter nachmessen), sondern übersichtlich die dBm-Werte an einer LED Kette auszugeben. Wir werden 10 LED verwenden, beginnend von -80 dBm bis + 10 dBm. 8 LEDs werden grün leuchten, die LED für 0 dBm gelb und die für + 10 dBm rot.

Die Hardware:

Da wir auf Timing keine Rücksicht nehmen müssen, genügt der ATMEG 8 mit internem Takt von 1 MHz. In der Stückliste sind weitere Bauteile aufgeführt:

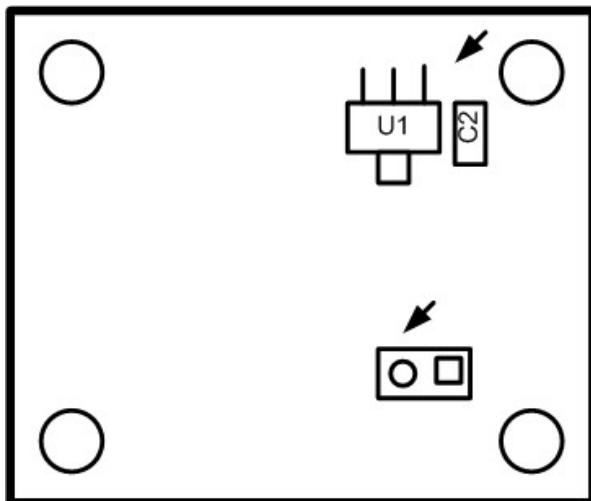
Stückliste für HF Leistungsmesser, wir rechnen mit etwa 22 Euro:

Anz	im Plan	Wert	Reichelt Best-Nr	Bemerkungen
1	IC1	Mikrocontroller	ATMEGA 8A-PU	
8	LED1-8	LED grün 3mm	LED 3MM 2MA GN	Pro Stück
1	LED9	LED gelb 3mm	LED 3MM 2MA GE	
1	LED10	LED rot 3mm	LED 3MM 2MA RT	
1	RN1	Widerstandsnetzwerk	SIL 9-8 470	
2	R1,R2	Widerstand 470 Ohm	METALL 470	Pro Stück
3	C1-C3	Kerko 100n	Z5U-2,5 100N	Pro Stück
1	C4	Elko 10µF/ 50V	NHG-A 10U 50	
1		Lochrasterplatine		
1	U1	HF Leistungsmesser	BC-011	Box 73



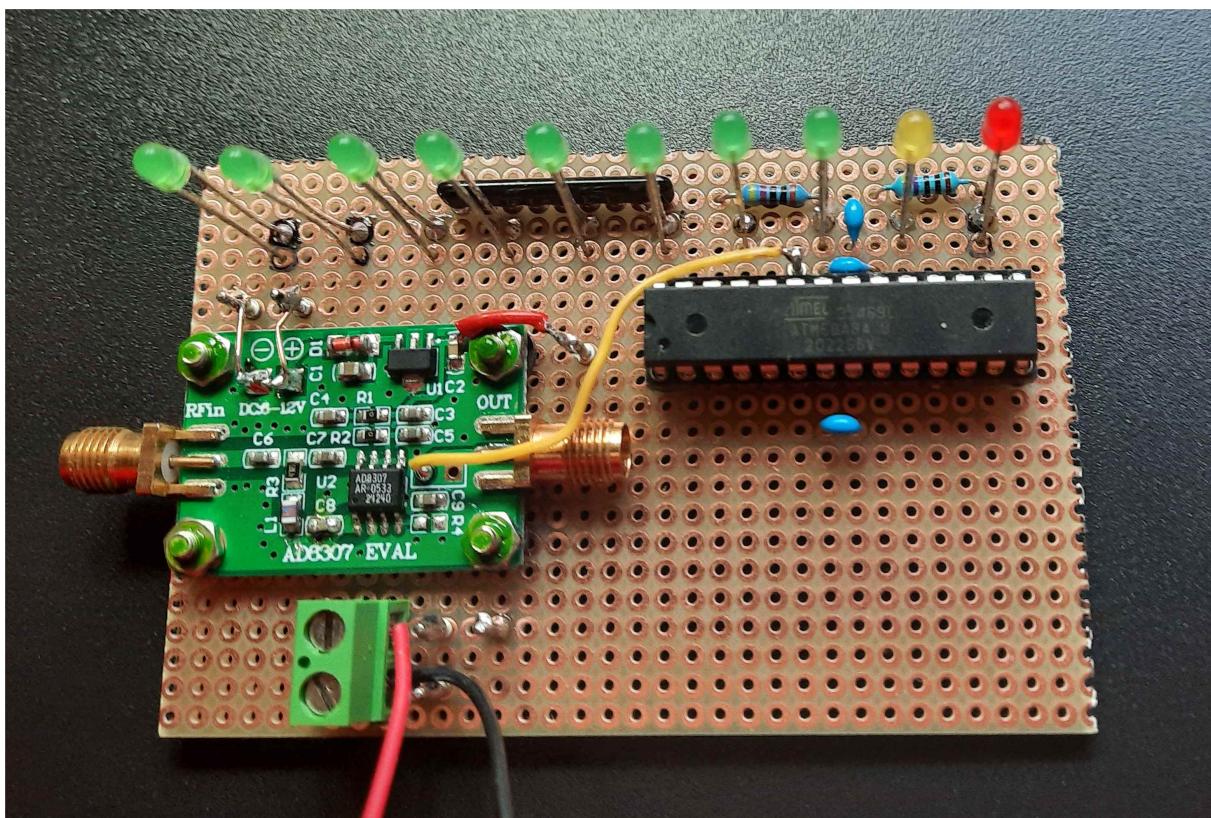
zur Erläuterung:

Die Schaltung wird mit einer 9-Volt Batterie betrieben. Da auf dem BC-011 bereits ein Spannungsregler vorhanden ist, bietet es sich an, diesen zu benutzen und dessen 5 Volt für die Versorgung des Mikrocontrollers und der LEDs zu verwenden. Dafür ist minimale Lötarbeit an der Platine des Leistungsmesser notwendig:



Rechts am U1 bzw. C2 oben kann die 5V-Leitung angelötet werden. Der Eingang des AD-Wandlers ADC0 kann an den Lötpunkt (Pfeil) angelötet werden, ohne die SMA Buchse des Ausgangs zu verwenden.

Ein Bild der fertigen Lochrasterplatine soll hier Klarheit schaffen:



Hier ist der Elko C4 weggelassen, auch der ISP-Anschluss wurde nicht bestückt! Der große Platz rechts unten dient zur Aufnahme der 9V-Batterie (sie steht auf ihrer Schmalseite).

Software Teil 12:

Die Software dazu ist recht einfach, denn anders als bei der vorherigen Aufgabe müssen wir nur 1 Analog-Kanal einlesen, können daher auf die Statemachine verzichten und für die Messdaten ein eindimensionales Array verwenden. Wie schon bemerkt, müssen die analogen Spannungen des Messmoduls in dBm-Werte gewandelt werden, das macht eine Funktion, die wir schreiben werden. Zum Glück liefert das Modul passende Spannungen, die direkt ohne weitere Anpassung vom Mikrocontroller verarbeitet werden können.

Voraussetzung dafür ist, dass die Referenzspannung des AD Wandlers auf 2,56 V eingestellt wird, siehe Beispiel davor. Eine gewisse Herausforderung liegt darin, die gemessenen dB-Werte auf einer 10-teiligen LED Kette auszugeben. Wir bedienen uns hier einer Look-Up-Tabelle. Das vereinfacht die Ausgabe stark, wir verwenden dafür ein komplettes 8 bit Port, sowie für die beiden letzten LEDs 2 bit eines weiteren Ports.

Man kann gerne die Werte übernehmen, welche in der Funktion der Wandlung „Spannung in dBm“ hinterlegt sind. Ich wollte diese berechnen, merkte aber bald, dass die Toleranzen des Analogteils des Mikrocontrollers signifikant sind. Um besser zu werden, habe ich mir mit einem Testprogramm die eingelesenen Zahlenwerte des AD Wandlers binär an der Diodenkette anzeigen lassen und aufgeschrieben. Der Wert – 80dBm ist in Übrigen illusorisch, denn die entsprechende LED leuchtet immer, das röhrt von dem Eigenrauschen des Messmoduls, dieses ist ja auch nur bis -74 dBm spezifiziert. Auch das ist schon eine bemerkenswert kleine Leistung! Zum Schluß noch: wenn man über 10 mW Leistung messen möchte, ist die Leistung mit geeigneten Dämpfungsgliedern herabzusetzen, eine Beschädigung des Messmoduls ist sonst nicht auszuschließen!

Hier kommt noch ein neues Zeichen: „\“. Wenn wir im C Code sehr lange Zeilen schreiben, müssen wir u.U. die Zeile umbrechen. In C ist das kein Problem nach einem Semikolon, innerhalb einer Anweisung aber schon, dafür brechen wir hier mit dem Zeichen „\“ um.

```

// Leistung mit AD 8307 messen und in 10 dB-Schritten
// an LED-Kette ausgeben fuer AT Mega8      27.12.2021 JK

#include <avr/io.h> // Einbinden von Einstellungen/Definitionen f Mikrocontroller
#include <util/delay.h>
#include <avr/interrupt.h>

// Portbelegung:

// PB3 MOSI Programmer
// PB4 MISO Programmer
// PB5 SCK  Programmer
// PC6 RESET Programmer

// PB0  gelbe LED 0 dB
// PB1  rote LED +10 dB

// PD0 - PD7  LEDs -80 bis - 10 dB

#define TABSIZE      100      // Groesse AD Sample Tabelle

uint8_t static volatile samples_ready;
uint8_t static volatile ad_index;
uint16_t static volatile ad_samples[TABSIZE];

ISR(ADC_vect) // Einlesen von Kanal 0
{
    ad_samples[ad_index] = ADCW;
    ad_index++;

    if(ad_index > TABSIZE - 1)
    {
        ad_index = 0;
        samples_ready = 1;
    }
}

// AD Wandler konfigurieren

void ad_init(void)
{
    ADMUX |= (1<<REFS1)|(1<<REFS0); // ADC Ref auf AREF geschaltet
    ADCSRA |= (1<<ADEN)|(1 << ADFR)| \
              (1 << ADIE)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0); // ADC eingeschaltet,
    // Freilaufend, Interrupt, Teiler / 128
    // Eingangstakt 8E6 / 128 / 13 (13 Zyklen fuer Wandlung)
    // T = 208 µs

    ADCSRA |= (1<<ADSC);           // eine ADC-Wandlung starten
}

```

```
uint8_t convert(unsigned int wert)
{
    // AD Werte in 10-dB Stufen umwandeln
    // Funktion gibt direkt die Indices fuer
    // Lookup_Tabelle aus!

    if(wert >= 0 && wert < 1)
        return 0;
    else if(wert >= 1 && wert <= 53)
        return 1;
    else if(wert >= 53 && wert <= 230)
        return 2;
    else if(wert >= 230 && wert <= 313)
        return 3;
    else if(wert >= 313 && wert <= 433)
        return 4;
    else if(wert >= 433 && wert <= 535)
        return 5;
    else if(wert >= 535 && wert <= 634)
        return 6;
    else if(wert >= 634 && wert <= 747)
        return 7;
    else if(wert >= 747 && wert <= 835)
        return 8;
    else if(wert >= 835 && wert <= 933)
        return 9;
    else if(wert >= 933 && wert <= 1023)
        return 10;
}
```


Projekt12: Wetterstation mit Internet-Anbindung und Solarstromversorgung

Hier geht es um ein sehr anspruchsvolles Projekt, es kann nicht an einem Nachmittag fertiggestellt werden. Es ist auch nicht nach einer Schablone abzuarbeiten, vielmehr lebt es von Variationen und eigenen Änderungen. Bei mir läuft dieses bereits mehrere Jahre und hat dabei schon viele Änderungen erfahren. Wir verwenden hier einen leistungsfähigeren Mikrocontroller, den AT Mega 1284 P, der leider auch etwas teurer ist. Er bietet jedoch neben 8 Analogeingängen sehr viel Speicherplatz. Für die Anbindung an das Internet reicht dessen Leistung nur unzureichend, ein 32 bit-Rechner ist dafür besser geeignet. Ich setze hier einen Raspberry PI ein, er läuft mit einem Linux-Betriebssystem. Solche Rechner sind bereits für sehr wenig Geld erhältlich und sie verbrauchen auch sehr wenig Energie! Wir werden bald sehen, wie wir das System mittels Solarzelle und Akku betreiben werden, das ist aber in unseren Breiten leider nur im Sommer möglich. Der Mikrocontroller und Raspberry PI sind miteinander gekoppelt. Ein zusätzliches Funkmodul kann Temperaturen von außen an die Zentrale übertragen. Die Messung von Luftdruck, Temperaturen und Luftfeuchte innerhalb des Hauses werden ermöglicht, die Überwachung der Solaranlage wird sichergestellt. Alle Daten können über einen eigenen WEB-Server, der auf dem Raspberry PI installiert wurde, abgerufen werden, auch über das Smartphone und weltweit! Wir programmieren den Mikrocontroller mit C, den Raspberry PI mit HTML und PHP. Das Vorgehen, soweit wir es benötigen, lernen wir in diesem Kapitel.

Wir benutzen viel Bekanntes, so den Analog-Digitalwandler, den Timer und den 1Wire-bus. Hinzu kommen I2C-Bus und serielle Schnittstelle. Den SPI-Bus kennen wir bereits von der Programmierschnittstelle, wir schließen dort zusätzlich ein Funkmodul an.

An dieser Stelle planen wir das Grundgerüst des Programms auf dem Mikrocontroller. Dieses soll durch das Funkmodul synchronisiert werden und alle 10 Minuten alle Messwerte verarbeiten, in physikalische Größen umwandeln und an den Raspberry Pi übertragen.

Auf dem Raspberry PI läuft der WEB Server. Er verarbeitet die letzten 12 Einträge der gespeicherten Messwerte. Diese werden auf einer WEB-Seite angezeigt. Da wir ein Betriebssystem haben, können wir auch Emails versenden, wenn wir kritische Situationen erkennen. Wir können auch Schaltvorgänge von außerhalb steuern, was ich in meinem Projekt nicht machen werde. Die Software auf dem Mikrocontroller ist in der Lage, bei Unterspannung des Solarakkus das Stromnetz zuzuschalten.

Hier ist zuerst eine Stückliste mit den Komponenten, welche wir für die Realisierung des WEB Servers benötigen. Es werden mehrere Stücklisten folgen, jeweils zu den danach beschriebenen Projekten:

Stückliste Web Server, wir rechnen mit etwa 70 Euro:

Anz	im Plan	Wert	Reichert Best-Nr	Bemerkungen
1		Raspberry PI	Raspberry PI Zero WH	
1		Stecker Netzgerät	HNP 12-120V2	
1		SD Karte	INTENSO 3424480	
1		Gehäuse für Server	BOPLA M 223 G	
1		5V Netzgerät mit µUSB	div. Lieferanten	Siehe Text

Um diese umfangreiche Information sortieren zu können, fange ich als erstes mit der Konfiguration des Servers an.

Vielleicht haben wir hier ein 5V Netzgerät herumliegen, dieses sollte einen Mikro-USB Stecker haben. Oder wir basteln uns einen Adapter für unser Labornetzgerät, sodass wir es über den Mikro-USB Anschluss mit dem Raspberry PI verbinden können. Ein Netzteil für den Raspberry PI zu kaufen, lohnt nicht unbedingt, wir betreiben ihn ja später über die Zentrale über einen DC-DC-Wandler. Aber da bei unserem Einstieg die Zentrale noch nicht existiert, benötigen wir für den Minicomputer eine Stromversorgung. Das Servergehäuse bestellen wir bereits jetzt schon, wir werden es aber erst später brauchen.

Uns liegt jetzt der Raspberry PI Nano, eine SD-Karte und die geeignete Stromversorgung von 5V mit Mikro-USB Anschluss vor. Wir benutzen den „Raspberry PI Imager“ (22), welcher für Windows, Ubuntu und Mac OS verfügbar ist. Aus Bequemlichkeit, da er bei mir schon installiert ist, nehme ich meinen Windows 10 Rechner. Mein Rechner verfügt über einen externen Card-Reader für die SD Karte. Wenn nicht bereits erfolgt, laden wir aus dem Web den Raspberry PI Imager herunter. In den Card Reader schieben wir die Micro-SD Karte und folgen der Anleitung des Imagers. Wir kommen mit der Version des Raspberry OS ohne den Desktop aus. Wenn die SD-Karte fertig ist, lassen wir sie im Card-Reader. Die Windows Warnungen wegen der Formatierung ignorieren wir an dieser Stelle. Wir erhalten ein neues Laufwerk mit der Bezeichnung „boot“. Hier fügen wir eine Datei namens ssh ein (ohne Dateierweiterung!). Das kann man mit einem beliebigen Editor machen, man erzeugt eine Datei mit 1 Leerzeichen und speichert diese in „boot“ ab. Wir brauchen diesen Schritt, um später per ssh auf den Raspberry PI zugreifen zu können. Denn wir werden diesen „headless“ betreiben, d.h. ohne Bildschirm und Tastatur (das wäre zwar möglich, ist für unsere Zwecke aber unnötig). Wenn wir den Zero mal genau betrachten, werden wir bemerken, dass dieser keine Netzwerkbuchse hat. Er kann nur „wireless“ mit dem Netzwerk kommunizieren. Es ist im Übrigen wichtig, dass wir den richtigen Raspberry PI Zero WH kaufen! Wie wir vermuten, kann der Rechner sich nicht in unser WLAN einloggen, da er keinerlei Zugangsdaten kennt. Unsere Aufgabe ist nun, eine Datei auf dem Laufwerk „boot“ auf der SD Karte anzulegen, diese enthält die erforderlichen Zugangsdaten. Sie nennt sich „wpa_supplicant.conf“.

