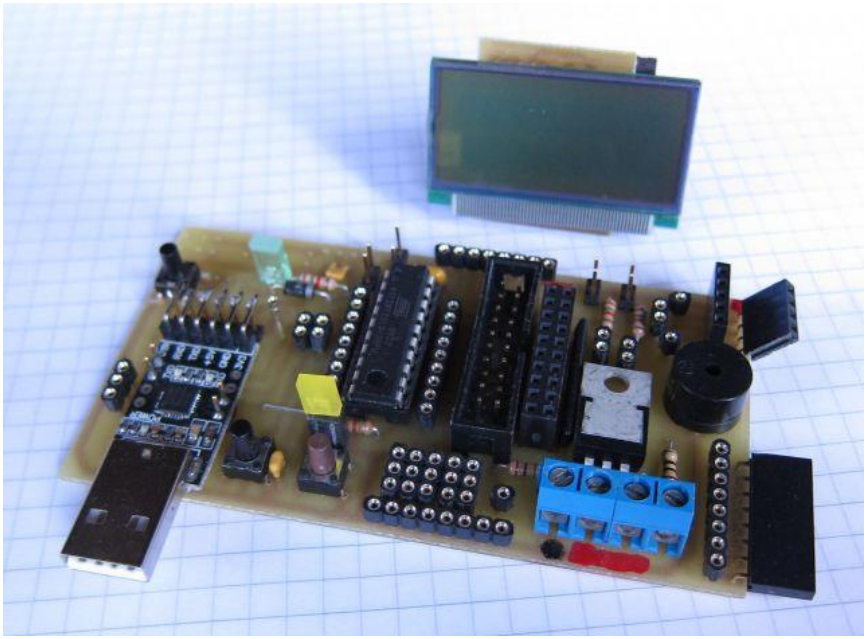


Das ATTINY-Projekt

Wieso FORTH?

Georg Heinrichs

Contents http://www.g-heinrichs.de/wordpress/index.php/attiny/

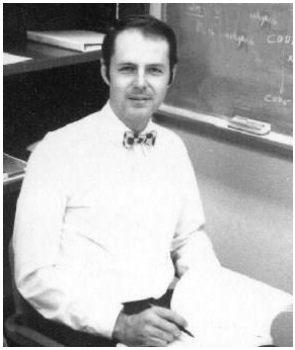
Tutorials:

Software:

1 Wie FORTH entstand

Forth wurde von Charles H. Moore 1969 entwickelt. FORTH weist eine Reihe von Eigentümlichkeiten auf, die es stark von herkömmlichen Programmiersprachen unterscheidet. FORTH stellt i. A. nicht nur eine Entwicklungsumgebung, sondern auch ein Betriebssystem dar.

Diese Eigentümlichkeiten lassen sich gut aus der Entstehungsgeschichte erklären. Moore hatte zur Steuerung des Teleskops einer Sternwarte einen Rechner ohne Software gekauft. Er hatte sich vorgenommen, alle Komponenten selbst zu programmieren, die für eine komfortable Programmierung und den Betrieb des Rechners notwendig sind. Hierzu gehören ein Betriebssystem, eine Hochsprache und eine Entwicklungsumgebung. All diese Komponenten wurden innerhalb eines einzelnen Programms verwirklicht – dem Forth-System.



Moore erzählte später selbst: *Ich entwickelte FORTH im Laufe mehrerer Jahre als eine Schnittstelle zwischen mir und den Computern, die ich programmierte. Die traditionellen Sprachen lieferten nicht die Leistungsfähigkeit, Einfachheit oder Flexibilität, die ich wollte. Ich missachtete viele geltende Weisheiten, um exakt die Fähigkeiten einzubauen, die ein produktiver Programmierer benötigt. Die allerwichtigste davon ist die Möglichkeit, neue Eigenschaften*

hinzuzufügen, die später einmal notwendig werden. Als ich zum ersten Mal die Ideen, die ich entwickelt hatte, zu einer Einheit zusammenfasste, arbeitete ich auf **Abbildung 1** *einer IBM 1130,*

