Kapitel 03: Über das Programmieren
Abs01: Wann ist ein Programm gut geschrieben?

Programmieren ist eine Ingenieurstätigkeit. Genauso, wie ein Maschinenbau-Ingenieur ein Gerät konstruiert, so konstruiert ein guter Programmierer sein Programm. Das Gerät des Ingenieurs soll

1. seinen Zweck fehlerfrei erfüllen und zwar unter möglichst vielen äusseren und inneren Bedingungen
2. aus möglichst wenig, möglichst billigem, aber bewährtem Material gebaut sein
3. vom Benutzer leicht zu bedienen sein; es soll möglichst offensichtlich sein, wie es zu bedienen ist
4. leicht zu reparieren sein

Das sind lauter Eigenschaften, die auch ein gut geschriebenes Programm erfüllt.

Dass ein Programm unter möglichst vielen Bedingungen seinen Zweck erfüllt, ist schwieriger zu erreichen als bei einer Maschine. Das kommt von den vielen Baugruppen, aus denen so ein MC auf dem Chip zusammengesetzt ist. Zunächst scheinen diese vielen Baugruppen ein Segen zu sein, denn durch sie werden dem Programmierer die vielen Register (allein an 8-Bit-Rechenregistern besitzt der ATmega16 schon 32 Stück) und die zahlreichen Umgebungsfunktionen (serielle Schnittstelle, TWI-Bus-Schnittstelle, SPI-Bus-Schnittstelle, Zeitzähler, usw.) beschert. Sie sind aber ein Fluch; jedes Register kann einen fehlerhaften Wert enthalten oder fehlerhaft eingestellt sein und damit die Fehlfunktion des Programms verursachen. Je mehr Register - desto mehr Fehlermöglichkeiten für den Programmierer.

Es gibt aber noch einen anderen Aspekt dieser Forderung: Erfahrungsgemäss sind Geräte besonders zuverlässig, die aus bereits bewährten Einzelteilen aufgebaut sind. Diese Erfahrung gilt auch für Programme. Deshalb lohnt es sich immer, Programme in Teile zu zerlegen, die jedes für sich, bestimmte Aufgaben erfüllen. Z.B. speichern (ein- und auslesen) von Daten im RAM, konfigurieren von Umgebungsfunktionen (z.B. der UART, TIMER0), Empfang und Senden von Botschaften über die serielle Schnittstelle usw.. Voraussetzung für so eine Zerlegung ist, dass man die Aufgabe des Programmteils klar eingrenzen können muss. Wenn dann dieselbe Aufgabe in einem anderen Programm wieder gebraucht wird, kann man einfach die entsprechenden bewährten Programmteile in das neue Programm hineinkopieren. Das AVRStudio unterstützt diese Arbeitsweise, indem es die „.include"-Pseudoanweisung bereitstellt. Damit werden ganze Dateien mit Assembler-Programmtext in ein Programm eingefügt. Das Kopieren der Programmteile ins Hauptprogramm bleibt einem dadurch erspart. Ein Programmteil, dessen Assembler-Programmtext in eine separate Datei ausgelagert sind, heisst in diesem Lehrgang „Modul" oder „Programmmodul".

Die zweite Anforderung, der sparsame Umgang mit Material, stellt für den Amateurprogrammierer heute keine so wichtige Anforderung mehr dar. Material - das ist für ein Assemblerprogramm gleichbedeutend mit der Anzahl Programmzeilen. Im ATmega16 hat er Platz für ganze 16k Programmzeilen zur Verfügung; da passt schon eine ganze Menge Programm hinein. Der Zwang, sparsam mit Programmzeilen umzugehen, ist dementsprechend gering. Beim Berufsprogrammierer ist das anders: Wenn es möglich ist, das Programm so zu verkleinern, dass es in einem billigeren MC mit kleinerem Programmspeicher unterzubringen, dann ist das ein wichtiger Kostenfaktor. Ausserdem bedeutet jede eingesparte Programmzeile eine Fehlermöglichkeit weniger.