Ihr Inhalt:

```
country=DE
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
network={
    ssid="WLAN-Name"
    psk="Passwort"
}
```

Wir können ab jetzt die SD-Karte in den Raspberry PI stecken, auf dessen linker Seite ist der entsprechene Kartenslot. Die Stromversorgung ist unten rechts die 2. Mikro-USB Buchse. Wenn wir 1 bis 2 Minuten warten, können wir in der Software unseres Router ein neues Netzwerkgerät finden (Raspberry PI), wir schreiben dessen IP-Adresse auf. Inzwischen haben wir „putty“ aus dem Internet heruntergeladen und auf unserem PC installiert. Nach Aufruf des Programms sollten wir eine Verbindung mit Port 22 und der gemerkten IP-Adresse herstellen können. Möglicherweise genügt anstatt der IP Adresse auch der Name „Raspberry PI“. Putty fragt nach der Speicherung eines Schlüssels, wir antworten darauf mit

dem Button „Ja“. Danach fragt die Shell nach einem Benutzernamen. Wir tippen ein: pi. Danach wird nach einem Passwort gefragt, dieses ist raspberry. Wir sollten dieses sofort nach dem ersten Einloggen ändern, dazu tippen wir ein: „passwd“. Wir werden nach dem alten Passwort gefragt und können dann ein neues Passwort eingeben, das ganze 2 Mal. Ich wäre vorsichtig mit Umlauten, da möglicherweise der Raspberry PI noch auf Englisch lokalisiert ist. Diese Änderung nehmen wir dann gleich vor. Nach einem Neustart (wenn er nicht automatisch erfolgt, durch „sudo reboot“) können wir Putty schließen, da die Verbindung abbricht. Wir müssen uns neu verbinden. Nach Eingabe des Benutzers pi und dem neuen Passwort geben wir ein: „raspi-config“. Hier können wir unter „Localisation options“ L1 und L2 richtig einstellen, für Deutschland. Bei L1 wählen wir de_DE.UTF8. Wenn wir schon mal in raspi-config sind, können wir auch gleich die serielle Schnittstelle aktivieren, wir brauchen diese später. Wir kommen über das Feld „back“ wieder ins Hauptmenü. Die Maus geht da nicht, aber mit der TAB-Taste und der UP/Down-Taste kommt man voran. Wir wählen „Interfache Options“ dann P6 (serial Port..) verneinen die Login-shell und schalten den Port frei. Leider müssen wir später noch ein paar Änderungen vornehmen, da der hier eingestellte serielle Port (miniUSART) sehr unzuverlässig arbeitet, er ist von der CPU-Taktfrequenz abhängig und für uns nicht brauchbar. Wir verwenden daher den seriellen Port PL011, der unter /dev/ttAMA0 von der Software angesprochen wird.

Wir rebooten nun den Raspberry.

Zuerst führen wir aus: „sudo apt-get update“ und anschließend „sudo apt-get upgrade“. Diese Schritte benötigen einiges an Zeit, daher etwas Geduld.

Dann installieren wir einen WEB-Server (23), wir nehmen „lighttpd“. Die Anleitung zur Installation finden wir im WWW. Es wird mit „sudo apt-get install lighttpd“ durchgeführt.

Sudo brauchen wir, um Systemverwalter-Rechte zu bekommen.

Ist der Server installiert, können wir ihn mit einem beliebigen Browser aufrufen, man gibt dazu einfach die IP Adresse in die Adresszeile des Browsers ein. Lighttpd sollte sich sogleich mit einer Eröffnungsseite präsentieren. Wir vergewissern uns, ob es ein Verzeichnis mit Name /var/www/html gibt. Wenn nicht, legen wir es an. Wir müssen sudo voranstellen: „sudo md /var/www/html“. Es ist sehr nützlich, sich gängige UNIX-Anweisungen zu erarbeiten, wir werden diese künftig verwenden!

Wie bereits erwähnt, benötigen wir auch PHP. Die Installation erfolgt ebenso mit „sudo apt-get install“, wir müssen dort die aktuelle Version von php angeben, bei mir war das 7.3. Der Vorgang dauert jetzt wesentlich länger, als die Installation des WEB Servers. Auch müssen wir anschließend die Konfigurationsdatei von lighttpd bearbeiten, was ich ziemlich umständlich fand (und der Erfolg sich nach erst nach längerer Zeit einstellte..). Im WWW findet man gute Beispiele für die Konfiguration.

Die Konfiguration findet man unter /etc/lighttpd/lighttpd.conf, für deren Editierung benötigt man Root-Rechte. Am Besten eine Kopie der vorliegenden lighttpd.conf machen! Zum Editieren kann man gut „Nano“ verwenden (sudo nano /etc/lighttpd/lighttpd.conf)..

Mit dieser Konfiguration hatte ich letztendlich Erfolg:

```
server.modules = (
    "mod_indexfile",
    "mod_access",
    "mod_alias",
    "mod_redirect",
)

server.document-root      = "/var/www/html"
server.upload-dirs        = ( "/var/cache/lighttpd/uploads" )
server.errorlog            = "/var/log/lighttpd/error.log"
server.pid-file            = "/var/run/lighttpd.pid"
server.username             = "www-data"
server.groupname            = "www-data"
server.port                  = 80

# strict parsing and normalization of URL for consistency and security
# https://redmine.lighttpd.net/projects/lighttpd/wiki/Server_http-parseoptsDetails
# (might need to explicitly set "url-path-2f-decode" = "disable"
# if a specific application is encoding URLs inside url-path)
server.http-parseopts = (
    "header-strict"           => "enable", # default
    "host-strict"             => "enable", # default
    "host-normalize"          => "enable", # default
    "url-normalize-unreserved"=> "enable", # recommended highly
    "url-normalize-required"  => "enable", # recommended
    "url-ctrls-reject"        => "enable", # recommended
    "url-path-2f-decode"      => "enable", # recommended highly (unless breaks app)
    "#url-path-2f-reject"     => "enable",
    "url-path-dotseg-remove"  => "enable", # recommended highly (unless breaks app)
    "#url-path-dotseg-reject" => "enable",
    "#url-query-20-plus"      => "enable", # consistency in query string
)

index-file.names           = ( "index.php", "index.html" )
url.access-deny            = ( "~", ".inc" )
static-file.exclude-extensions = ( ".php", ".pl", ".fcgi" )

compress.cache-dir          = "/var/cache/lighttpd/compress/"
compress.filetype            = ( "application/javascript", "text/css", "text/html",
"text/plain" )

# default listening port for IPv6 falls back to the IPv4 port
include_shell "/usr/share/lighttpd/use-ipv6.pl" + server.port
include_shell "/usr/share/lighttpd/create-mime.conf.pl"
include "/etc/lighttpd/conf-enabled/*.conf"

#server.compat-module-load  = "disable"
server.modules += (
    "mod_compress",
    "mod_dirlisting",
    "mod_staticfile",
)
```

Man testet das Ganze mit Aufruf der IP-Adresse im Browser, ergänzt durch /phpinfo.php. Eine umfangreiche Seite mit Einstellungen von PHP wird erscheinen. Falls nicht, ist die lighttpd-Konfiguration falsch!

Wenn das abgeschlossen ist, kümmern wir uns noch um die serielle Schnittstelle: Wir rufen auf: sudo nano /boot/config.txt. Ganz unten tragen wir die Zeilen ein:

```
enable_uart=1  
dtoverlay=disable-bt
```

Den Rest der Datei lassen wir unberührt, denn fehlerhafte Einträge können den Raspberry beim Booten hindern!

Zum Schluss müssen wir noch den Bluetooth Treiber ausschalten, denn PL011 ist mit Bluetooth verbunden – leider verlieren wir mit unserer Änderung den Zugriff des Zeros auf Bluetooth, was uns aber nicht weiter stört.

Wir geben in die Kommandozeile ein.

```
sudo systemctl disable hciuart
```

Und booten den Raspberry danach. Obige Eingabe müssen wir nur einmal machen.

Wir legen jetzt den Raspberry PI Zero zur Seite und beschäftigen uns mit der weiter benötigten Hardware. Zuerst bauen wir die **Solarstromversorgung** auf.

Stückliste für Solarstromversorgung, wir rechnen mit etwa 120 Euro:

Anz	im Plan	Wert	Reichelt Best-Nr	Bemerkungen
1	G1	Solarpanel 30 W	OFF 3-01-001530	
1	G2	Solarbatterie	WP 24-12	
1	P1	Laderegler	SOLARLADER	
		Ausreichend Leitung		2,5 mm ²

Wir verbinden Solarzelle, Solarakku und Laderegler, wie es in der Anleitung des Ladereglers vorgeschrieben ist. Das Solarmodul platzieren wir am Besten auf unserem Dach, es wird nach Süden ausgerichtet. Akku und Laderegler befinden sich innerhalb der Wohnung, sie werden mit geeigneten Kabeln mit 2,5 mm² Querschnitt verbunden. Das gleiche gilt auch für die Solarzelle, deren Anschluss witterfest sein muß! Später applizieren wir in der Nähe des Ladereglers eine Schaltung, welche die Betriebsdaten der Solaranlage ermitteln wird.

Wenn wir mit dem Aufbau fertig sind, haben wir 12 Volt vorliegen. Bitte diese entsprechend gegen Kurzschluss sichern, unsere Anwendung kommt mit viel weniger als 1 Ampere gut aus! Für den Raspberry PI und die weitere Elektronik brauchen wir 5 V und 3,3 V. diese erzeugen wir später aus den 12 V, dafür verwenden wir Schaltregler, welche einen guten Wirkungsgrad versprechen.

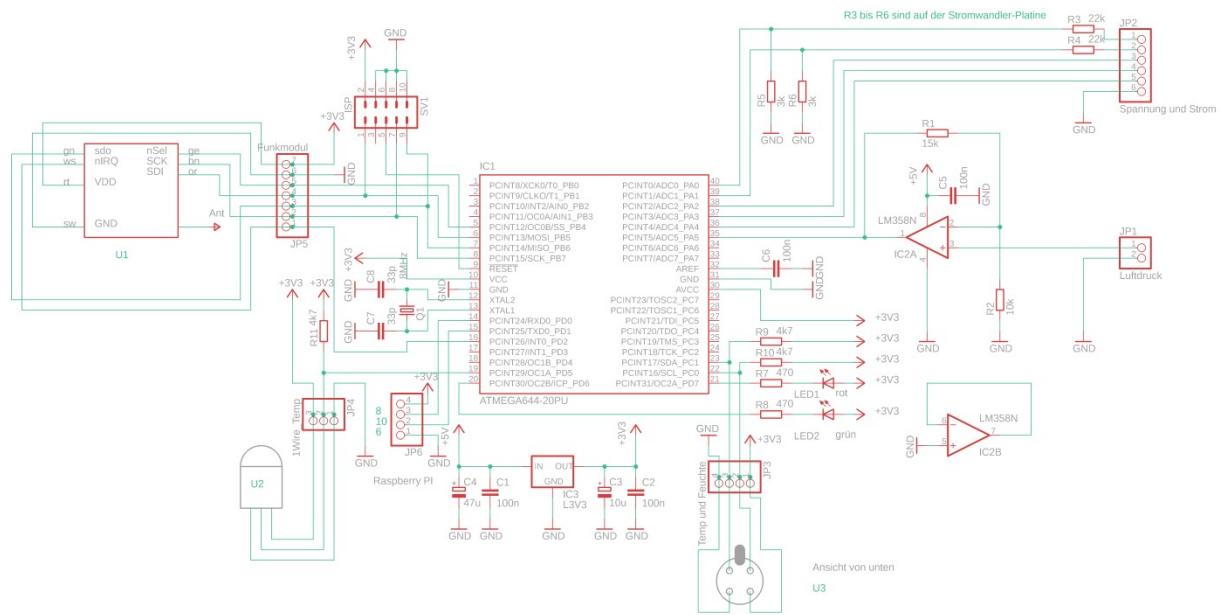
Nun können wir die **Steuerung der Wetterstation** aufbauen, das kann auf einer Lochrasterplatine geschehen. Diese muß nicht besonders groß sein, es genügen ca. 90 mal 50 mm.

Stückliste für Steuerung Wetterstation, wir rechnen mit etwa 60 Euro:

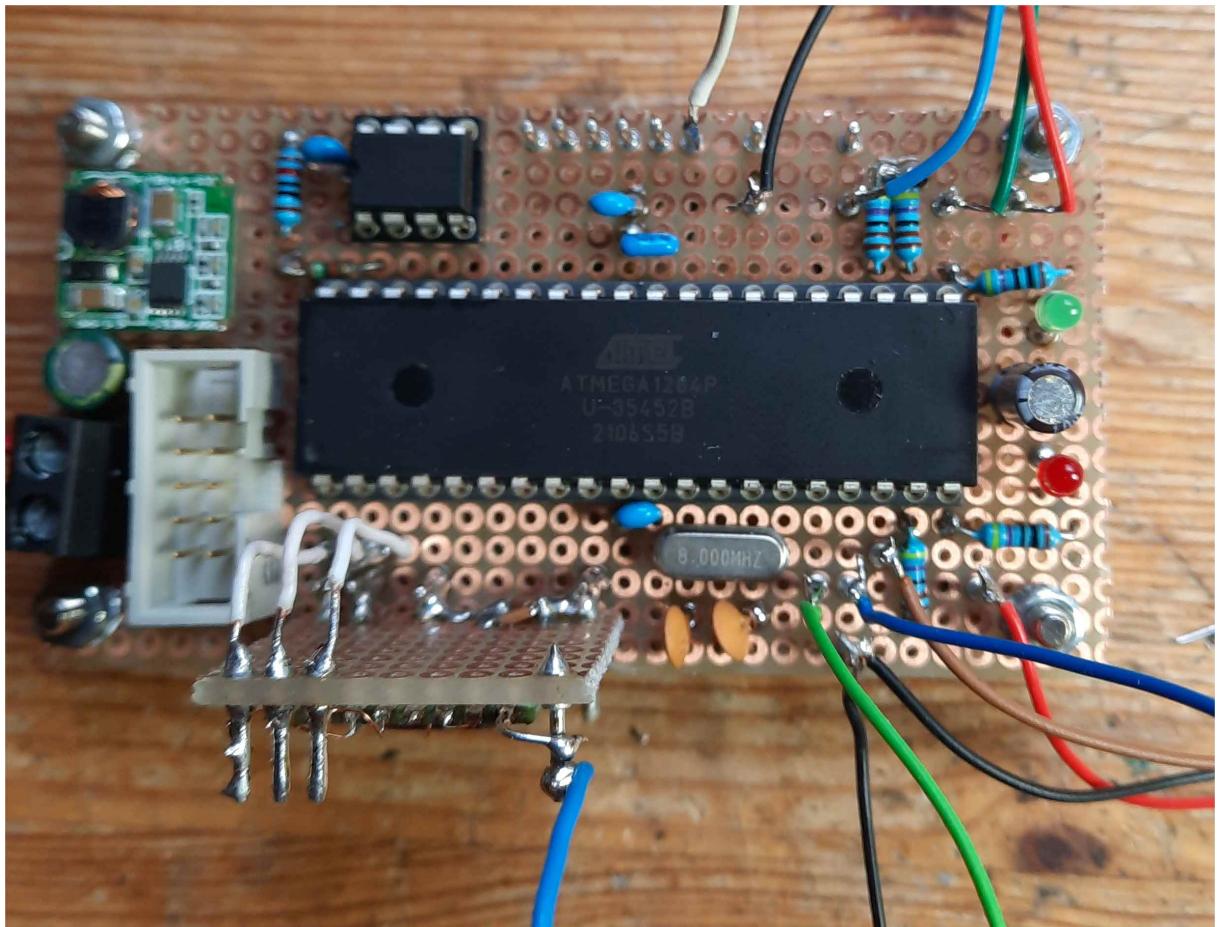
Anz	im Plan	Wert	Reichelt Best-Nr	Bemerkungen
1	U1	Funkmodul 430 MHz	810302	Pollin Elektronik
1	IC1	Mikrocontroller	ATMEGA 1284P	
1	Q1	Quarz 8 MHz	8,0000-HC49U-S	Pro Stück
2	C7,C8	Kerko 33 pF	KERKO 33P	Pro Stück
1	IC3	Spannungsregler 5/3,3V	TBA 1-0510	
1	U2	Temperatursensor	DS 18B20	
1	U3	Feuchte/Temp-Sensor	HYT 939	
1		Luftdrucksensor		Siehe Text
1	IC2	Operationsverstärker	LM 358 DIP	
1	SV1	ISP Buchse	WSL 6G	
1	C4	Elko 47 uF	SU-A 47U 35	
1	C3	Elko 10 uF	NHG-A 10U 50	
4	C1,2,5,6	Kerko 100 nF	Z5U-2,5 100N	Pro Stück
2	R7, R8	Widerstand 470 Ohm	METALL 470	Pro Stück
3	R9 – R11	Widerstand 4,7k	METALL 4,70K	
1	R1	Widerstand 15k	METALL 15,0K	
1	R2	Widerstand 10k	METALL 10,0K	
1	LED1	LED 3mm rot	LED 3MM 2MA RT	
1	LED2	LED 3mm grün	LED 3MM 2MA GN	

1		Leiterplattenklemme	DG127 5.08-2	2 polig
100		Lötnägel 1 mm	RTM 1-100	Vorrat

Schaltplan:



Ein Foto von meinem Aufbau (ich verwende eine andere ISP-Buchse und einen anderen 3,3V Spannungsregler):



Erklärung: Das Funkmodul habe ich auf ein Stück Lochrasterplatine gesetzt, denn es ist sehr empfindlich, denn seine Lötanschlüsse können schnell beschädigt werden. Dieses Platinchen wurde senkrecht auf die Mutterplatine gesetzt, mit Lötnägeln kontaktiert und mit Stabilit zusätzliche geklebt. Der blaue Draht ist die Antenne! Den Operationsverstärker mit seinen 2 Widerständen benötigen wir nur, wenn ein analoges Messmodul für den Luftdruck verwendet wird. Das Modul, welches ich hier verwendet habe, wird nicht mehr hergestellt. Die Lötreihe oben in der Mitte ist die Verbindung zu der Spannungs- und Strom-Messplatine, die wir später behandeln. Der Feuchte und Temperatursensor ist oben rechts mit der schwarzen, roten, grünen und blauen Leitung angeschlossen. Der andere Temperatursensor unten rechts mit rot, braun und schwarz. Links daneben ist die Verbindung zum Zero mit schwarz, grün und blau.

Um das ganze System betreiben zu können, benötigen wir 4 weitere Platten.

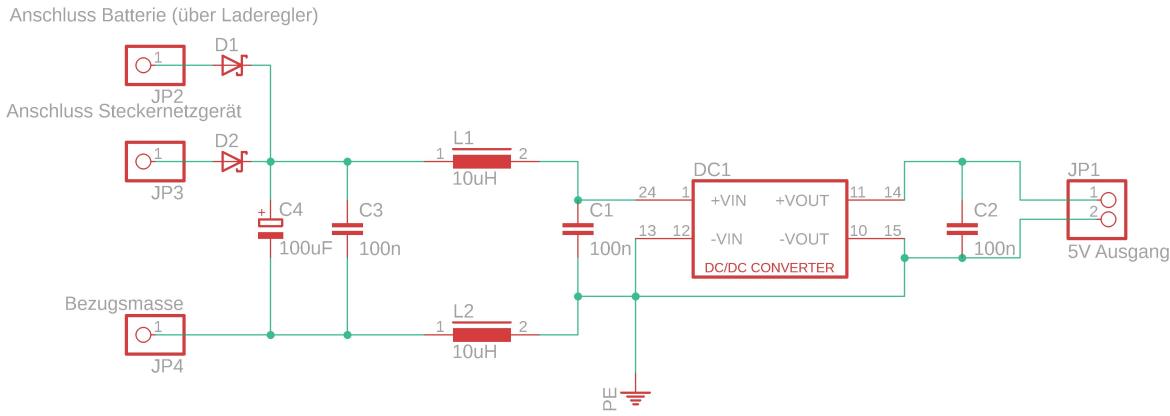
Die erste, hier als **5V Regler** bezeichnet, ist mit dem DC/DC- Wandler und 2 Dioden bestückt, sie liefert die 5 Volt für die weiteren Platten. Hier sind noch zusätzliche Bauteile für das Gehäuse mit aufgeführt.