Beispiel Quelle: http://computermuseum.informatik.uni-stuttgart.de/dev/ibm1130/ibm1130.html

einem Computer der "dritten Generation". Das Ergebnis schien mir so leistungsfähig, dass ich es für eine "Sprache der vierten Computergeneration" hielt. Ich würde sie FOURTH genannt haben, aber die 1130 erlaubte nur eine Kennung mit 5 Buchstaben. So wurde aus FOURTH FORTH, immerhin ein nettes Wortspiel. (forth = vorwärts)

(Zitiert nach L. Brodie: FORTH)

1 Wie MikroForth entstand

Eines Tages kam mein Sohn zu mir - er war gerade 14 Jahre alt - und fragte mich, wie man einen Compiler herstelle. Er würde gerne einen selbst programmieren. Nun hatte ich einmal gelesen, wie man Compiler baut; aber das war schon viele Jahre her und das meiste hatte ich wieder vergessen. Allerdings hatte ich sehr wohl noch in Erinnerung, dass der Compilerbau schon etwas komplexer ist und kaum etwas für einen 14-Jährigen. Und das sagte ich ihm dann auch.

Aber er ließ nicht locker. Einige Monate später - ich arbeitete gerade an einem Konzept für eine Mikrocontroller-Fortbildung - kamen wir auf die Idee, einen Compiler für den Mikrocontroller Attiny 2313 zu programmieren. Als Sprache wählten wir FORTH, nicht zuletzt wegen der einfachen Grundstruktur.

Unser FORTH-Compiler sollte allerdings nicht auf dem Mikrocontroller selbst laufen, sondern auf einem PC: Dieser sollte den FORTH-Code in Maschinencode umsetzen, welcher dann auf dem Attiny hochgeladen werden sollte. Der Compiler selbst ist relativ einfach, er greift auf eine Datenbank zurück, in der sich eine Sammlung von A-Wörtern (Befehlsfolgen in Maschinencode) und F-Wörtern (Befehlsfolgen in FORTH-Code) befindet. Da es uns hauptsächlich um das Prinzip ging, haben wir auch nicht alle gängigen FORTH-Wörter (Befehle) implementiert, sondern nur einen kleinen Bruchteil.

Es war äußerst lehrreich, derartige Programmschnipsel zu schreiben, und ich kann nur jedem raten, dies einmal selbst zu versuchen. Mikro-FORTH macht dies möglich; denn es ist ein offenes System: Die Datenbank kann nach eigenen Vorstellungen beliebig verändert und erweitert werden. Dies entspricht ja gerade auch der Moore'schen Leitvorstellung.

Faszinierend für uns war insbesondere folgender Umstand: Als wir mit dem Projekt begannen, kannten wir nur einige wesentliche Eigenschaften von FORTH. Später - als das Projekt fast fertig war -

haben wir einmal recherchiert, wie FORTH im Original beim Compilieren vorgeht. Und siehe da - wir fanden einige unserer Ideen wieder.

Was die Betrachtung von FORTH leistet

- Verständnis für einfachen Compiler
- Verständnis für Assembler und Maschinencode
- Verständnis und Übung im Umgang mit Stapeln
- Verständnis mit der Übergabe von Parametern
- last but not least: effizienten Maschinencode

2 Einstieg in MikroForth

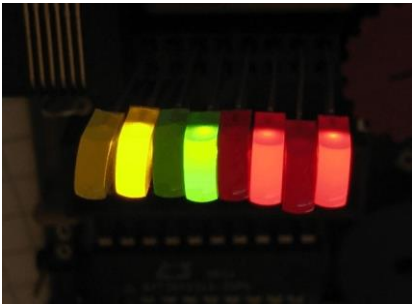


Abbildung 1

MikroForth ist ein FORTH-Compiler für den Attiny 2313. Damit ist gemeint: Das Programm wird auf einem PC eingegeben und kompiliert. Der dabei erzeugte HEX-Code wird anschließend mit einem Brennprogramm auf den Attiny übertragen.

Im Stile einer Schritt-für-Schritt-Anweisung wollen wir in diesem Kapitel zeigen, wie man mit MikroForth umgeht. Dazu werden wir mit MikroForth den Attiny so programmieren, dass er die Leuchtdioden an Port B in einem solchen Muster wie in Abb. 1 aufleuchten lässt. Die wenigen benötigten FORTH-Sprachelemente werden wir einfach angeben; wie sie funktionieren - und vor allem: wie man damit eigenständig Programme entwickelt, das werden wir dann in den folgenden Kapiteln darlegen.