Für den Amateur bleibt es natürlich eine sportliche Herausforderung, seine Programme möglichst genauso kompakt und schnörkellos zu schreiben, wie der Profi - solange die 4. Anforderung (sie läuft auf gute Verständlichkeit des Programms hinaus) dadurch nicht in Mitleidenschaft gezogen wird. Kürzere Programme sind ja natürlich, wenn nicht auch an den Kommentaren gespart wird, übersichtlicher. Zudem erfordert bei manchen Programmen es der Zweck, bestimmte Zeiträume einzuhalten. Und das geht viel leichter, wenn das Programm kurz ist. - Es gibt also viele Gründe, weshalb auch der Amateur trotz des reichlich und preiswert vorhandenen Programmspeichers seine Programme kurz und bündig zu halten.

Die Bedienung für den Benutzer einfach zu machen, ist beim Programmieren viel schwieriger als beim Maschinenbau. Oft ist der Teil des Programms, der die Bedienung organisiert, viel umfangreicher als der, der dem eigentlichen Zweck dient. Einfache Bedienung bedeutet nämlich auch, dass das Programm es dem Bediener nicht übelnehmen darf, wenn er etwas Unerwartetes oder Falsches macht. Und der Einfallreichtum der Bediener hinsichtlich der Reihenfolge und Art der Bedienung ist schier unbegrenzt. Grundsätzlich muss man davon ausgehen, dass alle Möglichkeiten die die Bedienelemente des Programms bieten - auch die ungeplanten - auch benutzt werden. Das ist nichts anderes, als eine andere Formulierung von Murphy's Gesetz: Jede Fehlermöglichkeit, die nicht durch besondere Massnahmen verriegelt ist, wird früher oder später auch eintreten. Für das Programmieren bedeutet das: Argumente, wie „das wird schon nicht vorkommen" oder „wer macht denn schon sowas" gelten nicht. Fehlerfrei kann ist nur das Programm, in dem jeder Fehler, der möglich ist, verhindert wird oder - mindestens - seine Folgen in vorhersehbare Bahnen gelenkt werden.

Geräte sind dann gut zu reparieren, wenn es eine gute Beschreibung gibt, und die Teile gut zugänglich sind. Die Zugänglichkeit ist ein grosser Vorteil des Programmierens in Assembler: Bei Hochsprachen ist der Compiler eine grosse, unzugängliche black box. Ein gute Beschreibung in Form von Kommentaren zu einzelnen Programmteilen und -zeilen ist etwas, was man sich antrainieren muss. Es ist unglaublich, wie schnell man wieder vergisst, was man sich beim Schreiben einzelner Programmabschnitte so alles gedacht hat. Ich habe mir deshalb z.B. angewöhnt, z.B. für jede Prozedur, die mit einer „call"-, „rcall"-, „icall"-Anweisung aufgerufen werden kann, einen Kommentarkopf mit folgender Struktur zu schreiben (hier am Beispiel der Funktion „FIFO_READ1"):

```
/*-----------------------------------------

          PROZEDUR FIFO_READ1

Liest ein Datenwort (1 Byte) aus einer FIFO.

Eingangsvariablen
          zh:zl:   enhält den Zeiger auf den FIFO-Steuerblock

Ausgangsvariablen
          zh:zl     : unverändert
          r16       : enthält das Datenbyte, das aus der FIFO ausgelesen
                              wurde
          SREG (T-FLAG)
                    T = 1 : Funktion erfolgreich beendet
                    T = 0 : Fehlschlag: FIFO war leer

geänderte Register
          r16

geänderte IO-Register
          SREG

Zyklen
 65       normaler Ablauf
 65       letztes Auslesen vor FIFOCTL_LEER
 36       Auslesen bei FIFOCTL_LEER
*/
```

Der erste Teil erklärt die Funktion des Programmteils, in diesem Fall einer Prozedur. Anschliessend werden die Register aufgezählt, in denen der Prozedur Eingangswerte übergeben werden, bzw. in denen die Prozedur ihre Ergebnisse an das aufrufende Programm zurückliefert. Dann wird aufgezählt, welche Register und IO-Register am Ende des Ablaufs der Prozedur geänderte Werte enthalten. Unter IO-Register werden alle Register verstanden, die im Datenblatt des ATmega auf Seite 334 und 335 (Kapitel „Register Summary") aufgezählt sind. Abschliessend sind noch die Anzahl Prozessorzyklen aufgezählt, die die Prozedur unter verschiedenen Bedingungen verbraucht. Mit diesen Angaben, so hat sich gezeigt, kann man die Prozedur in allen möglichen Anwendungen leicht wiederverwenden.