Stückliste für zusätzliche Bauteile, wir rechnen mit etwa 35 Euro:

Anz	im Plan	Wert	Reichelt Best-Nr	Bemerkungen
2	D1,D2	,Schottky Diode	SB 340 DIO	Pro Stück
2	L1,L2	Drossel 10uH	MESC 10 μ	Pro Stück
3	C1-C3	Kerko 100n	Z5U-2,5 100N	Pro Stück
1	C4			

1	DC1	DC DC Wandl.12/5V	TMR 6-2411WI	
1	JP1	Hohlsteckerbuchse	DELOCK 89910	
2		Einbaubuchse	BIL 20 SW/RT	Rot und schwarz
1		SUB-D Buchse 15 p	D-SUB BU 15	
1		SUB-D Stecker 15 p	D-SUB ST 15	
1		Leitungs-Sicherung	STV PTF/80A	
1		Feinsicherung 5*20	ESKA 521.524	

Ein Schema ist hier:



Wir bauen es deshalb auf einer extra Platine auf, um die restlichen Schaltungsteile von 12V zu schützen. L1 und L2 dienen der Störstrahlungs-Unterdrückung, wir wollen ja nicht die Solarleitungen als Sendeantenne missbrauchen. Die Spulen sollten 2 – 3 A aushalten, die beiden Dioden ebenso. Durch die beiden Dioden wird der Spannungswandler jeweils auf die höhere Eingangsspannung geschaltet. Wenn das Steckernetzteil aktiviert wird, dann wird der Wandler dadurch versorgt, andernfalls durch die Solarbatterie.

Der abgesetzte Temperaturfühler, welcher die Außentemperatur überträgt, wird auf einer kleineren Lochrasterplatine aufgebaut.

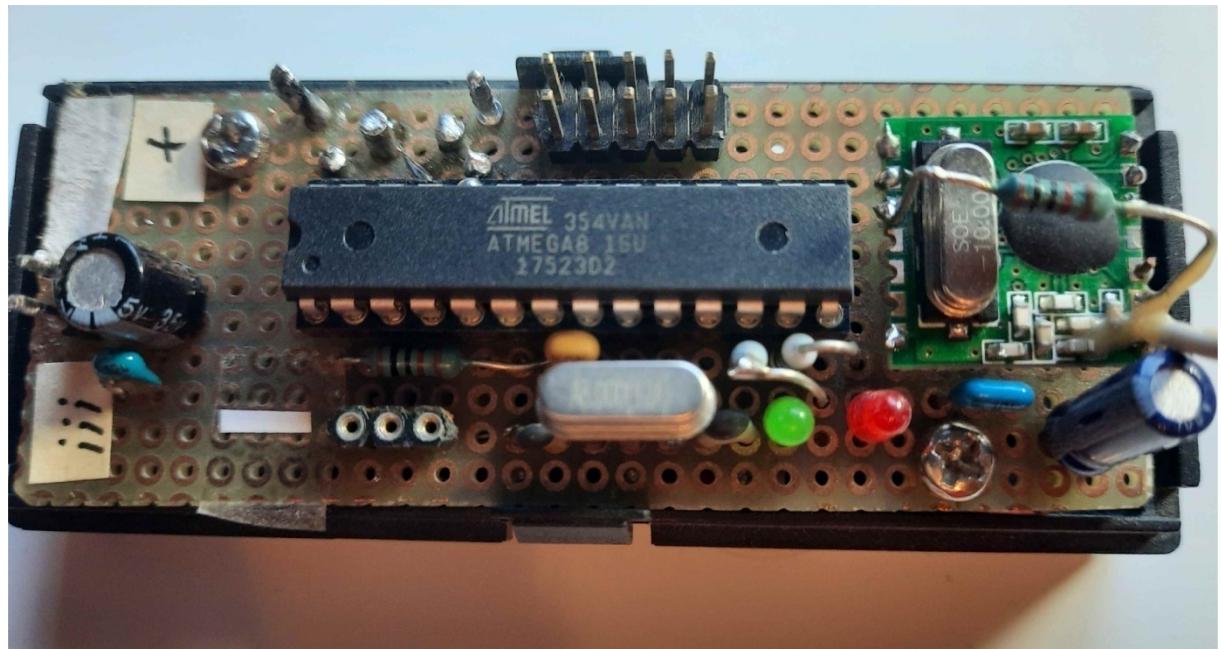
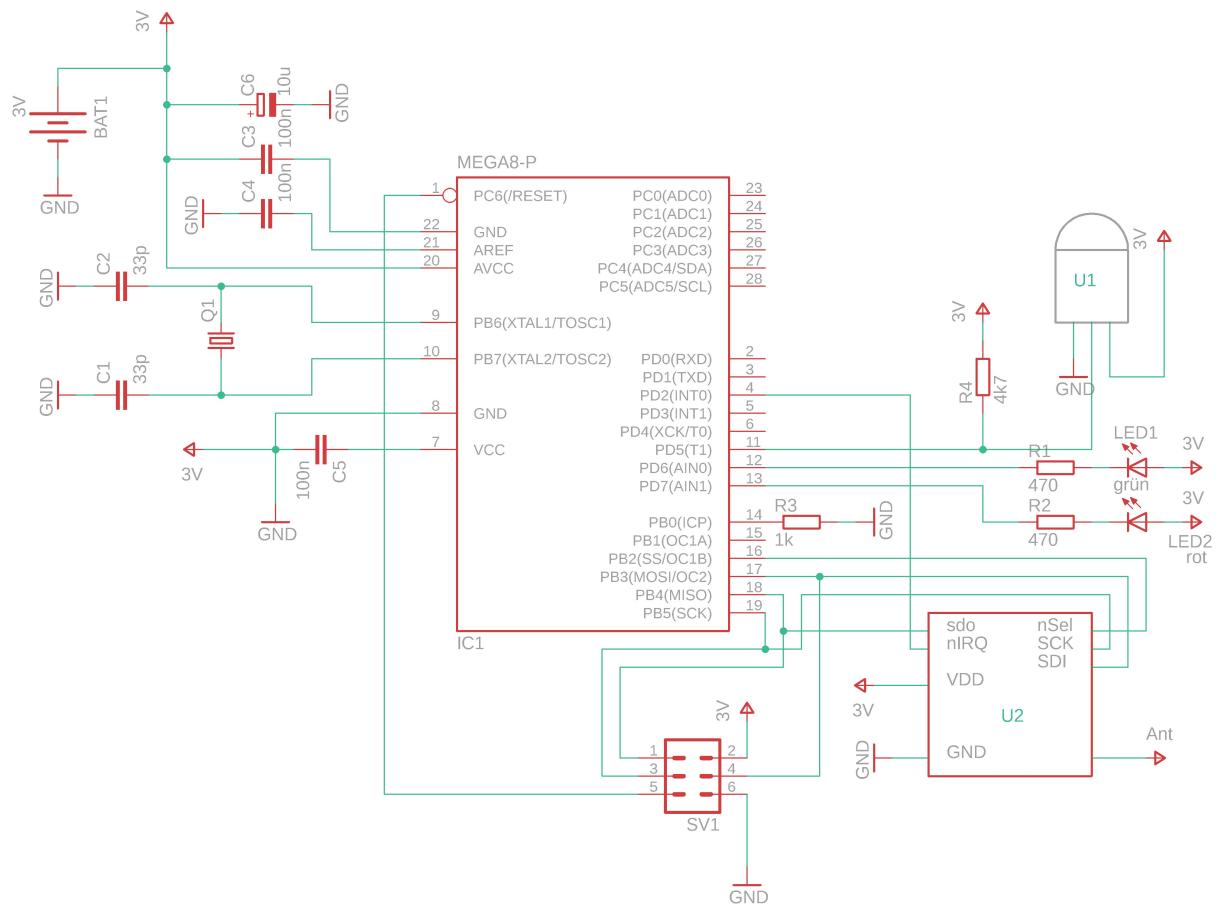
Das Modul, welches die Außentemperatur misst und als **Außensensor** bezeichnet wird, kann diese per Funk an unsere Zentrale übertragen. Das habe ich bei mir in dieser Weise implementiert; das Modul ist in einem Gartenhaus trocken verwahrt, nur der Temperatursensor ist im Freien. Das Modul wird von 2 AA-Batterien versorgt, dank geschickter Programmierung kommt es damit über 1 Jahr aus!

Stückliste für Außensensor, wir rechnen mit etwa 20 Euro:

Anz	im Plan	Wert	Reichelt Best-Nr	Bemerkungen
1	U2	Funkmodul 430 MHz	810302	Pollin Elektronik
1	IC1	Mikrocontroller	ATMEGA 8A-PU	
1	Q1	Quarz 8 MHz	8,0000-HC49U-S	Pro Stück
2	C1,C2	Kerkos 33 pF	KERKO 33P	Pro Stück
2	BAT1	2 AA 1,5V	ANS 5015663	4 Pack
3	C3-C5	Kerko 100n	Z5U-2,5 100N	Pro Stück
1	C6	Elko 10 uF	NHG-A 10U 50	
1	U1	Temperatursensor	DS 18B20	U2
1	SV1	ISP Buchse	WSL 6G	

2	R1, R2	Widerstand 470 Ohm	METALL 470	Pro Stück
1		Widerstand 4,7 k	METALL 4,70K	
1		Widerstand 1 k	METALL 1,00K	
1	LED2	LED 3mm rot	LED 3MM 2MA RT	
1	LED1	LED 3mm grün	LED 3MM 2MA GN	

Schaltplan:

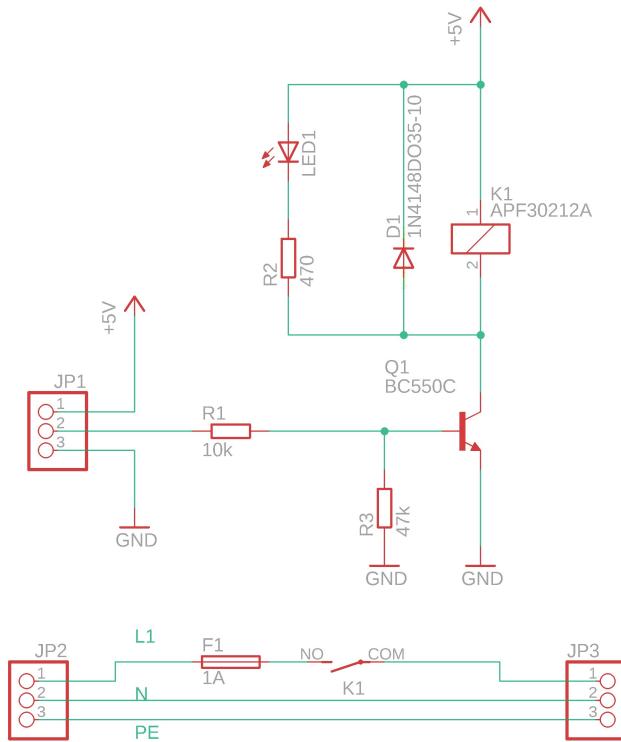


Die Zentrale bezieht ihre Energie aus dem Solar-Akku, dieser wird im Sommer durch das Solarpanel immer ausreichend geladen sein. Im Herbst wird das nicht mehr der Fall sein, eine zusätzliche Netzversorgung wird notwendig sein. Dafür verwenden wir ein handelsübliches 12 V Steckernetzteil. Dieses wird über eine **Relais-Schaltung** automatisch zugeschaltet, wenn die Spannung des Akkus unzureichend wird. Da wir mit diesem Relais Netzspannung schalten werden, müssen wir auch hier eine extra Platine in einem separaten Gehäuse verwenden.

Stückliste für Relaissteuerung, wir rechnen mit etwa 28 Euro:

Anz	im Plan	Wert	Reichelt Best-Nr	Bemerkungen
1		Gehäuse	BOPLA SE432 DE	
1	K1	Relais	G5NB1AEDC5-N-A	
1	Q1	Transistor BC550	BC 550C	
1	R1	Widerstand 10k	METALL 10,0K	
1	R2	Widerstand 470	METALL 470	
1	R3	Widerstand 47k	METALL 47,0K	
1	D1	Diode 1N4148	1N 4148	
1	F1	Feinsicherung 1A	ESKA 525.617	
1		Sicherungshalter	PL OGN-22,5	
1	LED1	LED 5mm rot	L-7113SRD-J4 KB	

Schaltplan:



Netzspannung: besonders sorgfältig arbeiten, Isolationsabstände einhalten!

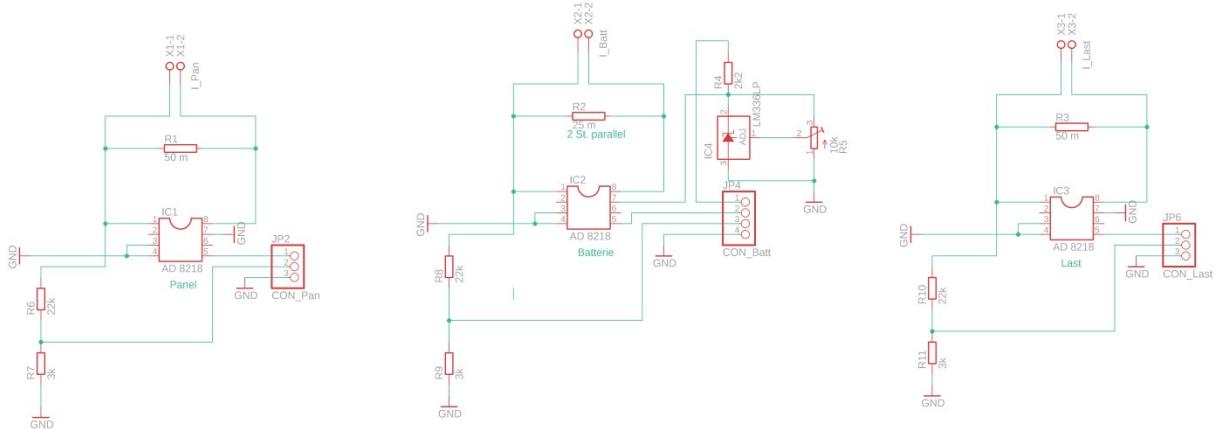
Zuletzt wird die Schaltung für die Erfassung von Strom und Spannung der Solaranlage erläutert, wir nennen dieses **U/I Messmodul**.

Stückliste für U/I Messmodul, wir rechnen mit etwa 38 Euro:

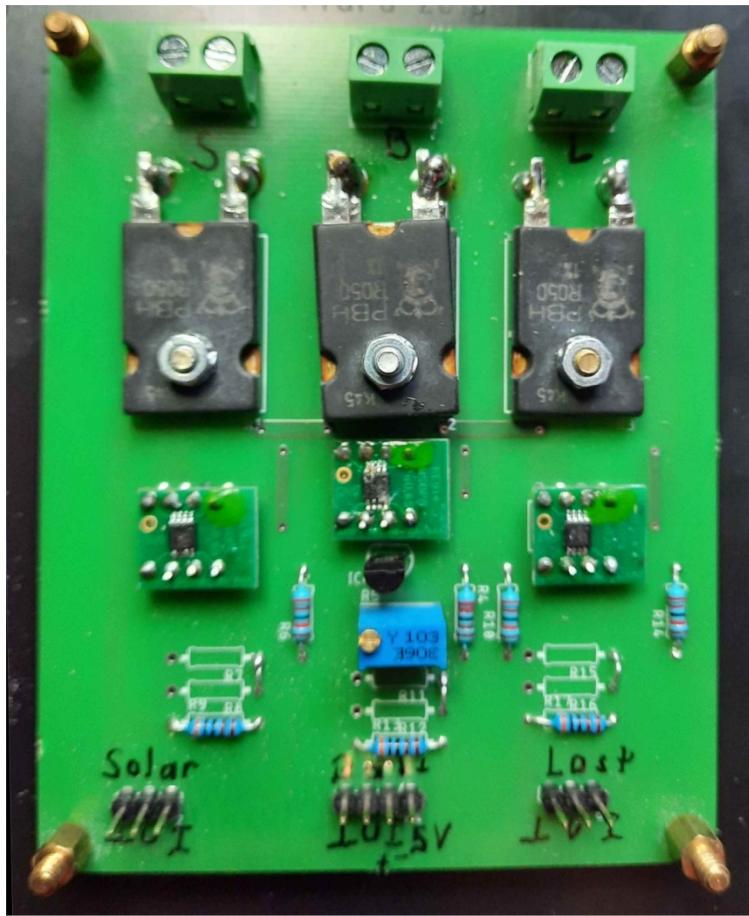
:

Anz	im Plan	Wert	Reichelt Best-Nr	Bemerkungen
4	R1 – R3	Wid. 50 Milliohm	ISA PBH-R050-F1	R2= 2 St. paralell
3	R6,8,10	Widerstand 22 k	METALL 22,0K	
3	R7,9,11	Widerstand 3k	METALL 3,00K	
1	R4	Widerstand 2,2k	METALL 2,20K	
1	R5	Trimmpoti 10k	64Y-10K	
1	IC4	U-Referenz	LM 336-Z2,5	
3	IC1 – IC3	I-U Wandler	AD 8218 B	Sehr klein!
3		MSOP8 0,65mm	RE 914	Adapterplatinen
3	X1 – X3	2 pol. Klemme	DG127 5,08-2	

Schaltplan:

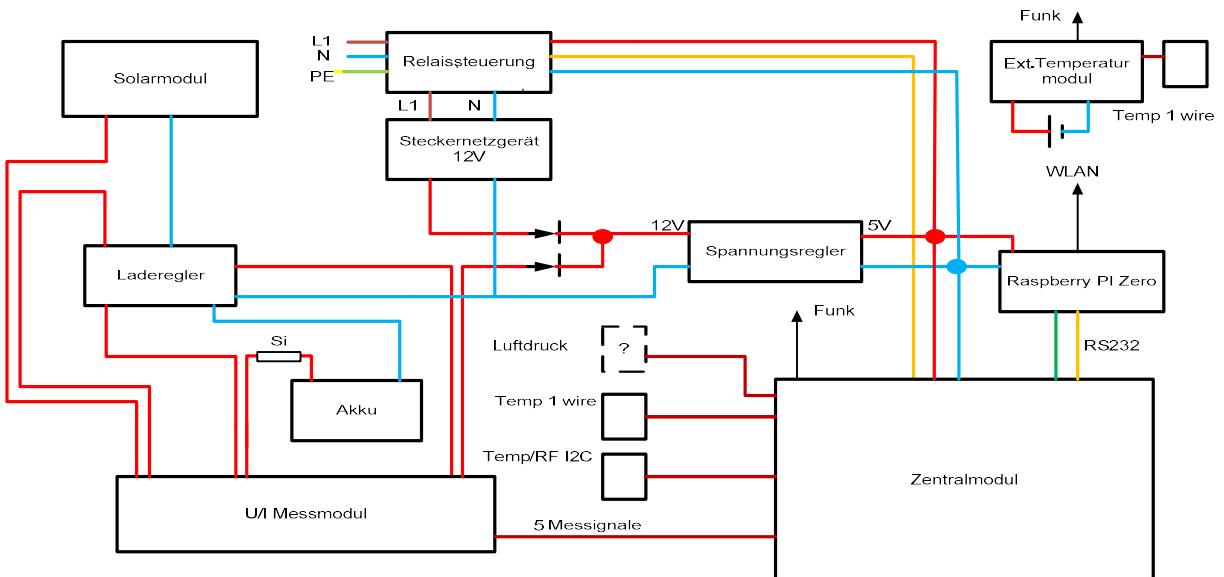


Auch diese Schaltung wird auf einer Platine aufgebaut, hier habe ich eine Platine anfertigen lassen, diese wird in ein separates Gehäuse eingebaut und nahe des Ladereglers platziert. Das Auflöten der AD 8218 auf die Adapterplatinen ist sehr schwierig, gelingt aber, wenn man diese Bauteile an 2 gegenüberliegenden Anschlüssen durch Anlöten fixiert, danach die Lötstellen mit Flux behandelt (im Elektronik-Fachhandel erhältlich) und dann alle Anschlüsse zusammen verzinnnt. Danach nochmal fluxen und mit Hilfe von Entlötlitzte alle Kurzschlüsse beseitigen. Zügig arbeiten, um unnötige Wärmeverbelastung zu vermeiden. Eine ruhige Hand, Pinzette, sichere Fixierung der Platine und eine starke Lupe sind nützlich. Da ich nur eine Referenzspannungsquelle mit 2,5V zur Hand hatte, muss ich nun den Stromausgang des Wandlers der Batterie mit einem Spannungsteiler 2:1 herunterteilen, hierfür sind 2 Widerstände von 10 Kilohm brauchbar. In der Software muss ich natürlich das Teilverhältnis mitberücksichtigen. Auch der Stromausgang für die Last habe ich 2:1 geteilt, somit kann ich auch bis 5 Ampere messen.