Beginnen wir ganz vorne: mit der Installation von MikroForth. Kopieren Sie dazu einfach das Forth-Verzeichnis in das Programme-Verzeichnis (oder auch ein anderes Verzeichnis Ihrer Wahl). Öffnen Sie nun dieses Verzeichnis und starten Sie das Programm Forth2.

Während das Programm startet, lädt es die Datei “forthvoc.vok” mit dem Vorrat an Forth-Befehlen.

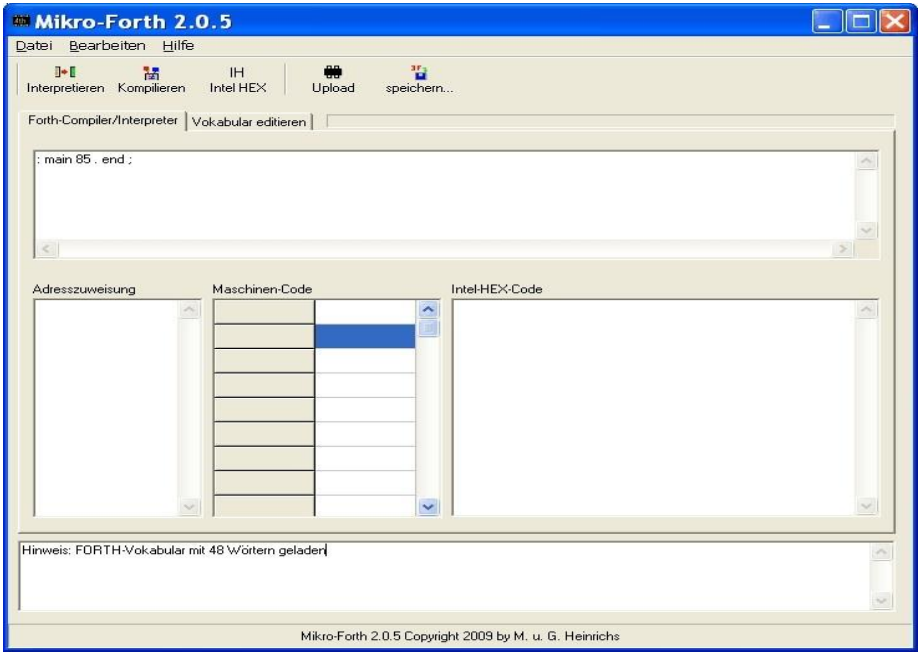


Abbildung 2

Unser erstes Programm besteht nur aus einer einzigen kurzen Zeile:

```
: main 85 . end ;
```

Geben Sie diesen Forth-Quelltext ganz oben im Formular ein (Abb. 2). Achten Sie auf die Leerzeichen zwischen den einzelnen Befehlen, insbesondere hinter dem Doppelpunkt und vor dem Semikolon; die Eingabe braucht man nicht mit der RETURN-Taste ⏎ abschließen.

Unser Programm gibt zunächst die Zahl 85 am Port B aus; da die Zahl 85 im Zweiersystem als 01010101 geschrieben wird, sollte dies das gewünschte Muster bei den LEDs erzeugen (vgl. Abb. 1). Anschließend führt das Programm eine Endlosschleife aus.

Als nächstes betätigen wir die “Interpretieren”-Schaltfläche. Wir erhalten die folgende Warnung:

Abbildung 3

Was bedeutet das?

“.” oder “`end`” sind sogenannte **Wörter**; diese stellen Befehle oder Befehlsfolgen dar, die der Mikrocontroller ausführen soll. Die Gesamtheit aller Wörter bezeichnen wir als **Forth-Vokabular**. Beim Interpretieren der eingegebenen Zeile wird hier dem Vokabular ein neues Wort “`main`” hinzugefügt, welches die Zahl 85 und die Wörter “.” und “`end`” zusammenfasst. Offensichtlich existierte schon ein Wort “`main`” imVokabular. Wie wir noch sehen werden, ist dieses Erstellen neuer Befehle ein wesentliches Konzept der Programmiersprache FORTH.

Da das alte “`main`”-Wort durch unser neues ersetzt werden soll, klicken wir bei dem Warnhinweis in Abb. 3 auf “Ja”.