Man kann den Nutzen solcher Beschreibungen gar nicht überschätzen; ganz besonders dann, wenn man Programmteile als Module wiederverwendet. Es zeigt sich auch schnell, dass es sich allemale lohnt, zu jedem Modul eine ausführliche Beschreibung seiner Aufgaben und Bestandteile, eine eigentliche Gebrauchsanweisung, zu schreiben.

Die grösste Fehlerquelle beim Programmieren sind die schier unbegrenzten Möglichkeiten, die moderne MC und mit ihrem Umfang an Assembleranweisungen dem Programmierer bieten. ATMEL hat bei den ATmega-Typen schon den Weg eingeschlagen, die Anzahl der Anweisungen möglichst gering zu halten (RISC -„Reduced_Instruction_Set_Controller"). Trotzdem kann man damit sehr viel Unfug anstellen, wenn man sich nicht mit ein paar Regeln freiwillig einschränkt. Auf Vorschläge für einen bewährten Satz von Einschränkungen werde ich im Laufe des Lehrgangs jeweils an den Stellen eingehen, an denen ich ihren Nutzen gut vorführen kann.

Das Beispiel des Ingenieurs soll auch noch ein anderes Vorgehen nahelegen: Die Planung. Die meisten Amateurprogrammierer können es gar nicht erwarten und fangen sofort an drauflos zu programmieren, sobald sie sich eine neue Aufgabe gestellt haben. Es lohnt sich aber sehr, voher Papier und Bleistift zur Hand zu nehmen, und für das neue Programm vorher schon eine Struktur zu planen. Die ist dann natürlich nicht unumstösslich, aber es ist sehr nützlich, beim Programmieren eine vorgeplante Linie zu verfolgen.

Die erste Frage ist immer, in welche Teilaufgaben man das Programm zerlegen kann. Das richtet sich zum Teil auch danach, für welche Teilaufgaben bereits Module zur Verfügung stehen. Wenn es noch nichts Vorgefertigtes gibt, dann empfiehlt es sich, für jede Teilaufgabe ein Modul zu entwickeln. Und zwar nicht im Hauptprogramm, sondern mit einem eigenen Testprogramm und in einem eigenen Verzeichnis. Es ist viel einfacher, in so einem Testprogramm Fehler zu jagen, wenn man sich dabei nicht auch noch durch die diversen Teile des späteren Hauptprogramms hinduch zu wühlen braucht. Erst wenn das Modul gründlich getestet ist, wird es ins Hauptprogramm eingebunden und getestet, ob es sich dort auch so wie gewünscht verhält.

Das AVRStudio unterstützt die Fehlersuche mit seiner Simulatorfunktion. Man kann in der Simulatorumgebung das Programm schrittweise oder von Haltepunkt zu Haltepunkt laufen lassen und dabei den Inhalt jeden Registers und jeder Speicherzelle im RAM, EEPROM oder im Programmspeichern beobachten. Davon werden wir noch ausgiebig Gebrauch machen.

Zur Planung gehört aber auch, das Programm seine Aufgabe nach bewährten Methoden (Algorithmen) erledigen zu lassen. Dazu schadet es nicht, sich ein paar Bücher über Programmieren zuzulegen; als Einstieg empfehle immer eines der Hauptwerke von Niklaus Wirth (Wirth, Niklaus, „Algorithmen und Datenstrukturen" / von Niklaus Wirth - 3. überarbeitete Auflage - Stuttgart: Teubner, 1983, ISBN 3-519-02250-8). Er geht nicht nur auf Standard-Algorithmen und -Datenstrukturen ein, sondern auch auf die Fehlermöglichkeiten, die sich dem Programmierer bieten, und wie er ihnen am Besten entgeht. Eine seiner Grosstaten war die Entwicklung der Sprache Pascal, in der es keine „GOTO"-Anweisung gab. Damit wurde dem damals noch weit verbreiteten „Spaghetti"-Programmierstil (die Pfade, die die GOTO-Sprünge kreuz und quer in die Programme legten, machten sie so verwickelt und verworren wie einen Teller voll weichgekochter Spaghetti) ein Riegel vorgeschoben.