Bestückte Platine U/I Messmodul. Gut sind die großen Messwiderstände zu erkennen.

Wenn alles getan ist, verdrahten die aufgebauten Komponenten nach folgendem Schema:



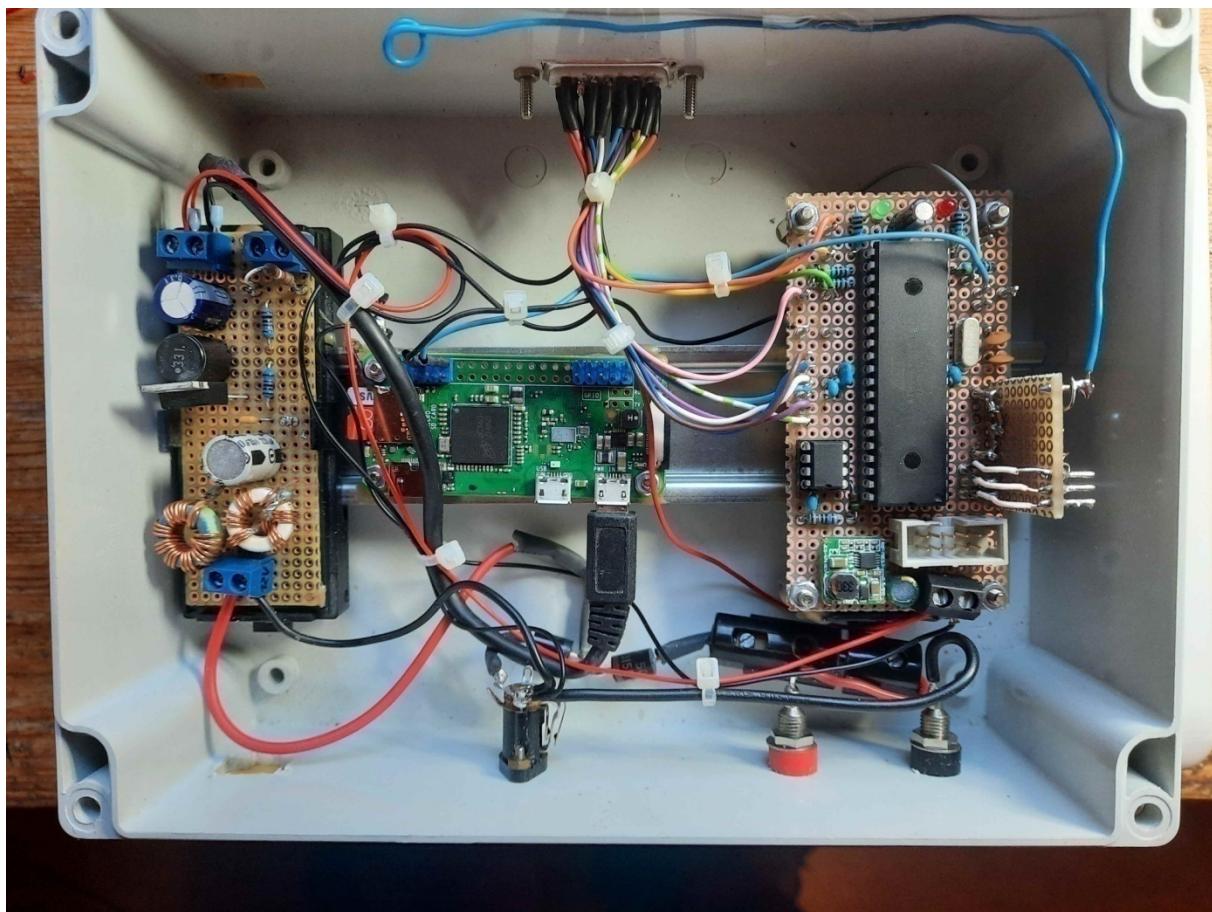


Bild des kompletten Aufbaus der Zentrale. Links der Spannungswandler 12V/5V, den ich hier selbst entworfen habe, dann folgend der Raspberry PI Zero, danach das Modul der Zentrale. Die SUB-D Buchse im hinteren Teil ist der Anschluss für die Sensoren und das U/I Modul.

Software Teil 13:

Es fehlt noch die Software für die Zentrale, den externen Temperaturgeber und die WEB-Seite.

Wir fangen mit der Zentrale an. Dafür benötigen wir 5 C-Programme und 6 Headerfiles:

main.c	main.h
rfm12.c	rfm12.h
1wire.c	rfm12_config.h
TWI_Master.c	TWI_Master.h
serial.c	serial.h
	General.h

Die Dateien mit Name rfm stammen aus der Bibliothek von Peter Fuhrmann von „das Labor“ (24) und dienen als Treiber für das Funkmodul. Sofern die Lizenz (25) eingehalten wird (GPLv2+), kann diese Software verwendet werden.

Die Verzeichnisstruktur ist genauso vorzunehmen, wie es dort beschrieben ist, also die „kleine“ rfm12.c und rfm12.h zusammen mit den oben genannten Dateien im Hauptverzeichnis, es ist ein Unterverzeichnis mit Namen rfm12lib zu erstellen, in diesem befinden sich die beiden „großen“ rfm12c und rfm12.h, sowie ein Unterverzeichnis namens include. In rfm12_config sind wesentliche Einstellungen des Funkmoduls hinterlegt, u.a die Frequenz, hier hatte ich 430,8 MHz eingestellt. Später müssen wir auch bei dem Geber des Außensensors die gleiche Konfiguration verwenden! Es ist hier wenig zu tun, man muss nur die richtige Frequenz in der Datei eintragen. Hier folgt ein Beispiel der Konfiguration.

1wire.c kennen wir bereits, an dieser Stelle wird jedoch eine stark vereinfachte Version des Programmes verwendet, hier müssen wir das richtige Port und den passenden Pin für den Temperaturfühler einsetzen. Auch serial.c werde ich hier aufführen, im WWW kann man unzählige Programmversionen für diese Aufgabe finden! . TWI_Master dient zur Kommunikation mit dem Luftfeuchte/ Temperatursensor, es kann direkt aus der Quelle verwendet werden, denn die Anschlüsse des I2C-Ports sind bei dem Mikrocontroller festgelegt.

In General.h finden wir einige Vereinfachungen für unsere Programme, in main.h alle Include, sowie die Deklaration der verwendeten Funktionen.

Im makefile sind sämtliche C-Quellen einzutragen.

Hier folgen die entsprechenden Dateien:

rfm12_config.h

```
***** RFM 12 library for Atmel AVR Microcontrollers *****
*
* This software is free software; you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published
* by the Free Software Foundation; either version 2 of the License,
* or (at your option) any later version.
*
* This software is distributed in the hope that it will be useful, but
* WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
* General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this software; if not, write to the Free Software
* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
* USA.
*
* @author Peter Fuhrmann, Hans-Gert Dahmen, Soeren Heisrath
*/
***** C O N F I G U R A T I O N *****
/*
Connect the RFM12 to the AVR as follows:

  RFM12      | AVR
-----+-----
  SDO       | MISO
  nIRQ      | INT0
  FSK/DATA/nFFS | VCC
  DCLK/CFIL/FFIT | -
  CLK        | -
  nRES       | -
  GND        | GND
  ANT        | -
  VDD        | VCC
  GND        | GND
  nINT/vDI   | -
  SDI        | MOSI
  SCK        | SCK
  nSEL       | Slave select pin defined below
*/
***** 
* RFM12 PIN DEFINITIONS
*/
```

```

*****  

* Debug LED for examples  

*/  
  

#define LED_PORT PORTD  

#define LED_DDR DDRD  

#define LED_BITR PD7  

#define LED_BITG PD6  
  

*****  

*/  
  

//Pin that the RFM12's slave select is connected to  

#define DDR_SS DDRB  

#define PORT_SS PORTB  

#define BIT_SS 4  
  

//SPI port  

#define DDR_SPI DDRB  

#define PORT_SPI PORTB  

#define PIN_SPI PINB  

#define BIT_MOSI 5  

#define BIT_MISO 6  

#define BIT_SCK 7  

#define BIT_SPI_SS 4  

//this is the hardware SS pin of the AVR - it  

//needs to be set to output for the spi-interface to work  

//correctly, independently of the CS pin used for the RFM12  
  

*****  

* RFM12 CONFIGURATION OPTIONS  

*/  
  

//baseband of the module (either RFM12_BAND_433, RFM12_BAND_868 or RFM12_BAND_912)  

#define RFM12_BASEBAND RFM12_BAND_433  
  

//center frequency to use (+- FSK frequency shift)  

#define RFM12_FREQUENCY 430800000UL  
  

//Transmit FSK frequency shift in kHz  

#define FSK_SHIFT 125000  
  

//Receive RSSI Threshold  

#define RFM12_RXCTRL_RSSI_79  
  

//Receive Filter Bandwidth  

#define RFM12_RXCTRL_BW_400  
  

//Output power relative to maximum (0dB is maximum)  

#define RFM12_POWER RFM12_TXCONF_POWER_0  
  

//Receive LNA gain  

#define RFM12_RXCTRL_LNA_6  
  

//crystal load capacitance - the frequency can be verified by measuring the  

//clock output of RFM12 and comparing to 1MHz.  

//11.5pF seems to be o.k. for RFM12, and 10.5pF for RFM12BP, but this may vary.  

#define RFM12_XTAL_LOAD RFM12_XTAL_11_5PF  
  

//use this for datarates >= 2700 Baud  

// #define DATARATE_VALUE RFM12_DATARATE_CALC_HIGH(9600.0)

```

```

//use this for 340 Baud < datarate < 2700 Baud
#define DATARATE_VALUE      RFM12_DATARATE_CALC_LOW(1200.0)

//TX BUFFER SIZE
#define RFM12_TX_BUFFER_SIZE 30

//RX BUFFER SIZE (there are going to be 2 Buffers of this size for double_buffering)
#define RFM12_RX_BUFFER_SIZE 30

/******************
 * RFM12 INTERRUPT VECTOR
 * set the name for the interrupt vector here
 */

//the interrupt vector
#define RFM12_INT_VECT (INT0_vect)

//the interrupt mask register
#define RFM12_INT_MSK EIMSK

//the interrupt bit in the mask register
#define RFM12_INT_BIT (INT0)

//the interrupt flag register
#define RFM12_INT_FLAG EIFR

//the interrupt bit in the flag register
#define RFM12_FLAG_BIT (INTF0)

//setup the interrupt to trigger on level
#define RFM12_INT_SETUP() MCUCR &= ~(1<<ISC01)

/******************
 * FEATURE CONFIGURATION
 */

#define RFM12_LIVECTRL 0
#define RFM12_LIVECTRL_CLIENT 0
#define RFM12_LIVECTRL_HOST 0
#define RFM12_LIVECTRL_LOAD_SAVE_SETTINGS 0
#define RFM12_NORETURN 0
#define RFM12_NO_COLLISION_DETECTION 0
#define RFM12_TRANSMIT_ONLY 0
#define RFM12_SPI_SOFTWARE 0
#define RFM12_USE_POLLING 0
#define RFM12_RECEIVE_ASK 0
#define RFM12_TRANSMIT_ASK 0
#define RFM12_USE_WAKEUP_TIMER 0
#define RFM12_USE_POWER_CONTROL 0
#define RFM12_LOW_POWER 0
#define RFM12_USE_CLOCK_OUTPUT 0
#define RFM12_LOW_BATT_DETECTOR 0

#define RFM12_LBD_VOLTAGE          RFM12_LBD_VOLTAGE_3V0

```

```

#define RFM12_CLOCK_OUT_FREQUENCY      RFM12_CLOCK_OUT_FREQUENCY_1_00_MHz

/* use a callback function that is called directly from the
 * interrupt routine whenever there is a data packet available. When
 * this value is set to 1, you must use the function
 * "rfm12_set_callback(your_function)" to point to your
 * callback function in order to receive packets.
 */
#define RFM12_USE_RX_CALLBACK 0

*****  

* RFM12BP support (high power version of RFM12)
*/  

//To use RFM12BP, which needs control signals for RX enable and TX enable,
//use these defines (set to your pinout of course).
//The TX-Part can also be used to control a TX-LED with the nomral RFM12

/*
#define RX_INIT_HOOK  DDRD |= _BV(PD5)
#define RX_LEAVE_HOOK PORTD &= ~_BV(PD5)
#define RX_ENTER_HOOK PORTD |= _BV(PD5)

#define TX_INIT_HOOK  DDRD |= _BV(PD4)
#define TX_LEAVE_HOOK PORTD &= ~_BV(PD4)
#define TX_ENTER_HOOK PORTD |= _BV(PD4)
*/  

*****  

* UART DEBUGGING
* en- or disable debugging via uart.
*/  

#define RFM12_UART_DEBUG 0

```

general.h

```

#ifndef _General

#define _General

#define TRUE    1
#define FALSE   0
/*
** Here are some deinitions, used in all programs
*/
#define SYSCLOCK           8000000 // Quarz Frequenz in Hz

#define SET_BIT(PORT, BITNUM) ((PORT) |= (1<<(BITNUM)))
#define CLEAR_BIT(PORT, BITNUM) ((PORT) &= ~(1<<(BITNUM)))
#define TOGGLE_BIT(PORT,BITNUM) ((PORT) ^= (1<<(BITNUM)))

```

```
struct BitsOfByte
{
    uint8_t b0:1;
    uint8_t b1:1;
    uint8_t b2:1;
    uint8_t b3:1;
    uint8_t b4:1;
    uint8_t b5:1;
    uint8_t b6:1;
    uint8_t b7:1;
} __attribute__((__packed__));

#define SBIT(port,pin) ((*(volatile struct BitsOfByte*)&port).b##pin)

#endif
```

main.h

```
#include <math.h>
#include <avr/io.h>
#include <util/delay.h>
#include <stdlib.h>
#include <stdio.h>
#include <avr/interrupt.h>

//uint8_t fehler = 0;

#define uchar unsigned char
#define uint unsigned int
#define bit uchar
#define idata
#define code

#define MATCH_ROM      0x55
#define SKIP_ROM0xCC 0xCC
#define SEARCH_ROM    0xF0

#define CONVERT_T      0x44          // DS1820 commands
#define READ           0xBE
#define WRITE          0x4E
#define EE_WRITE0x48   0x48
#define EE_RECALL      0xB8

#define SEARCH_FIRST   0xFF          // start new search
#define PRESENCE_ERR   0xFF
#define DATA_ERR0xFE   0xFE
#define LAST_DEVICE    0x00          // last device found
//                           0x01 ... 0x40: continue searching

void warten(void);
void selbsttest(void);
void zahl_ausgeben(int);
bit w1_reset(void);
uchar w1_bit_io(bit);
uint w1_byte_wr(uchar);
uint w1_byte_rd(void);
uchar w1_rom_search( uchar, uchar idata * );
void w1_command( uchar , uchar idata * );
uint8_t read_meas( char * );
void start_meas( void );
void ad_init(void);
int ad_read(uint8_t);
void senden(uint8_t*, uint8_t);
uint8_t* empfangen(void);
void tf_messen(unsigned int* , unsigned int*);
```

Das Hauptprogramm ist bereits sehr umfangreich, um es zu verstehen, ist es nützlich, erst einmal ein Struktogramm davon zu zeigen.