Aus diesem neuen Wort “`main`” muss nun Maschinencode für den Mikrocontroller erzeugt werden. Dazu betätigen wir die “Kompilieren”-Schaltfläche. Unser FORTH-Compiler besorgt sich aus der Datei “forthvoc.vok” die Programmschnipsel für die einzelnen Bestandteile von “`main`”, also für die Wörter “.” und “`end`” und fügt sie zu einem Gesamtprogramm zusammen.

Im Adresszuweisungsbereich (Abb. 4) erkennt man die Zuweisung dieser Unterprogramme zu bestimmten Programmspeicheradressen; die Zeile “ **.** ? `$001A` “ bedeutet z.B.: Das Unterprogramm für “.”,

welches für die Ausgabe auf Port B verantwortlich ist, beginnt bei der Adresse 26 = $1A. Das gesamte Programm wird dann im Maschinencodebereich angezeigt. Unter jeder Adresse finden wir ein Maschinencode-Wort, bestehend aus 2 Byte.

Im Logbuch entdecken wir weitere Einträge; auf deren Bedeutung werden wir zu einem späteren Zeitpunkt eingehen.

So wie es imMaschinencodebereich angezeigt wird, so wird das Programm auch im Mikrocontroller abgelegt (wenn man davon absieht, dass im Speicher des Mikrocontrollers das höherwertige Byte eines Maschinencode-Wortes nicht wie hier vor, sondern hinter dem niederwertigen steht). Die meisten Brennprogramme benutzen allerdings ein anderes Format, welches noch zusätzliche Kontrollbytes besitzt: das Intel-HEX-Format. Mit der Schaltfläche “Intel-HEX” wandeln wir deswegen nun unseren Maschinencode in dieses Format um. Der Intel-HEX-Code erscheint sogleich im Intel-HEX-Bereich (Abb. 4).

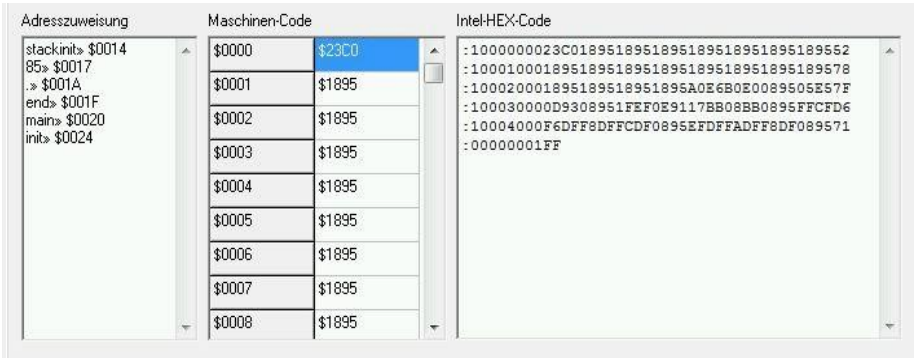
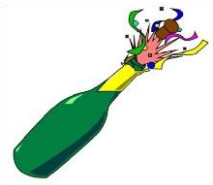


Abbildung 4

Zu guter letzt müssen wir den Code noch in den Attiny laden. Bei unserer Attiny-Platine kommt wieder das Uploader-Programm zum Einsatz. Wir rufen es auf, indem wir die “Upload”-Schaltfläche betätigen; der Intel-HEX-Code wird dann automatisch vom Uploader-Programm

übernommen und kann auf dem üblichen Weg in den Attiny übertragen werden.

Und wenn Sie keinen Eingabefehler gemacht haben, die LEDs korrekt in die Buchsen gesteckt wurden, die Batterie noch voll ist und die Übertragung reibungslos funktioniert hat, ja - dann sollte auch das Bitmuster aus Abb. 1 tatsächlich angezeigt werden und wir gratulieren Ihnen zur erfolgreichen Implementation Ihres ersten FORTH-Programms!



3 Wörter definieren

Damit ein Mikrocontroller eine bestimmte Aufgabe erfüllt, muss man ihmentsprechende Befehle geben. Bislang hatten wir dazu die Programmiersprache BASCOM oder den Assembler von AVR benutzt. Jetzt soll gezeigt werden, wie man hierzu unser MikroForth-System einsetzen kann.

FORTH-Befehle werden **Wörter** genannt. Derartige Wörter kann man zu Befehlsgruppen zusammenfassen; so entstehen neue Wörter. Die Gesamtheit aller Wörter, welche FORTH zur Verfügung stehen, bezeichnet man als **Vokabular**. Wenn man ein neues Wort herstellt, bedeutet dies letztlich eine Erweiterung des Vokabulars.

Am Beispiel eines Ampelprogramms wollen wir dies verdeutlichen. Zur Vereinfachung lassen wir dabei im Folgenden die in Deutschland übliche Rot-Gelb-Phase weg.

```
: ampelzyklus rotphase grünphase gelbphase ;
```

Unser FORTH-Compiler arbeitet grundsätzlich in zwei Schritten. Im ersten Schritt wird der eingegebene Quelltext interpretiert. Der Teil des FORTH-Programms, welcher dafür zuständig ist, wird **Interpreter** genannt. In unserem Fall stößt der Interpreter zunächst auf den Doppelpunkt; dieser zeigt ihm, dass ein neues Wort mit dem Namen `ampelzyklus` erzeugt werden soll. Dieses Wort setzt sich aus den folgenden Befehlen `rotphase`, `grünphase` und `gelbphase` zusammen. Das Semikolon zeigt dem Interpreter das Ende der Befehlsfolge an.

Da unser Interpreter grundsätzlich nur eine Wortdefinition pro Zeile zulässt, könnte man eigentlich auf das Semikolon verzichten. Andere FORTH-Compiler lassen aber auch mehrere Wortdefinitionen pro Zeile zu; da wird das **Semikolon** als **Begrenzer** unverzichtbar.

Ein neues Wort wird demnach allgemein so definiert:

Doppelpunktdefinition

`:` <Name des neuen Wortes> <Befehlsfolge mit bereits definierten Wörtern> `;`

Sämtliche Wörter - auch der Doppelpunkt und das Semikolon - müssen dabei durch (mindestens) ein Leerzeichen getrennt werden.

Testen wir nun unsere erste Wort-Schöpfung: Wir starten das Programm Forth2 und geben den FORTH-Quelltext ein. Groß-Klein-Schreibung spielt für MikroForth übrigens keine Rolle. Anschließend betätigen wir die Interpretieren-Schaltfläche. Im Statusfeld am unteren Rand des Forth2-Formulars erscheinen sogleich die folgenden Meldungen:

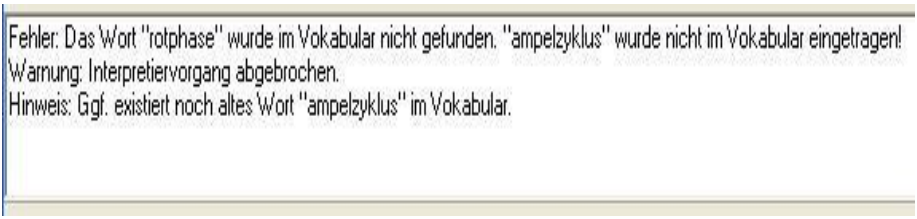


Abbildung 5

Was haben sie zu bedeuten? Die erste Meldung ist eine Fehlermeldung. Fehler führen in der Regel zum Abbruch eines Vorgangs. In diesem Fall weist die nächste Warnung darauf hin, dass der Interpretiervorgang abgebrochen wurde.

Um unseren Fehler beseitigen zu können, müssen wir seine Ursache finden. Offensichtlich kennt unser FORTH-System das Wort `rotphase` nicht. Das ist nicht schlimm, denn wir können diesen

“Fehler” beseitigen , indem wir die Definition von `rotphase` nachholen. Dazu fügen wir *vor* der Definition von `ampelzyklus` die folgende Zeile ein:

```
: rotphase rotesLicht an warte rotesLicht aus ;
```

Jetzt beschwert sich unser FORTH-System nicht mehr über das nicht gefundene Wort `rotphase`, dafür aber meldet es, dass es das Wort `rotesLicht` nicht finden kann. Also müssen wir auch dieses Wort noch definieren. Ähnliches gilt für die Wörter `grünphase`, `gelbphase`, `an`, `warte`, `aus` sowie die Wörter `gelbesLicht` und `grünesLicht`.