Hauptprogramm Wetterzentrale

```
#defines
Port Definition
globale Variablen

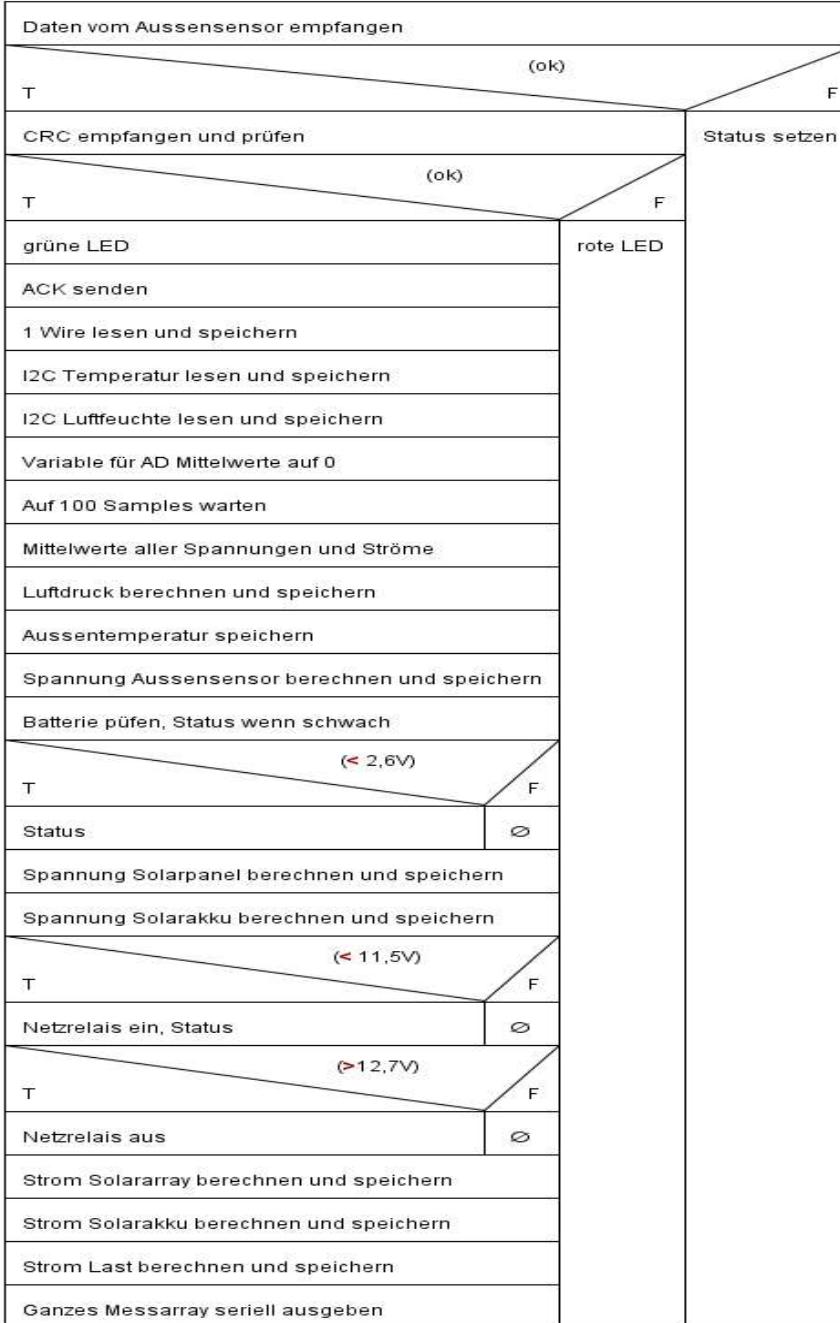
ADC Interrupt

Timer1 Interrupt

Deklaration
senden() -> rfm12
empfangen() -> rfm12
tf_messen() -> I2C

Main:
alle Variablen deklarieren und initialisieren
Hardware initialisieren
LED blinken lassen
Messarray initialisieren und Statusarray vorbelegen
```

```
while (1)
```



main.c

```
// Zentrale Wetterstation und Solarueberwachung
// 26.01.22 JK

// Portbelegung AT Mega 1284

// Analog:
// U Solar          PA0 Pin 40
// U Akku und Last PA1 Pin 39
// I Solar          PA2 Pin 38
// I Akku           PA3 Pin 37
// I Last           PA4 Pin 36
// Luftdruck        PA5 Pin 35

// Digital:
// Relais           PC2 Pin 24
// Feuchte / Temp SDA PC1 Pin 23
// Feuchte / Temp SCL PC0 Pin 22
// Temp 1Wire       PD5 Pin 19
// LED gruen        PD6 Pin 20
// LED rot          PD7 Pin 21
// Funkmodul INT0   PD2 Pin 16
// Funkmodul SCK    PB7 Pin 8
// Funkmodul MISO   PB6 Pin 7
// Funkmodul MOSI   PB5 Pin 6
// Funkmodul SS     PB4 Pin 5
// Raspberry RX    PD0 Pin 14
// Raspberry TX    PD1 Pin 15

// ich verwende zur Uebertragung an den Web-Server ein
// Mess-Array mit physikalischen Messwerten
// Es beginnt mit einer Kennung und hat immer 4 Byte
// die Sortierung geht nach Temperatur, Luftfeuchte, Luftdruck, Spannung, Strom und Status
// es werden Ganzzahlen gespeichert
// die Temperatur wird mit Vorzeichen, 2 Stellen und 1 Nachkommastelle gespeichert
// die Feuchte mit 2 Stellen und 1 Nachkommastelle
// der Luftdruck mit 3 bis 4 Stellen
// die Spannung 2 Stellen, eine Nachkommastelle
// der Strom 1 Stelle, 2 Nachkommastellen, Ausnahme Batterie, hier auch das Vorzeichen
// der Status nur 0 oder Fehlercode
// Fehlercode: 0001 Timeout; 0010 Uebertragungsf.; 0100 Batt Geber leer ; 1000 Akku Unterspannung

#include "main.h"

#include "rfm12.h"
#include "TWI_Master.h"
#include "serial.h"

#include <util/crc16.h>
#include <string.h>
#include <avr/sleep.h>

#define Dev_HUMCP_HYT_271 0x28      // Adresse des Feuchtesensors

#define LO_BATT_ON PORTC |= (1 << PC2);
#define LO_BATT_OFF PORTC &= ~(1 << PC2);

#define uint unsigned int

#define true 1
#define false 0
```

```

uint8_t timeout = 0;

static volatile uint16_t cnt = 0;
uint8_t laenge;
uint8_t sende_daten_ack[] = "ack";
uint8_t* empfangs_daten;
uint8_t g;

#define TABSIZE           100      // Groesse AD Sample Tabelle
#define CH0               0        // State Machine fuer Kanalzuordnung
#define CH1               1
#define CH2               2
#define CH3               3
#define CH4               4
#define CH5               5

uint8_t static volatile chan_status;      // 6 Zustaende
uint8_t static volatile ad_index;         // Tabellenindex
uint8_t static volatile samples_ready;    // 100 Werte geschrieben
uint16_t static volatile ad_samples[TABSIZE][6]; // Tabelle fuer AD Werte

```

```

ISR(ADC_vect) // Einlesen von 6 Kanälen
{
    // das muss tatsächlich so verschoben stehen
    // sonst werden die Kanäle nicht richtig zugeordnet
    // denn Multiplexer wird erst beim folgenden Sample gültig

    switch(chan_status)
    {
        case CH0:
            // Einlesen Kanal 1 (Spannung Solarpanel)
            ad_samples[ad_index][CH5] = ADCW;

            ADMUX |= (1 << MUX0);
            chan_status = CH1;
            break;
        case CH1:
            // Einlesen Kanal 2 (Spannung Solarakku)

            ad_samples[ad_index][CH0] = ADCW;
            ADMUX &= ~(1 << MUX0);
            ADMUX |= (1 << MUX1);
            chan_status = CH2;
            break;
        case CH2:
            // Einlesen Kanal 3 (Strom Solarzelle)
            ad_samples[ad_index][CH1] = ADCW;
            ADMUX |= (1 << MUX0) | (1 << MUX1);
            chan_status = CH3;
            break;
        case CH3:
            // Einlesen Kanal 4 (Strom Solarakku)
            ad_samples[ad_index][CH2] = ADCW;
            ADMUX &= ~((1 << MUX0) | (1 << MUX1));
            ADMUX |= (1 << MUX2);
            chan_status = CH4;
            break;
        case CH4:
            // Einlesen Kanal 5 (Strom Last)
            ad_samples[ad_index][CH3] = ADCW;

            ADMUX |= (1 << MUX0) | (1 << MUX2);
            chan_status = CH5;
            break;
        case CH5:
            // Einlesen Kanal 6 (Spannung Luftdruck)
            ad_samples[ad_index][CH4] = ADCW;
            ADMUX &= ~((1 << MUX0) | (1 << MUX1) | (1 << MUX2) | (1 << MUX3) | (1 << MUX4));

            chan_status = CH0;
            ad_index++; // Tabellenindex erhöhen
            break;
        default:
            break;
    }
    if(ad_index > TABSIZE - 1) // alle 100 Werte gelesen
    {
        ad_index = 0;
        samples_ready = 1;
    }
}

```

```

ISR(TIMER1_OVF_vect)
{
    cnt++;
}

void ad_init(void)
{
    ADMUX |= (1<<REFS1)|(1<<REFS0);           // ADC Ref auf AREF geschaltet
    ADCSRA |= (1<<ADEN)|(1 << ADIE)| (1 << ADATE) | (1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
                                                // ADC eingeschaltet,
                                                // Freilaufend, Interrupt, Teiler / 128
                                                // Eingangstakt 8E6 / 128 / 13
                                                // (13 Zyklen fuer Wandlung) T = 208 µs
    ADCSRA |= (1<<ADSC);                      // eine ADC-Wandlung starten
}

void senden(uint8_t string[], uint8_t adr)
{
    uint8_t i;
    rfm12_tx((strlen((const char*)string)+1),adr, string);
    for(i = 0; i < 20;i++)
        rfm12_tick();
}

uint8_t* empfangen(void)
{
    uint8_t* buffer = NULL;
    uint8_t finished = false;

    cnt = 0;
    while(finished == false)
    {
        if(cnt > 316) // 2,09s * 316 = 660,44s / 60 = 11 Min)
        {
            finished = true;
            timeout = 1;
        }
        if(rfm12_rx_status() == STATUS_COMPLETE)
        {
            buffer = rfm12_rx_buffer();
            finished = true;
            timeout = 0;
        }
    }
    return buffer;
}

```

```

void tf_messen(unsigned int* rf_raw, unsigned int* t_raw)
{
    uint8_t i;
    uint8_t Data[4];
    if (!TWIM_Start (Dev_HUMCP_HYT_271, TWIM_WRITE))
    {
        TWIM_Stop ();
        USART_puts("1. Sensorfehler !\r\n");
        while(1);
    }
    else
    {

        TWIM_ReadAck ();
        TWIM_Stop ();
        _delay_ms(100);
    }

    if (!TWIM_Start (Dev_HUMCP_HYT_271, TWIM_READ))
    {
        TWIM_Stop ();
        USART_puts("2. Sensorfehler !\r\n");
        while(1);
    }
    else
    {
        for (i=0;i<3;i++)
        {
            Data[i] = TWIM_ReadAck ();
        }
        Data[3] = TWIM_ReadNack ();
        TWIM_Stop ();
    }

    *rf_raw = ((Data[0] << 8) & 0x3FFF) | (Data[1] & 0xFFFF); // als 14 bit Wert
    *t_raw = ((Data[2] << 8) | Data[3]) >> 2; // als 14 bit Wert
}

```

```

int main ( void )
{
    int i,k;
    char messwert[30];

    uint8_t messarray[12][6];

    uint16_t zt,t,h,z,e;
    uint16_t crc = 0;
    uint8_t crc_array[5];

    uint16_t spannung_aussensensor_zahlenwert;
    double spannung_aussensensor_double;
    uint16_t spannung_aussensensor_ganzzahl;

    uint16_t spannung_solarpanel_zahlenwert;
    double spannung_solarpanel_double;           // max. 20 Volt
    uint16_t spannung_solarpanel_ganzzahl;

    uint16_t spannung_batterie_zahlenwert;
    double spannung_batterie_double;           // max. 20 Volt
    uint16_t spannung_batterie_ganzzahl;

    uint16_t strom_solarpanel_zahlenwert;
    double strom_solarpanel_double;           // max 2A
    uint16_t strom_solarpanel_ganzzahl;

    uint16_t strom_batterie_zahlenwert;
    double strom_batterie_double;           // max +/-6A
    int16_t strom_batterie_ganzzahl;

    uint16_t strom_last_zahlenwert;
    double strom_last_double;           // max 6A
    uint16_t strom_last_ganzzahl;

    uint16_t spannung_airpressure_zahlenwert;
    double spannung_airpressure_double;       // 0.01 bis 1V = 4mV/hPa 850 - 1100 Hpa
    uint16_t spannung_airpressure_ganzzahl;

    double luftfeuchte_double;
    uint16_t luftfeuchte_ganzzahl;

    double temperatur_double;
    uint16_t temperatur_ganzzahl;

    unsigned long spannung_solar_summe;        // Summe fuer Mittelwert
    unsigned long spannung_batterie_summe;
    unsigned long strom_solar_summe;
    unsigned long strom_batterie_summe;
    unsigned long strom_last_summe;
    unsigned long spannung_airpressure_summe;

    float vref = 2.553;                      // am Chip nachgemessen

    unsigned int rf_raw;
    unsigned int t_raw;

    LED_DDR |= (1 <<LED_BITR) | (1 << LED_BITG);      // LEDs als Ausgang definieren
    DDRC |= (1 << PORTC2);                            // Relais als Ausgang definieren

    // Power on Indikator, beide LEDS blinken 1 mal
    // LEDs sind Low-Aktiv

```

```

LED_PORT &= ~(1 << LED_BITG); // gruene LED an
LED_PORT &= ~(1 << LED_BITR); // rote LED an
_delay_ms(1000);
LED_PORT |= (1 << LED_BITG); // gruene LED aus
LED_PORT |= (1 << LED_BITR); // rote LED aus

ad_init();

_delay_ms(100); // RFM braucht etwas Zeit zum Hochfahren

rfm12_init(); // Initialisieren RFM12

TWIM_Init(100000); // I2C initialisieren

// Timer 1 aktivieren

TCCR1B |= (1 << CS12); // Prescaler 256, Overfl alle 2,09 s
TIMSK1 |= (1 << TOIE1); // Interrupt bei T1 Overfl.

ad_index = 0;
chan_status = CH0; // State initialisieren

sei(); //Interrupts aktiviert

USART_init(); // serielle Schnittstelle initialisieren

// ganzes Messarray initialisieren:

for(i = 0; i < 12;i++)
{
    for(k = 0; k < 6;k++)
        messarray[i][k] = 0;

messarray[11][0] = 'L'; // Status-Array vorbelegen
messarray[11][1] = ' ';
messarray[11][2] = '0';
messarray[11][3] = '0';
messarray[11][4] = '0';
messarray[11][5] = '0';

while (1) // groÙe Hauptschleife
{
    // warten, bis irgendwas empfangen wird, steht dann in ret_wert
    // kein Empfang: warte 11 Minuten, dann abbrechen (erfolgt im Modul "empfangen")

    empfangs_daten = empfangen();

    if(timeout == 1)
    {
        messarray[11][5] = '1';
        timeout = 0;

    }
    else
        messarray[11][5] = '0';
}

```

```

// CRC-Pruefung

crc = 0x00; // simple CRC
for (i=0;i<19;i++)
    crc = _crc_ccitt_update(crc,empfangs_daten[i]);

e = crc;
z = crc/10;
h = crc /100;
t = crc / 1000;
zt = crc / 10000;

crc_array[0] = (zt % 10) + 0x30;
crc_array[1] = (t % 10) + 0x30;
crc_array[2] = (h % 10) + 0x30;
crc_array[3] = (z % 10) + 0x30;
crc_array[4] = (e % 10) + 0x30;

// auf Datenfehler pruefen:

g = rfm12_rx_type();           // aus Paket Absender-Adresse herausholen
laenge = rfm12_rx_len()-1;     // Nutzdaten-Laenge ermitteln
rfm12_rx_clear();             // jetzt brauchen wir den Empfangspuffer nicht mehr

// alles in 1 Zeile schreiben - oder vor dem && umbrechen
if((crc_array[0] == empfangs_daten[22]) && (crc_array[1] == empfangs_daten[23])
&& (crc_array[2] == empfangs_daten[24]) && (crc_array[3] == empfangs_daten[25])
&& (crc_array[4] == empfangs_daten[26]))
{
    messarray[11][4] = '0';

    LED_PORT &= ~(1 << LED_BITG); // gruene LED an
    _delay_ms(100);                // warten, Geber beritet Empfang vor
    senden(sende_daten_ack,g);    // ein ACK als Handshake senden
    LED_PORT |= (1 << LED_BITG); // gruene LED wieder ausschalten

    start_meas();                 // 1 Wire Sensor lesen
    if(read_meas(messwert) == 0)
    {
        messarray[0][0] = 'A';   // Innentemperatur 1 Wire
        messarray[0][1] = ' ';
        messarray[0][2] = messwert[2];
        messarray[0][3] = messwert[3];
        messarray[0][4] = ',';
        messarray[0][5] = messwert[4];
    }
    else
    {
        messarray[0][0] = 'A';   // Innentemperatur 1 Wire
        messarray[0][1] = 'S';
        messarray[0][2] = 'E' ;
        messarray[0][3] = 'N';
        messarray[0][4] = 'S';
        messarray[0][5] = '?';
    }
}

```

```

    tf_messen(&rf_raw, &t_raw);      // I2C-Sensor lesen

    temperatur_double = (double) 1650 *      t_raw / 16383.0 -400 ;
    temperatur_ganzzahl = (uint16_t) round(temperatur_double);

    messarray[1][0] = 'B';      // Temperatur I2C
    messarray[1][1] = ' ';
    messarray[1][2] = (uint8_t) 0x30 + temperatur_ganzzahl / 100;
    messarray[1][3] = (uint8_t) 0x30 + temperatur_ganzzahl % 100 /10;
    messarray[1][4] = ',';
    messarray[1][5] = (uint8_t) 0x30 + temperatur_ganzzahl % 10;

    luftfeuchte_double = (double) 1000 *      rf_raw / 16383.0 ;
    luftfeuchte_ganzzahl = (uint16_t) round(luftfeuchte_double);

    messarray[2][0] = 'C';      // Luftfeuchte I2C
    messarray[2][1] = ' ';
    messarray[2][2] = (uint8_t) 0x30 + luftfeuchte_ganzzahl / 100;
    messarray[2][3] = (uint8_t) 0x30 + luftfeuchte_ganzzahl % 100 /10;
    messarray[2][4] = ',';
    messarray[2][5] = (uint8_t) 0x30 + luftfeuchte_ganzzahl % 10;

    spannung_solar_summe = 0;      // Variablen für Mittelung auf 0 setzen
    spannung_batterie_summe = 0;
    strom_solar_summe = 0;
    strom_batterie_summe = 0;
    strom_last_summe = 0;
    spannung_airpressure_summe = 0;
    samples_ready = 0;           // Ready-Flag loeschen

    while(samples_ready == 0);   // warte auf 100 Samples

    for(i = 0; i < TABSIZE; i++) // Summe ueber Iteration (fuer Mittelwert)
    {
        spannung_solar_summe = spannung_solar_summe + ad_samples[i][0];
        spannung_batterie_summe = spannung_batterie_summe + ad_samples[i][1];
        strom_solar_summe = strom_solar_summe + ad_samples[i][2];
        strom_batterie_summe = strom_batterie_summe + ad_samples[i][3];
        strom_last_summe = strom_last_summe + ad_samples[i][4];
        spannung_airpressure_summe = spannung_airpressure_summe + ad_samples[i][5];
    }

    // Mittelwerte

    spannung_solarpanel_zahlenwert  = spannung_solar_summe / TABSIZE;
    spannung_batterie_zahlenwert = spannung_batterie_summe / TABSIZE;
    strom_solarpanel_zahlenwert = strom_solar_summe / TABSIZE;
    strom_batterie_zahlenwert = strom_batterie_summe / TABSIZE;
    strom_last_zahlenwert = strom_last_summe / TABSIZE;
    spannung_airpressure_zahlenwert = spannung_airpressure_summe / TABSIZE;

    spannung_airpressure_double = (double) 850.0
    + spannung_airpressure_zahlenwert / 4.092 ;
    spannung_airpressure_ganzzahl = (uint16_t) round(spannung_airpressure_double);

    messarray[3][0] = 'D';      // Luftdruck
    messarray[3][1] = ' ';
    messarray[3][2] = (uint8_t) 0x30 + spannung_airpressure_ganzzahl / 1000;
    messarray[3][3] = (uint8_t) 0x30 + spannung_airpressure_ganzzahl % 1000 / 100;
    messarray[3][4] = (uint8_t) 0x30 + spannung_airpressure_ganzzahl % 100 / 10;
    messarray[3][5] = (uint8_t) 0x30 + spannung_airpressure_ganzzahl % 10;

    messarray[4][0] = 'E';           // Aussentemperatur