All diese Definitionen liegen schon fix und fertig in der Datei ampel.frth vor. Öffnen Sie diese Datei mit “Datei - öffnen”. Der Quelltext sieht dann so aus:

```
: initialisierePortB 7 DDRB ;
: warte 3 wait ;
: rotesLicht 2 ; : gelbesLicht 1 ;
: grünesLicht 0 ;
: an 1 outPortB ;
: aus 0 outPortB ;
: rotphase    rotesLicht an warte
              rotesLicht aus ;
: grünphase   grünesLicht an warte
              grünesLicht aus ;
: gelbphase   gelbesLicht an warte
              gelbesLicht aus ;
: ampelzyklus rotphase grünphase
              gelbphase ;
: main initialisierePortB ampelzyklus
              ampelzyklus ;
```

Damit der Interpreter keine Fehler mehr meldet, müssen unsere neuen Wörter - über Zwischenstufen - auf solche Wörter zurückgeführt werden, die sich bereits im Vokabular befinden. In diesem Fall sind das die Zahlen 0, 1, 2, 3 und 7 (Auch diese können als Wörter angesehen werden!) sowie die Wörter `wait`, `DDRB` und `outPortB`.

Das Wort `wait` veranlasst den Attiny zu warten, `outPortB` gibt Werte am Port B aus und `DDRB` stellt das Datenrichtungsregister von PortB ein. Wie diese drei Wörter funktionieren, werden wir in den nächsten Kapiteln noch eingehend betrachten. Hier sollte nur eines deutlich werden:

Komplexe Wörter wie unser Wort `ampelzyklus` können wir Schritt für Schritt auf elementare Wörter zurückführen; diese Vorgehensweise nennt man auch **Top-Down-Programmierung.**

Wir hätten natürlich auch genau umgekehrt vorgehen können: Ausgehend von den elementaren Wörtern hätten wir immer komplexere Wörter definieren können, bis wir schließlich bei unserem Wort `ampelzyklus` ausgekommen wären. Diese Vorgehensweise bezeichnet man als **Bottom-Up-Programmierung**. In der Praxis arbeitet man häufig mit beiden Methoden gleichzeitig.

Wichtig ist allerdings für uns: Wörter, die zum Definieren eines neuen Wortes benutzt werden, müssen vorher bereits definiert worden sein. Das bedeutet: Sie müssen schon zum Grundvokabular von FORTH gehören oder in den vorangehenden Zeilen definiert und somit beim Interpretieren bereits zum Vokabular hinzugefügt worden sein. Die grundlegenden Worte müssen im FORTH-Quellcode also immer oben stehen, die daraus abgeleiteten weiter unten.

Unabhängig davon, ob wir die Top-Down-Methode oder die Bottom-Up-Methode benutzen - im Ergebnis ist das zu lösende Problem, eine Ampelanlage zu programmieren, schrittweise in viele kleine

Teilprobleme zerlegt worden. Solche Teilprobleme nennt man auch **Module** und die Zerlegung selbst wird als **Modularisierung** bezeichnet. Modularisierung ist ein wesentliches Merkmal der Programmiersprache FORTH. Gute FORTH-Programme zeichnen sich dadurch aus, dass die einzelnen Wort-Definitionen sinnvolle Einheiten bilden und nicht zu lang sind. Natürlich sollten auch die benutzten Wortnamen aussagekräftig sein.

Schauen wir daraufhin noch einmal den Quelltext an. Erfüllt er die Kriterien eines guten FORTHCodes? Sicherlich sind die ersten Zeilen - so kurz sie auch sein mögen - nicht unmittelbar einleuchtend; das hängt aber damit zusammen, dass wir die Wörter `wait`, `DDRB` und `outPortB` noch nicht genügend kennen. Wort-Folgen wie

```
grünesLicht an warte grünesLicht aus
```

lassen sich dagegen auch ohne Programmierkenntnisse leicht verstehen.

Das wichtigste Wort im ganzen Quelltext haben wir noch nicht besprochen; es ist das Wort `main`. Wenn der Attiny eingeschaltet wird, startet er immer mit der Ausführung genau dieses Wortes. Das Wort **`main`** hat demnach die Bedeutung eines Hauptprogramms; daher stammt auch die Wahl des Wortnamens (“main” = “haupt”). Alle Aktionen, welche der Mikrocontroller ausführen soll, müssen letztlich von diesem Wort ausgehen.