```

```

messarray[4][1] = empfangs_daten[3];
messarray[4][2] = empfangs_daten[4];
messarray[4][3] = empfangs_daten[5];
messarray[4][4] = ',';
messarray[4][5] = empfangs_daten[7];

spannung_aussensensor_zahlenwert = 1000 * (empfangs_daten[11] - 0x30)
+ 100 * (empfangs_daten[12] - 0x30) + 10 * (empfangs_daten[13] - 0x30)
+ (empfangs_daten[14] - 0x30);
spannung_aussensensor_double = (double) spannung_aussensensor_zahlenwert / 1024.0
* 3.48; // Kalibrierfaktor // mit Gleitkommazahl rechnen
spannung_aussensensor_ganzzahl = (uint8_t) round(spannung_aussensensor_double
* 10); // Ganzzahl daraus machen und runden

if(spannung_aussensensor_double < 2.6) // Funksender Batterie schwach
    messarray[11][3] = '1';
else
    messarray[11][3] = '0';

messarray[5][0] = 'F'; // Spannung Funksender
messarray[5][1] = ' ';
messarray[5][2] = ' ';
messarray[5][3] = (uint8_t) 0x30 + spannung_aussensensor_ganzzahl / 10;
messarray[5][4] = ',';
messarray[5][5] = (uint8_t) 0x30 + spannung_aussensensor_ganzzahl % 10;

spannung_solarpanel_double = (double) spannung_solarpanel_zahlenwert / 1024.0
* vref * 8.38; // Kalibrierfaktor
spannung_solarpanel_ganzzahl = (uint8_t) round(spannung_solarpanel_double * 10);

messarray[6][0] = 'G'; // Spannung Solarpanel
messarray[6][1] = ' ';
messarray[6][2] = (uint8_t) 0x30 + spannung_solarpanel_ganzzahl / 100;
messarray[6][3] = (uint8_t) 0x30 + spannung_solarpanel_ganzzahl % 100 / 10;
messarray[6][4] = ',';
messarray[6][5] = (uint8_t) 0x30 + spannung_solarpanel_ganzzahl % 10;

spannung_batterie_double = (double) spannung_batterie_zahlenwert / 1024.0 * vref
* 8.38;
spannung_batterie_ganzzahl = (uint8_t) round(spannung_batterie_double * 10);

if(spannung_batterie_double < 11.5) // Solarakku schwach
{
    messarray[11][2] = '1';
    LO_BATT_ON;
}
else if(spannung_batterie_double > 12.7) // Solarakku ausreichend
{
    messarray[11][2] = '0';
    LO_BATT_OFF;
}

messarray[7][0] = 'H'; // Spannung Solarakku
messarray[7][1] = ' ';
messarray[7][2] = (uint8_t) 0x30 + spannung_batterie_ganzzahl / 100;
messarray[7][3] = (uint8_t) 0x30 + spannung_batterie_ganzzahl % 100 / 10;
messarray[7][4] = ',';
messarray[7][5] = (uint8_t) 0x30 + spannung_batterie_ganzzahl % 10;

strom_solarpanel_double = (double) strom_solarpanel_zahlenwert / 1024.0 * vref
* 1.86 ; // Kalibrierfaktor
strom_solarpanel_ganzzahl = (uint8_t) round(strom_solarpanel_double * 100);

```

```

messarray[8][0] = 'I';      // Strom Solarzelle
messarray[8][1] = ' ';
messarray[8][2] = (uint8_t) 0x30 + strom_solarpanel_ganzzahl / 100;
messarray[8][3] = ',';
messarray[8][4] = (uint8_t) 0x30 + strom_solarpanel_ganzzahl % 100 / 10;
messarray[8][5] = (uint8_t) 0x30 + strom_solarpanel_ganzzahl % 10;

strom_batterie_double = (double) (strom_batterie_zahlenwert / 1024.0 * vref - 1.28)
* 3.8;    // Kalibrierfaktor, I Batt - und +, daher Offset von 1,28

strom_batterie_ganzzahl = (int16_t) round(strom_batterie_double * 100);

messarray[9][0] = 'J';      // Strom Solarakku
if(strom_batterie_double < 0)
{
    messarray[9][1] = '-';
    messarray[9][2] = (uint8_t) 0x30 - strom_batterie_ganzzahl / 100;
    messarray[9][3] = ',';
    messarray[9][4] = (uint8_t) 0x30 - strom_batterie_ganzzahl % 100 / 10;
    messarray[9][5] = (uint8_t) 0x30 - strom_batterie_ganzzahl % 10;
}
else
{
    messarray[9][1] = ' ';
    messarray[9][2] = (uint8_t) 0x30 + strom_batterie_ganzzahl / 100;
    messarray[9][3] = ',';
    messarray[9][4] = (uint8_t) 0x30 + strom_batterie_ganzzahl % 100 / 10;
    messarray[9][5] = (uint8_t) 0x30 + strom_batterie_ganzzahl % 10;
}

strom_last_double = (double) strom_last_zahlenwert / 1024.0 * vref * 1.86;
strom_last_ganzzahl = (uint8_t) round(strom_last_double * 100);

messarray[10][0] = 'K';    // Strom Verbraucher
messarray[10][1] = ' ';
messarray[10][2] = (uint8_t) 0x30 + strom_last_ganzzahl / 100;
messarray[10][3] = ',';
messarray[10][4] = (uint8_t) 0x30 + strom_last_ganzzahl % 100 / 10;
messarray[10][5] = (uint8_t) 0x30 + strom_last_ganzzahl % 10;

}

else
{
    messarray[11][4] = '1';
    LED_PORT &= ~_BV(LED_BITR);    // rote LED an
    _delay_ms(100);
    LED_PORT |= _BV(LED_BITR);    // rote LED aus
}

```

```

// gesamtes Mess- Array ausgeben:

// fuer LINUX 0A fuer Zeilenende nehmen!
// Trenner fuer PHP ist "_"

USART_putc('_');
USART_putc(messarray[0][0]);USART_putc('_');USART_putc(messarray[0][1]);
USART_putc(messarray[0][2]);USART_putc(messarray[0][3]);
USART_putc(messarray[0][4]);USART_putc(messarray[0][5]);
USART_putc(0x0A);
USART_putc('_');

USART_putc(messarray[1][0]);USART_putc('_');USART_putc(messarray[1][1]);
USART_putc(messarray[1][2]);USART_putc(messarray[1][3]);
USART_putc(messarray[1][4]);USART_putc(messarray[1][5]);
USART_putc(0x0A);
USART_putc('_');

USART_putc(messarray[2][0]);USART_putc('_');USART_putc(messarray[2][1]);
USART_putc(messarray[2][2]);USART_putc(messarray[2][3]);
USART_putc(messarray[2][4]);USART_putc(messarray[2][5]);
USART_putc(0x0A);
USART_putc('_');

USART_putc(messarray[3][0]);USART_putc('_');USART_putc(messarray[3][1]);
USART_putc(messarray[3][2]);USART_putc(messarray[3][3]);
USART_putc(messarray[3][4]);USART_putc(messarray[3][5]);
USART_putc(0x0A);
USART_putc('_');

USART_putc(messarray[4][0]);USART_putc('_');USART_putc(messarray[4][1]);
USART_putc(messarray[4][2]);USART_putc(messarray[4][3]);
USART_putc(messarray[4][4]);USART_putc(messarray[4][5]);
USART_putc(0x0A);
USART_putc('_');

USART_putc(messarray[5][0]);USART_putc('_');USART_putc(messarray[5][1]);
USART_putc(messarray[5][2]);USART_putc(messarray[5][3]);
USART_putc(messarray[5][4]);USART_putc(messarray[5][5]);
USART_putc(0x0A);
USART_putc('_');

USART_putc(messarray[6][0]);USART_putc('_');USART_putc(messarray[6][1]);
USART_putc(messarray[6][2]);USART_putc(messarray[6][3]);
USART_putc(messarray[6][4]);USART_putc(messarray[6][5]);
USART_putc(0x0A);
USART_putc('_');

USART_putc(messarray[7][0]);USART_putc('_');USART_putc(messarray[7][1]);
USART_putc(messarray[7][2]);USART_putc(messarray[7][3]);
USART_putc(messarray[7][4]);USART_putc(messarray[7][5]);
USART_putc(0x0A);
USART_putc('_');

USART_putc(messarray[8][0]);USART_putc('_');USART_putc(messarray[8][1]);
USART_putc(messarray[8][2]);USART_putc(messarray[8][3]);
USART_putc(messarray[8][4]);USART_putc(messarray[8][5]);
USART_putc(0x0A);
USART_putc('_');

```

```

USART_putc(messarray[9][0]);USART_putc('_');USART_putc(messarray[9][1]);
USART_putc(messarray[9][2]);USART_putc(messarray[9][3]);
USART_putc(messarray[9][4]);USART_putc(messarray[9][5]);
USART_putc(0x0A);
USART_putc('_');

USART_putc(messarray[10][0]);USART_putc('_');USART_putc(messarray[10][1]);
USART_putc(messarray[10][2]);USART_putc(messarray[10][3]);
USART_putc(messarray[10][4]);USART_putc(messarray[10][5]);
USART_putc(0x0A);
USART_putc('_');

USART_putc(messarray[11][0]);USART_putc('_');USART_putc(messarray[11][1]);
USART_putc(messarray[11][2]);USART_putc(messarray[11][3]);
USART_putc(messarray[11][4]);USART_putc(messarray[11][5]);
USART_putc(0x0A);

}

```

Soweit die Theorie. Leider hat die Software ein nicht perfektes Verhalten, denn nach einem Timeout oder Übertragungsfehler wird ein leeres Messarray generiert, es enthält nur die Statusinformation. Das Problem ist für den Leser, der bisher alles verstanden hat, sicher leicht zu lösen!

1wire.c

```
#include "main.h"

// Anfang Unterprogramme 1-Wire

//****************************************************************************
/*
 *          Access Dallas 1-Wire Devices
 */
/*
 *      Author: Peter Dannegger
 *            danni@specs.de
 */
//****************************************************************************

#ifndef W1_PIN
#define W1_PIN  PD5
#define W1_IN   PIND
#define W1_OUT  PORTD
#define W1_DDR  DDRD
#endif

bit w1_reset(void)
{
    bit err;

    W1_OUT &= ~(1<<W1_PIN);
    W1_DDR |= 1<<W1_PIN;
    _delay_us( 480 );           // 480 us
    cli();
    W1_DDR &= ~(1<<W1_PIN);
    _delay_us( 66 );
    err = W1_IN & (1<<W1_PIN); // no presence detect
    sei();
    _delay_us( 480 - 66 );
    if( (W1_IN & (1<<W1_PIN)) == 0 )// short circuit
        err = 1;
    return err;
}
```

```

uchar w1_bit_io( bit b )
{
    cli();
    W1_DDR |= 1<<W1_PIN;
    _delay_us(1);
    if( b )
        W1_DDR &= ~(1<<W1_PIN);
    _delay_us( 15 - 1 );
    if( (W1_IN & (1<<W1_PIN)) == 0 )
        b = 0;
    _delay_us( 60 - 15 );
    W1_DDR &= ~(1<<W1_PIN);
    sei();
    return b;
}

uint w1_byte_wr( uchar b )
{
    uchar i = 8, j;
    do{
        j = w1_bit_io( b & 1 );
        b >>= 1;
        if( j )
            b |= 0x80;
    }while( --i );
    return b;
}

uint w1_byte_rd( void )
{
    return w1_byte_wr( 0xFF );
}

```

```

uchar w1_rom_search( uchar diff, uchar idata *id )
{
    uchar i, j, next_diff;
    bit b;

    if( w1_reset() )
        return PRESENCE_ERR;                                // error, no device found
    w1_byte_wr( SEARCH_ROM );                            // ROM search command
    next_diff = LAST_DEVICE;                            // unchanged on last device
    i = 8 * 8;                                         // 8 bytes
    do{
        j = 8;                                           // 8 bits
        do{
            b = w1_bit_io( 1 );                         // read bit
            if( w1_bit_io( 1 ) ){ // read complement bit
                if( b )                                // 11
                    return DATA_ERR; // data error
                }else{
                    if( !b ){                      // 00 = 2 devices
                        if( diff > i ||          // now 1
                            ((*id & 1) && diff != i ) ){
                            b = 1;                  // now 1
                            next_diff = i; // next pass 0
                        }
                    }
                }
            w1_bit_io( b );                         // write bit
            *id >>= 1;
            if( b )                                // store bit
                *id |= 0x80;
            i--;
        }while( --j );
        id++;                                         // next byte
    }while( i );
    return next_diff;                                // to continue search
}

```

```

void w1_command( uchar command, uchar idata *id )
{
    uchar i;
    w1_reset();
    if( id ){
        w1_byte_wr( MATCH_ROM );           // to a single device
        i = 8;
        do{
            w1_byte_wr( *id );
            id++;
        }while( --i );
    }else{
        w1_byte_wr( SKIP_ROM );          // to all devices
    }
    w1_byte_wr( command );
}

void start_meas( void ){
    if( W1_IN & 1<< W1_PIN ){
        w1_command( CONVERT_T, NULL );
        W1_OUT |= 1<< W1_PIN;
        W1_DDR |= 1<< W1_PIN;           // parasite power on
    }
}

uint8_t  read_meas( char *s )
{
    uchar id[8], diff;
    int temp;

    for( diff = SEARCH_FIRST; diff != LAST_DEVICE; ){
        diff = w1_rom_search( diff, id );

        if( diff == PRESENCE_ERR ){
            return 1;

        }
        if( diff == DATA_ERR ){
            return 1;

        }
        w1_byte_wr( READ );             // read command
        temp = w1_byte_rd();           // low byte
        temp |= (uint)w1_byte_rd() << 8; // high byte
        if( id[0] == 0x10 )            // 9 -> 12 bit
        temp <<= 3;
        sprintf( s, "%4d%d",temp >> 4, 5* (((temp << 12) / 6553) & 01));
    }
    return 0;
}

```

serial.c

```
// Unterprogramme fuer serielle Kommunikation
#include <avr/io.h>
#include <stdio.h>
#include <stdlib.h>
#include "serial.h"

// fuer MEGA 1284P verwenden wir USART 0
void USART_init(void)
{

/** Initialisierung der seriellen Schnittstelle.<br>
 * mit 9600,8,n,1 bei 8 MHz
 * @param[in] keine.
 * @return Nichts.
 *
 */
    UBRR0H = 0;
    UBRR0L = 103;
    UCSR0A |= (1 << U2X0);
    UCSR0B |= (1 << RXEN0) | (1 << TXEN0);
    UCSR0C |= (1 << UCSZ00) | (1 << UCSZ01);
}

// Ein Byte senden
void USART_putc(uint8_t byte)
{
/** ein Byte seriell senden.
 * @param[in] zu sendendes Byte (uint8_t)
 * @return Nichts.
 *
 */
    while(!(UCSR0A&(1<<UDRE0))); //warten auf Datenregister empty
    UDR0=byte;
}

// Pruefen, ob Byte empfangen
uint8_t USART_byte_avail(void)
{
/** Pruefen, ob Byte empfangen wurde.
 * @param[in] keine.
 * @return 1, wenn ja, 0, wenn nein (uint8_t).
 *
 */
    if(bit_is_set(UCSR0A,RXC0))
        return 1;
    else
        return 0;
}

// Ein Byte empfangen
uint8_t USART_getc(void)
{
/** Ein Byte empfangen.
 * @param[in] keine.
 * @return empfangenes Byte (uint8_t).
 *
 */
    while(bit_is_clear(UCSR0A,RXC0)); // warten auf Empfang fertig
    return UDR0;
}
```

```

// printf String ohne Auto LF
void USART_puts(char *s)
{
/** String seriell ausgeben, ohne Auto Linefeed.
 *  @param[in] String (char *).
 *  @return Nichts.
 *
 */
while(*s != 0)
{
    USART_putc(*s);
    s++;
}
}

// Zahl von Typ float senden
// sges Gesamtstellen, nach Nachkommastellen, werden gerundet
void USART_putf(float zahl, int sges, int snach)
{
/** Zahl vom Typ float seriell ausgeben.
 *  @param[in] Zahl (float), Stellen (int), Nachkommastellen (int).
 *  @return Nichts.
 *
 */
char buffer[16];
dtostrf(zahl,sges,snach,buffer);
USART_puts(buffer);
}

// Zahl vom Typ int senden
// Int Zahl formatiert mit sges Stellen, Ausgabe
void USART_puti(int zahl, int sges)
{
/** Zahl vom Typ int seriell senden.
 *  @param[in] Zahl (int), Stellen (int).
 *  @return Nichts.
 *
 */
char buffer[17];
uint8_t I = 0,n;
char *z = buffer;
itoa(zahl,buffer,10);
while(*z != 0)
{
    I++; z++;} // Bufferlaenge I
for(n = I; n < sges; n++) USART_putc(' ');
USART_puts(buffer);
}

```

```

// eine Zahl vom Typ unsigned int senden
void USART_putui(unsigned int zahl, int sges)
{
    /** Zahl vom Typ unsigned int seriell senden.
     *  @param[in] Zahl (unsigned int), Stellen (int).
     *  @return Nichts.
     *
     */
    char buffer[17];
    uint8_t I = 0,n;
    char *z = buffer;
    utoa(zahl, buffer,10);
    while(*z != 0)
    {
        I++; z++;
    }
    for(n = I; n < sges; n++) USART_putc(' ');
    USART_puts(buffer);
}

// eine Zahl vom Typ unsigned int senden (Leerstellen = 0)
void USART_putui_0(unsigned int zahl, int sges)
{
    /** Zahl vom Typ unsigned int mit Leerstellen = 0 seriell senden.
     *  @param[in] Zahl (unsigned int), Stellen (int).
     *  @return Nichts.
     *
     */
    char buffer[17];
    uint8_t I = 0,n;
    char *z = buffer;
    utoa(zahl, buffer,10);
    while(*z != 0)
    {
        I++; z++;
    }
    for(n = I; n < sges; n++) USART_putc('0');
    USART_puts(buffer);
}

// eine Zahl vom Typ unsigned int hexadezimal senden
void USART_putui_hex(unsigned int zahl, int sges)
{
    /** Zahl vom Typ unsigned int hexadezimal seriell senden.
     *  @param[in] Zahl (unsigned Int), Stellen (int).
     *  @return Nichts.
     *
     */
    char buffer[17];
    uint8_t I = 0,n;
    char *z = buffer;
    utoa(zahl, buffer,16);
    while(*z != 0)
    {
        I++; z++;
    }
    for(n = I; n < sges; n++) USART_putc(' ');
    USART_puts(buffer);
}

```

```

// eine Zahl vom Typ unsigned int binaer senden
void USART_putui_bin(unsigned int zahl, int sges)
{
/** Zahl vom Typ unsigned int binaer hexadezimal senden.
 *  @param[in] Zahl (unsigned int), Stellen (int).
 *  @return Nichts.
 *
 */
char buffer[17];
uint8_t I = 0,n;
char *z = buffer;
utoa(zahl, buffer,2);
while(*z != 0)
{
    I++; z++;
}
for(n = I; n < sges; n++) USART_putc('0');
USART_puts(buffer);
}

// ein byte binaer senden
void USART_putb(uint8_t a)
{
/** Zahl vom Typ byte binaer seriell senden.
 *  @param[in] Zahl (uint8_t).
 *  @return Nichts.
 *
 */
USART_putui_bin(a,8);
}

// Ende Unterprogramme fuer serielle Kommunikation

```

Software Teil 14:

Es wird hier die Software für den Außensensor beschrieben. Es sind nebenbei die Software-Module rfm12 und 1wire notwendig.

main.h

```
#include <avr/io.h>
#include <util/delay.h>
#include <stdlib.h>
#include <stdio.h>
#include <avr\interrupt.h>

//uint8_t fehler = 0;

#define uchar unsigned char
#define uint unsigned int
#define bit uchar
#define idata
#define code

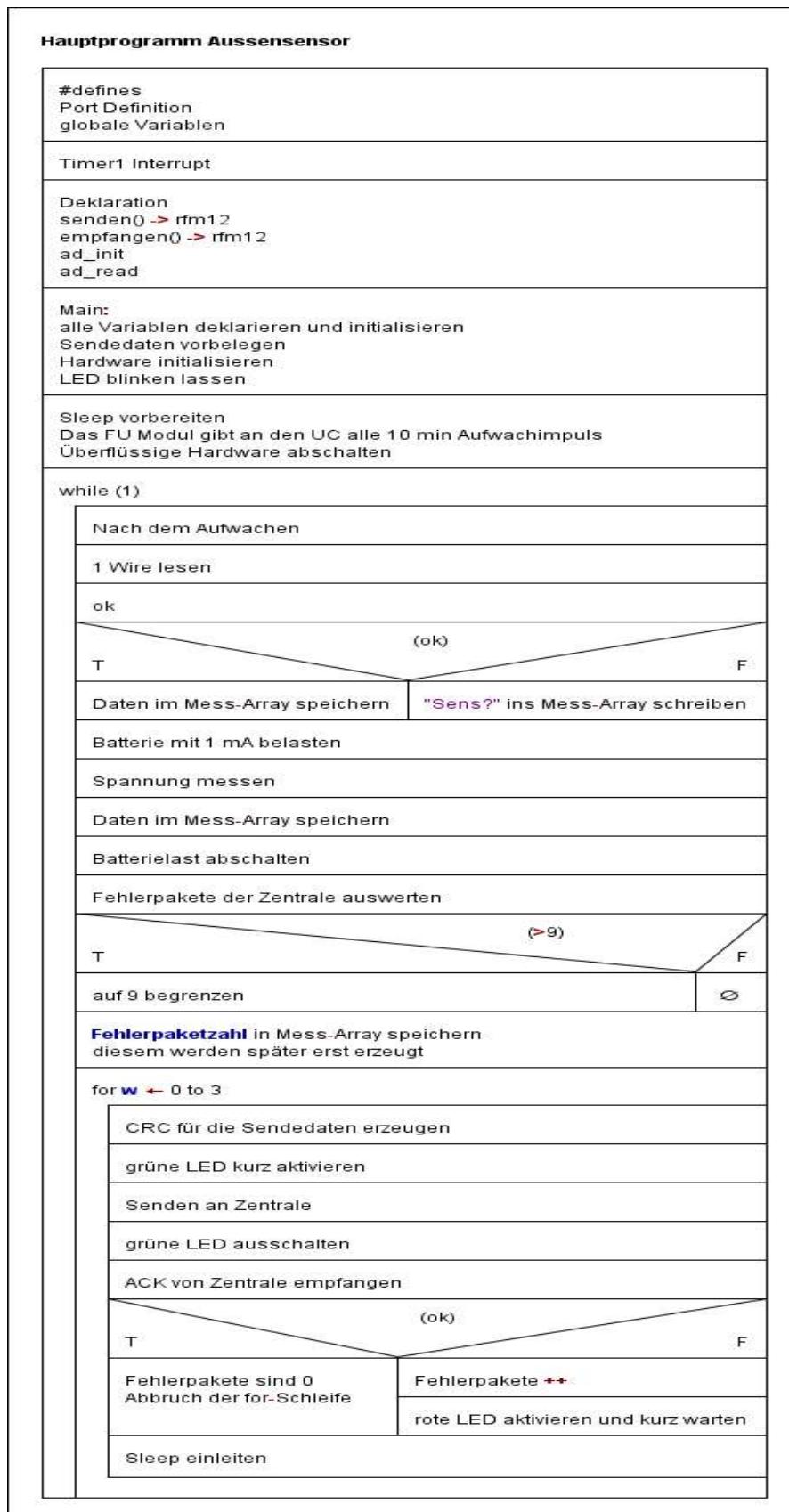
#define MATCH_ROM      0x55
#define SKIP_ROM0xCC
#define SEARCH_ROM     0xF0

#define CONVERT_T      0x44          // DS1820 commands
#define READ           0xBE
#define WRITE          0x4E
#define EE_WRITE0x48
#define EE_RECALL      0xB8

#define SEARCH_FIRST   0xFF          // start new search
#define PRESENCE_ERR   0xFF
#define DATA_ERR0xFE
#define LAST_DEVICE    0x00          // last device found
//                                0x01 ... 0x40: continue searching

void warten(void);
void selbsttest(void);
void zahl_ausgeben(int);
bit w1_reset(void);
uchar w1_bit_io(bit);
uint w1_byte_wr(uchar);
uint w1_byte_rd(void);
uchar w1_rom_search( uchar, uchar idata * );
void w1_command( uchar , uchar idata * );
uint8_t read_meas( char * );
void start_meas( void );
```

Auch hier bietet sich ein Struktogramm an.



main.c

```
// Diese Version fuer Temperatursensor mit Power Management
// i kleiner 10 µA im Sleep-Modus!
// gedacht fuer Temperaturueberwachung im Aussenbereich vom Haus
// 16.08.2014 jk
// 27.08.14 Sendezyklus alle 10 min
// Sendezyklus jede Minute
// wenn keine Antwort, nach 1s Sendevorschuss wiederholen.
// Fehlerzaehler erhöhen.
// nach 3 Versuchen Abbruch.
// mit CRC
// 

#include <avr/io.h>
#include <util/crc16.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <string.h>
#include <avr/sleep.h>

#include "rfm12.h"
#include "main.h"

#define true 1
#define false 0

#define IOPORT PORTB
#define DDRIOPORT DDRB
#define BATTMESS PB0

#define Geber 0x31

#define BATTMESS_AN IOPORT |= (1 << BATTMESS)
#define BATTMESS_AUS IOPORT &= ~(1 << BATTMESS)

static volatile uint8_t cnt = 0;
uint8_t timeouts = 0;
uint8_t fehlerpakete = 0;

void senden(uint8_t*, uint8_t);
uint8_t* empfangen(void);
void ad_init(void);
int ad_read(void);

ISR (TIMER1_OVF_vect)
{
    cnt++;
    TIFR |= (0x01 << TOV1); // Interrupt quittieren
}

void senden(uint8_t string[], uint8_t adr)
{
    uint8_t i;
    rfm12_tx((strlen((const char*)string)+1),adr, string);
    for(i = 0; i < 20; i++)
        rfm12_tick();
}
uint8_t* empfangen(void)
{
    uint8_t* buffer = NULL;
    uint8_t fehler_daten[] = "tmo";
    uint8_t finished = false;
```

```

cnt = 0;
while(finished == false)
{
    if (rfm12_rx_status() == STATUS_COMPLETE)
    {
        cnt = 0;
        timeouts = 0;
        buffer = rfm12_rx_buffer(); // rx buffer abspeichern
        finished = true;
    }
    if( cnt > 5)
    {
        cnt = 0;
        timeouts++;
        buffer = fehler_daten;
        finished = true;
    }
}
return buffer;
}

void ad_init(void)
{
    ADMUX=(1<<REFS1)|(1<<REFS0); // ADC Ref auf interne 2,56V
    ADCSRA=(1<<ADPS0)|(1<<ADPS1)|(1<<ADPS2); // ADC noch ausgeschaltet, 128 Bit Prescaler
}

int ad_read(void)
{
    ADCSRA |= (1 << ADEN); // AD ein
    // ADMUX = (0 << MUX0);
    ADCSRA |= 0x40; // Start AD
    while(ADCSRA & (1<<ADSC)); //warten bis konvertierung abgeschlossen
    ADCSRA &= ~(1 << ADEN); // AD aus
    return(ADCW);
}

int main ( void )
{
    uint8_t sende_daten_payload[] = "T:-125.3 B:1234 F:0 C:      ";
    uint8_t* empfangs_daten;
    uint8_t empf_string[6];
    uint8_t stringr[] = "ack";
    uint8_t i,w;
    uint8_t adr;
    int batteriespannung;
    uint16_t zt,t,h,z,e;
    uint16_t crc = 0;

    char messwert[30];

    LED_DDR |= _BV(LED_BITR);
    LED_DDR |= _BV(LED_BITG); //Kontroll-LEDs aktivieren (low-aktiv)

    DDRIOPORT |= (1 << PB0);
    ACSR = 0x80; // Analog Comparator abschalten

    //AVR sleep mode aktivieren
}

```

```

set_sleep_mode(SLEEP_MODE_PWR_DOWN);

_delay_ms(100); //kleine Verzoegerung, das Hochfahren des rfm12 braucht etwas Zeit

ad_init();

rfm12_init(); //init the RFM12

// Timer 1 aktivieren (fuer Time-Out Registrierung)

TCCR1B |= (1 << CS12); // Prescaler 256, alle 2,09s Overflow
TIMSK |= (1 << TOIE1); // Interrupt bei T1 overflow

sei(); //interrupts aktiv

// Wake Up Timer konfigurieren: Zeit = 4 * (2^8)ms = 1.024s
// rfm12_set_wakeup_timer(0x804);
// hier aber 10 * (2^10)ms = ca. 10s

// rfm12_set_wakeup_timer(0xA0A);

// fuer 1 Minute 2^10 * 59

// rfm12_set_wakeup_timer(0xA3B);

// fuer 10 min (600000 ms) 2 ^ 12 * 146 etwa 598016 ms

rfm12_set_wakeup_timer(0xC92);

// alle LED ausschalten

LED_PORT |= _BV(LED_BITR);
LED_PORT |= _BV(LED_BITG);

while (1) // lange Aussenschleife
{
    // Temperatur und Spannung lesen

    start_meas();
    if(read_meas(messwert) == 0)
    {
        sende_daten_payload[2] = messwert[0];
        sende_daten_payload[3] = messwert[1];
        sende_daten_payload[4] = messwert[2];
        sende_daten_payload[5] = messwert[3];
        sende_daten_payload[6] = '.';
        sende_daten_payload[7] = messwert[4];
    }
    else
    {
        sende_daten_payload[2] = 'S';
        sende_daten_payload[3] = 'e';
        sende_daten_payload[4] = 'n';
        sende_daten_payload[5] = 's';
        sende_daten_payload[6] = '?';
        sende_daten_payload[7] = '?';
    }
}

```

```

// Batteriestatus feststellen, mit etwa 1 mA messen

BATTMESS_AN; // fuer Spannungsmessung an
batteriespannung = ad_read();
BATTMESS_AUS; // fuer Spannungsmessung wieder ausschalten

e = batteriespannung;
z = batteriespannung/10;
h = batteriespannung /100;
t = batteriespannung / 1000;

sende_daten_payload[11] = (t % 10) + 0x30;
sende_daten_payload[12] = (h % 10) + 0x30;
sende_daten_payload[13] = (z % 10) + 0x30;
sende_daten_payload[14] = (e % 10) + 0x30;

if(fehlerpakete < 9)
{
    sende_daten_payload[18] = fehlerpakete + 0x30;
}
else
{
    sende_daten_payload[18] = 0x39;
}

// Strategie: wenn vorhergehende Uebertragung schief ging
// ohne Sleep nochmals senden, das Ganze 3 mal mit jew. 1 s Pause
// wuerde die Batterie mehr schonen als permanentes Warten auf Handschlag
// w = 0;

for(w=0; w<3 ;w++)
{
    LED_PORT &= ~_BV(LED_BITG); // gruene LED an

    // erst mal schoen lesbare CRC8 basteln

    crc = 0x00; // simple CRC
    for (i=0;i<19;i++)
        crc = _crc_ccitt_update(crc,sende_daten_payload[i]);

    e = crc;
    z = crc/10;
    h = crc /100;
    t = crc / 1000;
    zt = crc / 10000;

    sende_daten_payload[22] = (zt % 10) + 0x30;
    sende_daten_payload[23] = (t % 10) + 0x30;
    sende_daten_payload[24] = (h % 10) + 0x30;
    sende_daten_payload[25] = (z % 10) + 0x30;
    sende_daten_payload[26] = (e % 10) + 0x30;

    //Senden
    senden(sende_daten_payload,Geber);
}

```

```

    _delay_ms(50); // kurz warten, LED besser zu sehen

    LED_PORT |= _BV(LED_BITG); // gruene LED aus

    // Echo der Zentrale empfangen, oder auch nichts empfangen
    // bedeutet dann Time-Out

    empfangs_daten = empfangen(); // ist das "ACK" der Zentrale
    adr = rfm12_rx_type(); // aus dem empf. Pk die bestaetigte Abs-Adr. extr.
    rfm12_rx_clear(); // Empfangspuffer loeschen, wird nicht mehr gebraucht

    // Empfangsdaten untersuchen:
    // richtige Adresse zurueckgekommen?

    if( adr == Geber || timeouts > 0) // ja, dann auf "ACK" pruefen
    {
        for (i=0;i<4;i++) // davor in ein Array umkopieren, fuer strcmp
        {
            empf_string[i] = empfangs_daten[i];
        }
        empf_string[i] = '\0';

        if(strcmp((const char*)empf_string,(const char*)stringr) == 0)
        {
            LED_PORT &= ~_BV(LED_BITG); // alles ok, kurz gruene LED
            fehlerpakete = 0;
            w = 3; // Abbruch der for-Schleife
        }
        else
        {
            LED_PORT &= ~_BV(LED_BITR);
            // nein, weder richtige Adresse noch ack oder timeout
            fehlerpakete++;

        }
        _delay_ms(50); // vor Abschalten der LEDs warten (Sichtbarkeit)

        LED_PORT |= _BV(LED_BITR); // LEDs abschalten
        LED_PORT |= _BV(LED_BITG);
        // _delay_ms(1000);
    }

}
//*** Schlaufen vorbereiten ***
rfm12_power_down(); //zuerst den Empfaenger abschalten

// dann die CPU stillsetzen

while(!ctrl.wkup_flag)
{
    sleep_mode(); //solange kein Wake Up Flag vom Wake Up Timer kommt
}

```

```

// hier wachen wir wieder auf

ctrl.wkup_flag = 0; //Wake Up Flag quittieren

rfm12_power_up(); // Empfaenger wieder einschalten

_delay_ms(100); // etwas warten, bis alles wieder stabil ist ...

// wiederholen = 1;

} // große Schleife Ende
} // Programm-Ende (wird nie erreicht!)

```

Software Teil 15:

Für die Automatisierung unserer Anwendung benötigen wir 2 Linux-Skripte, 2 Cron-Jobs und eine Messdaten-Datei. Weiterhin eine Datei mit dem zuletzt gültigen Messarray. Dieses wird auf der WEB-Seite angezeigt. Die Messdaten-Datei wird ständig fortgeführt. Erklärungen hierzu: Die Zentrale sendet alle 10 Minuten, getriggert durch das Außensensor über die serielle Schnittstelle an den Raspberry PI. Dieser wartet auf die seriellen Daten. Dafür ist Skript Nr. 1 notwendig, er hat den Dateinamen „datenerfassung.sh“. Sein Inhalt:

```
# Skript zur Datenerfassung ueber RS 232 JK 14.01.2022
```

```
cat /dev/ttyAMA0 | while read i; do printf "%s_%s:$\n" $(date +"%d.%m.%Y") $(date +%X);
done >> /home/pi/datenerfassung.dat
```

Er macht folgendes: Die Daten der seriellen Schnittstelle /dev/ttyAMA0 werden von Zeile 1 bis 12 gelesen und jeweils am Beginn einer jeden Zeile mit Datum und Zeit markiert. Wichtig: durch die Anweisung (date +"%d.%m.%Y") wird das Datum erst richtig angezeigt, ohne diese wären die Trenner Schrägstriche. Das Ergebnis wird in der Datei „datenerfassung.dat“ gespeichert.