Somit muss der Quelltext immer mit der Definition von `main` enden, und beim Interpretieren muss man das Überschreiben eines bereits bestehenden `main`-Wortes stets zulassen; ansonsten arbeitet FORTH mit einem solchen alten “Hauptprogramm”. Und das hat womöglich gar nichts mit unserer Ampelsteuerung zu tun.

In unserem Fall sehen wir als letzte Zeile:

```
: main initPortB ampelzyklus ampelzyklus ;
```

Das bedeutet: Der Attiny soll zunächst das Port B initialisieren und danach zwei volle Ampelzyklen durchlaufen.

Bestimmt haben Sie inzwischen der Versuchung nicht mehr widerstehen können und den Quelltext unseres vollständigen Ampelprogramms interpretieren lassen. Wenn Sie ihn nicht abgeändert haben, müsste im Statusfeld jetzt angezeigt werden, dass das (im Vokabular schon) bestehende Wort `main` (wie gewünscht) überschrieben wurde.

Der für uns aufwändige Teil des Programmierens ist damit getan. Der Quelltext wurde erstellt und die neu definierten Wörter ins Vokabular übernommen. Jetzt muss unser FORTH-System ans Arbeiten: Die Wörter müssen in Attiny-Maschinencode umgesetzt werden. Diesen Schritt bezeichnet man als **Kompilieren**. Wie dieses Kompilieren im Einzelnen funktioniert, lässt sich bei FORTH recht gut nachvollziehen. In einem späteren Kapitel werden wir darauf ausführlich eingehen.

Jetzt aber machen wir es uns einfach: Wir drücken die Kompilieren-Schaltfläche und im Anschluss daran die Intel-HEX-Code-Schaltfläche. Den HEX-Code übertragen wir schließlich wie üblich mit dem Uploader-Programm auf den Attiny, auf dessen Platine wir in weiser Voraussicht eine rote LED bei PortB.2, eine gelbe bei PortB.1 und eine grüne bei PortB.0 eingesteckt haben.

Probieren Sie es selbst aus. Bestimmt werden auch Sie bei Ihrer Attiny-Platine die zwei Ampelzyklen beobachten können.

Aufgabe 1:

Ergänzen Sie die Datei ampel.frth so, dass eine “deutsche Ampel” mit einer zusätzlichen GelbGrün-Phase entsteht.

4 Arbeiten mit dem Stack

Der Stack ist einer der wichtigsten Konzepte von FORTH. Wir können uns den **Stack** vorstellen als einen Stapel von Zahlen. In der Tat heißt das englische Wort "stack" auf deutsch nichts anderes als **Stapel**. Wozu dient nun der Stack und wie wird er praktisch eingesetzt? Das soll in diesem Kapitel erklärt werden.

Schauen wir uns zunächst das Wort `stapeln` an:

```
: stapeln 11 22 33 44 55 ;
```

Wird dieses Wort ausgeführt, so werden die Zahlen `11`, `22`, `33`, `44` und `55` der Reihe nach auf den Stack gelegt. Anschaulich können wir uns das so vorstellen:

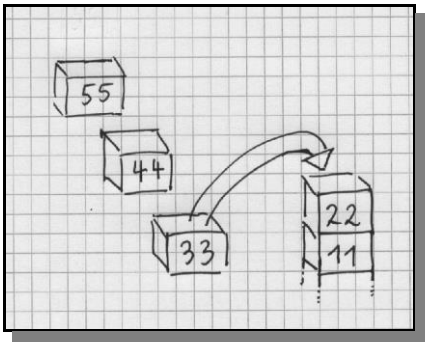


Abb. 1 (6)

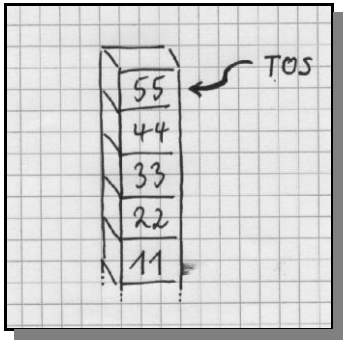


Abb. 2 (7)

Am Ende liegen unsere fünf Zahlen übereinander, die 11 zuunterst, die 55 ganz oben. Die oberste Zahl wird auch **TOS** (= Top Of Stack) genannt.