Um den Skript ausführen zu können, müssen wir das sogenannte x-Bit in seinen Dateirechten setzen; das geht mit der Anweisung: „chmod u+x datenerfassung.sh“. Ausgeführt wird er mit „./datenerfassung.sh &“. Da wir die Linux-Shell nicht konfiguriert haben und den Pfad für ausführbare Dateien nicht festgelegt haben, müssen wir vor dem Start eines Skripes sein Verzeichnis mit angeben, das machen wir mit „..“ Wer nachlesen möchte, kann eine Profildatei erstellen und den Pfad mit „export“ festlegen. Das „&“ dient dazu, den Prozess im Hintergrund laufen zu lassen, eine ausführbare Datei wird in Linux stets als Prozess behandelt, sie bekommt eine eindeutige Prozess-ID, welche man beim Start des Hintergrundprozesses mitgeteilt bekommt. Auch kann zu jeder Zeit mit der Anweisung „ps -ax“ die Prozessliste angezeigt werden. Einen Prozess kann man auch beenden, man macht dieses mit „sudo kill -9 <prozess-Nummer>“. Es ist für uns nicht sinnvoll, den Prozess in der Kommandozeile zu starten, denn wir vergessen das garantiert nach einem Neustart des Systems. Daher schreiben wir die Anweisung in einen Cron-Job, das ist vergleichbar mit dem Taskplaner von Windows. Linux hat einen Cron-Job für root und für jeden Benutzer einen eigenen. Der Aufruf von Cron unterscheidet sich, welchen wir verwenden wollen. Für root schreiben wir „sudo crontab -e“, für den Benutzer lediglich „crontab -e“. Wir werden zuerst nach dem bevorzugten Editor gefragt und wählen „nano“. Dann tragen wir in den Cron-Job von root in der letzten Zeile folgendes ein:

```

# Edit this file to introduce tasks to be run by cron.
#
# Each task to run has to be defined through a single line
# indicating with different fields when the task will be run
# and what command to run for the task
#
# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'any').
#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h dom mon dow   command
@reboot /home/pi/datenerfassung.sh

```

Der obere Teil liegt bereits vor und besteht aus Kommentaren. Die Anweisung unten bedeutet, dass nach dem Start unseres Betriebssystems der Skript „datenerfassung.sh“ gestartet wird. Wir müssen an dieser Stelle den kompletten Pfad der Anwendung angeben, da Cron schon ausgeführt wird, bevor das komplette System konfiguriert ist.

Es nützt uns nichts, die ganze Datenerfassung auf unserer Web-Seite auszugeben. Wir beschränken uns daher auf die letzten 12 Zeilen der Datenerfassung. Dazu benötigen wir Skript Nr. 2, wir nennen ihn „schieben.sh“:

```
tail -12 datenerfassung.dat > datenerfassung_ende.dat
```

Wir speichern diesen und machen ihn ausführbar. Der Skript liest die letzten 12 Zeilen aus der Datei „datenerfassung“ und schreibt sie in die Datei „datenerfassung_ende.dat“, wobei sie diese immer neu überschreibt.

Wir generieren einen Cron-Job für den User: „crontab -e“, hier tragen wir unten folgendes ein:

```
# Edit this file to introduce tasks to be run by cron.  
#  
# Each task to run has to be defined through a single line  
# indicating with different fields when the task will be run  
# and what command to run for the task  
#  
# To define the time you can provide concrete values for  
# minute (m), hour (h), day of month (dom), month (mon),  
# and day of week (dow) or use '*' in these fields (for 'any').  
#  
# Notice that tasks will be started based on the cron's system  
# daemon's notion of time and timezones.  
#  
# Output of the crontab jobs (including errors) is sent through  
# email to the user the crontab file belongs to (unless redirected).  
#  
# For example, you can run a backup of all your user accounts  
# at 5 a.m every week with:  
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/  
#  
# For more information see the manual pages of crontab(5) and cron(8)  
#  
# m h dom mon dow command  
*/10 * * * * /home/pi/schieben.sh
```

Der Cron-Job führt „schieben.sh“ alle 10 Minuten aus!

Der PHP-Code, welcher unter /var/www/html abgespeichert wird, hat den folgenden Inhalt, er wird unter „index.php“ gespeichert; Erklärungen folgen im Anschluß:

:

```
<!DOCTYPE html>  
<!-- Das ist die Version vom 15.01.2022 -->  
<html lang="de">  
<head>  
    <meta http-equiv=Content-Type content=text/html; charset=iso-8859-1>  
    <meta http-equiv="refresh" content="60; url=index.php" />  
    <title>Raspberry Pi Webserver</title>  
    <link rel="stylesheet" type="text/css" href="formate.css">  
</head>  
  
<body>  
<div align="center"><h1>Raspberry Pi Webserver</h1></div><br>  
  
<h1>Anzeige von Wetterdaten Anwesen xyz</h1>  
  
<?php  
  
$taus_info = implode('_', file('/home/pi/datenerfassung_ende.dat'));  
$taus_array = explode("_",$taus_info);  
  
$u_solarpanel = (float) substr($taus_array[27], ',', '.');  
$u_solarbatterie = (float) substr($taus_array[31], ',', '.');  
  
$i_solarpanel = (float) substr($taus_array[35], ',', '.');  
$i_solarbatterie = (float) substr($taus_array[39], ',', '.');  
$i_last = (float) substr($taus_array[43], ',', '.');
```

```
$watt_panel = $u_solarpanel * $i_solarpanel;
$watt_batt = $u_solarbatterie * $i_solarbatterie;
$watt_last = $u_solarbatterie * $i_last;

$watt_panel_ausg = (string) strtr($watt_panel,'.',',');
$watt_batt_ausg = (string) strtr($watt_batt,'.',',');
$watt_last_ausg = (string) strtr($watt_last,'.',',');

echo "<span style=\"color:#228811;\">";
echo "<br> am ".$taus_array[0];
echo " um ".$taus_array[1];
echo "<br> <br>";
```

```

echo "Messwerte im Haus <br> <br>";

echo "
<table>
  <tr><th>Sensortyp:</th><th>Messgroesse:</th><th>Messwert:</th><th>Einheit:</th></tr>
  <tr><td>DS18B20 OW</td><td>Temperatur</td><td>$taus_array[3]</td><td>°C</td></tr>
  <tr><td>HYT 939 I2C</td><td>Temperatur</td><td>$taus_array[7]</td><td>°C</td></tr>
  <tr><td>HYT 939 I2C</td><td>Luftfeuchte</td><td>$taus_array[11]</td><td>% rF</td></tr>
  <tr><td>KPY 10</td><td>Luftdruck</td><td>$taus_array[15]</td><td>hP</td></tr>
</table>
";

echo "<br>Messwerte im Freien <br><br>";

echo "
<table>
  <tr><th>Messort:</th><th>Messgroesse:</th><th>Messwert:</th><th>Einheit:</th></tr>
  <tr><td>Gartenhaus</td><td>Temperatur</td><td>$taus_array[19]</td><td>°C</td></tr>
  <tr><td>Gartenhaus</td><td>Spg. Bat.</td><td>$taus_array[23]</td><td>V</td></tr>
  <tr><td>Solarpanel</td><td>Spannung</td><td>$taus_array[27]</td><td>V</td></tr>
  <tr><td>Solarbatterie</td><td>Spannung</td><td>$taus_array[31]</td><td>V</td></tr>
  <tr><td>Solarpanel</td><td>Strom</td><td>$taus_array[35]</td><td>A</td></tr>
  <tr><td>Solarbatterie</td><td>Strom</td><td>$taus_array[39]</td><td>A</td></tr>
  <tr><td>Last</td><td>Strom</td><td>$taus_array[43]</td><td>A</td></tr>
  <tr><td>Solarpanel</td><td>Leistung</td><td>$watt_panel_ausg</td><td>W</td></tr>
  <tr><td>Solarbatterie</td><td>Leistung</td><td>$watt_batt_ausg</td><td>W</td></tr>
  <tr><td>Last</td><td>Leistung</td><td>$watt_last_ausg</td><td>W</td></tr>
</table>
";
echo "<br> <br>";
echo "Anlagenstatus:";

echo $taus_array[47];
if ($taus_array[47] != 0)
{
    echo "<br>Fehler aufgetreten!";
    echo "<br>0001 = Timeout";
    echo "<br>0010 = CRC-Fehler";
    echo "<br>0100 = Lo-Batt Aussen";
    echo "<br>1000 = Lo-Batt Solar";
}
?>

</body>
</html>

```

Wie wir in der Anleitung zu lighttpd entnehmen können, sollte die Berechtigung für unsere Web Präsenz entsprechend konfiguriert werden, das dient der Sicherheit.

```

sudo groupadd www-data
sudo usermod -G www-data -a pi
sudo chown -R www-data:www-data /var/www/html
sudo chmod -R 775 /var/www/html

```

„index.php“ sollte ebenfalls diese Berechtigungen aufweisen!

Im oberen Teil von „index.php“ unter <head> wird die WEB-Seite konfiguriert, ihr Aussehen hängt von „formate.css“ ab. Die Anweisung „refresh“ sorgt für die Aktualisierung der Seite alle 60 Minuten. Ein Beispiel einer (nicht besonders gut gelungenen) „formate.css“ ist hier aufgelistet, dort kann man noch vieles verbessern!

```
<style type="text/css">
<!--
BODY { margin: 10 10 10 10; background-color: white; text-align:justify; }
P, DIV, H1, H2 {font-family: verdana, arial; font-size: 10px;}
H1 { background-color: #D7DF01; padding-top:5px; padding-bottom:5px;
font-size: 16px; color: #004060; font-weight:bold; letter-spacing: 5px;
border-top: 2px solid #EEEEEE; border-left: 2px solid #EEEEEE;
border-bottom: 0px solid #004060; border-right: 2px solid #004060;}
H2 { background-color: #CCCCFF; padding-top:2px; padding-bottom:2px;
font-size: 8px; color: #004060; font-weight:bold; letter-spacing: 5px;
border-top: 2px solid #EEEEEE; border-left: 2px solid #EEEEEE;
border-bottom: 2px solid #004060; border-right: 2px solid #004060;}
P, H1, H2 {margin: 0px 0px 3px 0px;}
-->
</style>
```

Unter <body> wird eine Überschrift generiert.

Dann erfolgt das Umschalten auf PHP.

Die Datei „datenerfassung_ende.dat“ wird eingelesen und in das Array \$taus_array überführt. Hier spielen die Trenner „_“ eine Rolle, sie sorgen für die Indizierung des Arrays und der richtigen Ordnung seiner Inhalte. Die Arrayindices mit den Alphabetischen Tags werden hier nicht verwendet, ich hatte diese für die Fehlersuche eingeplant. Folgende 11 Zeilen beheben die Unstimmigkeit des deutschen Dezimal-Kommas und dem US Dezimalpunkt. In meinem Array ist das Komma verwendet, zum Rechnen mit Gleitkomma muss jedoch der Dezimalpunkt verwendet werden, daher die Umsetzung in den oberen 5 Zeilen. In den nächsten Zeilen wird Watt berechnet, Produkt aus Spannung und Strom. Um das Ganze wieder mit Dezimalkomma darstellen, wird in den folgenden 3 Zeilen wieder zurückverwandelt. Das war für mich eine Übung, man kann natürlich im Messarray gleich die Punkte verwenden. sorgt für grüne Schrift. Die folgenden 4 Zeilen extrahieren Datum und Zeit und bringen diese zur Anzeige. Es folgt eine Überschrift für die Tabelle der Messungen im Haus, danach diese Tabelle. Eine weitere Überschrift für die Messung im Freien folgt, dann diese Tabelle.

Mit diesen Voraussetzungen wird es gelingen, einen WEB Server zum Laufen zu bekommen und die Wetterdaten, sowie den Zustand der Solaranlage dort darzustellen. Natürlich lässt sich die Anordnung nach persönlichen Wünschen erweitern und anpassen.

Schlusswort

Mit diesen Beispielen soll gezeigt werden, wie mit einfacher, kostengünstiger Hardware anspruchsvolle Aufgaben gelöst werden können. Alleine in der Gestaltung der Software sind hier viele Möglichkeiten gegeben. Mit wenig Aufwand können die Programmbeispiele erweitert werden, man denke an die Ankopplung an einen PC über die im Mikrocontroller vorhandene serielle Schnittstelle. Wie das bewerkstelligt werden kann, erfährt der Leser über die Dokumentation der Firma Microchip. Mit dem Wissen, was er über die vorangegangenen Beispiele erworben hat, wird es ihm nicht schwerfallen, diese Änderungen vorzunehmen.

Auch die weiteren, im Mikrocontroller vorhandenen Komponenten, wie z.B. EEPROM, SPI und I2C-Schnittstelle, Analog-Komparator können erschlossen werden. Schließlich lässt sich, ähnlich wie das beim LCD Display erfolgt ist, weitere Hardware an das System anschließen, zum Beispiel Digital-Analogwandler oder eine Echtzeituhr.

Wer sich bis hierhin durchgearbeitet hat, wird über recht gute Erfahrungen mit der Programmiersprache C verfügen. In der Regel führt die Beschäftigung in diesem Metier dazu, dass oft auf Quellen anderer Programmierer zugegriffen wird. Dadurch werden weitere Erfahrungen gewonnen, welche uns ständig weiterbringen werden.

Daneben erweitert die Beschäftigung mit Elektrotechnik und Elektronik unseren Horizont und hilft uns, viele Zusammenhänge zu verstehen. Auch dort gilt: Nutze die Erfahrungen von Experten, sei es durch Arbeiten mit entsprechender Literatur – oder Internetseiten, es gibt derer viele!

Ganz besonderen Dank an Herrn Peter Dannegger, Herrn Peter Fleury und Herrn Peter Fuhrmann und „das Labor“, sie haben durch ihre sehr gute Arbeit mitgeholfen, die Theorie der Kommunikation mit diversen Sensoren besser zu verstehen.

Hoffentlich hat diese Zusammenstellung Ihnen viele neue Erfahrungen bieten können und Ihre Neugierde geweckt.

Berlin, im Dezember 2022

Joachim Kraatz

Verzeichnis der Fotos, Stücklisten, Schaltplänen und Programmlistings

Fotos:

Titelseite	\fotos\relaisplatine.jpg.....	
HF Leistungsmessgerät	\fotos\hf_powermeter_platine.jpg	60
Platine Zentrale	\fotos\wetterzentrale	71
Platine Außensensor	\fotos\außenmodul.jpg	74
Strom/Spannungswandler	\fotos\strom_spaltungswandler.jpg	77
Gehäuse Zentrale	\fotos\gehäuse_komplett.jpg	78

Stücklisten:

Projekt 1	3
Projekt 2	14
Projekt 7	29
Projekt 8	32
Option Quarztaktung	36
Projekt 10	52
Projekt 11	59
Projekt 12: WEB Server	65
Projekt 12: Solaranlage	70
Projekt 12: Zentrale	70
Projekt 12: 12 V zu 5 V Umsetzer	72
Projekt 12: Außensensors	73
Projekt 12: Netzrelais	75
Projekt 12: Strom/Spannungswandler	76

Verzeichnis der Fotos, Stücklisten, Schaltplänen und Programmlistings

Schaltpläne:

Projekt 1	\schaltpläne\simple_mega8.png	3
Mega8 DIL	\zusätze\mega8_dil.jpg.....	3
Taster	\zusätze\taster.jpg	9
Projekt 4	\schaltpläne\lcd_display.png	14
Projekt 7	\schaltpläne\relaisplatine.png	28
Projekt 8	\schaltpläne\netzankopplung.png	33
Projekt 9	\zusätze\DS18B20.jpg	37
Projekt 10	\zusätze\teiler.jpg	49
Projekt 10	\zusätze\shunt.jpg	51
Projekt 10	\schaltpläne\leistungsmesser.png	52
Projekt 11	\schaltpläne\hf_leistungsmesser.png.....	59
Projekt 11: Anschluss 5V	\zusätze\ad8307_platine.jpg.....	60
Projekt 12: Zentrale	\schaltpläne\zentrale.png	71
Projekt 12: 5V Versorgung	\schaltpläne\5v_regler.png	72
Projekt 12: Außensensor	\schaltpläne\aussensensor.png	74
Projekt 12: Netzrelais	\schaltpläne\netzrelais.png	75
Projekt 12: U/I Messmodul	\schaltpläne\strom-spannungsmessung.png	76
Projekt 12: Übersichtsplan	\gesamtplan1.vsd	77

Verzeichnis der Fotos, Stücklisten, Schaltplänen und Programmlistings

Listings:

Software Teil 1	LED leuchtet	5
Software Teil 2	LED blinkt	7
Software Teil 3	Port abfragen.....	10
Software Teil 4	LCD Ausgabe	15
LCD_setcursor, LCD_string	LCD steuern	16
Software Teil 5	Zähler inkrementieren	17
Software Teil 5	Zähler ausgeben	18
Software Teil 6	Encoder lesen	19
Software Teil 7	Zustandsautomat	21
Software Teil 8	8 Kanal Relaissteuerung	24
Software Teil 9	Netzfrequenz messen.....	33
Makefile	printf für float	38
Software Teil 10	Temperatur messen	39
Software Teil 11	Leistung messen	54
Software Teil 12	HF Leistung messen.....	61
WPA Suplicant	zu Raspberry PI Zero	66
Konfiguration Lighttpd	WEB Server	68
Software Teil 13	Wetterstation Zentrale	79
Software Teil 14	Außensensor	108
Software Teil 15	PHP und LINUX Skripte	115

Struktogramme:

Projekt 12: Zentrale	\zusätze\Hauptprogramm_Wetterzentrale.png	86
Projekt 12: Außensensor	\zusätze\Hauptprogramm_Aussensensor.png	109

Links und Quellenangaben:

1. <https://de.wikipedia.org/wiki/Microchip AVR>
2. <https://reichelt.de>
3. <https://www.geany.org/>
4. <https://de.wikipedia.org/wiki/Lubuntu>
5. <http://www.sax.de/~joerg/mfile/>
6. [ATmega8 – RN-Wissen.de](#)
7. https://de.wikipedia.org/wiki/The_C_Programming_Language
8. https://de.wikipedia.org/wiki/Boolesche_Algebra
9. <https://www.mikrocontroller.net/articles/AVR-GCC-Tutorial/LCD-Ansteuerung>
10. <https://www.mikrocontroller.net/articles/Drehgeber>
11. <https://de.wikipedia.org/wiki/HD44780>
12. <https://www.vde.com/de>
13. [https://de.wikipedia.org/wiki/Eagle_\(Software\)](https://de.wikipedia.org/wiki/Eagle_(Software))
14. <https://de.wikipedia.org/wiki/Z-Diode>
15. https://de.wikipedia.org/wiki/Dallas_Semiconductor
16. <https://www.mikrocontroller.net/articles/Burn-o-mat>
17. <https://www.mikrocontroller.net/topic/14792>
18. <https://de.wikipedia.org/wiki/Spannungsteiler>
19. <https://de.wikipedia.org/wiki/Operationsverstärker>
20. https://de.wikipedia.org/wiki/Analog_Devices
21. <https://www.box73.de>
22. <https://www.raspberrypi.com/software/>
23. <https://www.kampis-elektroecke.de/raspberry-pi/raspberry-pi-webserver/>
24. <https://github.com/das-labor/librfm12>
25. <https://www.gnu.org/licenses/old-licenses/gpl-2.0.html>