Wörter wie z. B. "." und `wait` greifen auf diesen Stapel zu. Das Wort "." holt sich z. B. den TOS vom Stapel und gibt diese Zahl auf dem Port

B aus; das Wort `wait` greift sich auch den TOS und wartet entsprechend viele Sekunden. Wird das folgende Wort

```
: ausgabe . wait . ;
```

nach dem Wort `stapeln` ausgeführt, geschieht folgendes: "." holt die Zahl 55 vom Stack und gibt sie auf Port B aus.

`wait` holt die Zahl 44 vom Stack und wartet entsprechend viele Sekunden.

"." holt die Zahl 33 vom Stack und gibt sie auf Port B aus.

Am Schluss befinden sich nur noch die Zahlen 11 und 22 auf dem Stack.

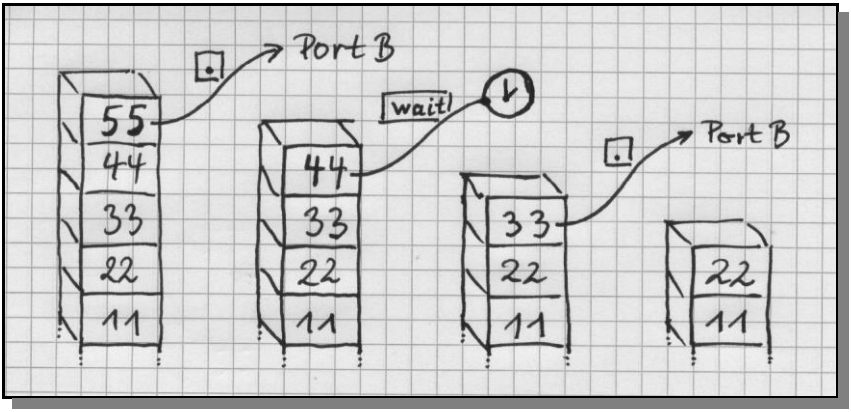


Abb. 3 (8)

Wörter verändern i. A. den Stack: Unser Wort `stapeln` legt 5 Zahlen auf den Stack, unser Wort `ausgabe` entfernt die obersten drei Zahlen. Manche Worte holen erst Zahlen vom Stapel und legen anschließend neue Werte auf dem Stapel ab. Dies gilt insbesondere auch für Rechenoperationen wie Plus und Minus.

Wir schauen uns einmal etwas genauer an, wie man mithilfe des Wortes "+" in FORTH zwei Zahlen addiert. Zum Beispiel sollen die Zahlen 4 und 9 addiert werden. Von den meisten Taschenrechnern, aber auch von

vielen Programmiersprachen, ist man es gewohnt, die folgende Anweisung zu schreiben:

4 + 9

Das Rechenzeichen steht zwischen den beiden Summanden; man spricht hier von einer **Infix**Schreibweise.

In FORTH schreibt man dies so:

```
4 9 +
```

(Leerzeichen zwischen 4 und 9 nicht vergessen!)

Hier werden zuerst die beiden Summanden eingegeben und anschließend erst das Rechenzeichen; man spricht hier von einer **Postfix**-Schreibweise.

Was steckt dahinter? Natürlich unser Stapel! Zunächst werden die Zahlen 4 und 9 auf den Stapel gelegt; dann holt das Wort "+" diese beiden Zahlen vom Stapel, addiert sie und legt das Ergebnis (also 13) wieder auf den Stapel.

Der Vorteil dabei: Das Wort "+" führt bei FORTH die Addition sofort aus; beide Summanden liegen ja bereits vor. Bei der Infix-Schreibweise ist das nicht so einfach möglich. Hier müssen sich Taschenrechner oder Computer das Pluszeichen zunächst merken; die eigentliche Addition kann erst ausgeführt werden, wenn nach dem zweiten Summanden noch ein weiterer Befehl z. B. in Form von "=" erfolgt.

Wenden wir unsere Kenntnisse nun an, um den Attiny mit FORTH die Rechenaufgabe 4 + 9 durchführen zu lassen. Unser Programm sieht so aus:

```
: main 4 9 + . ;
```

Wir geben es in das Quelltext-Feld ein, interpretieren, kompilieren und übertragen es. Die Leuchtdioden am Port B zeigen tatsächlich die Zahl 13 an (&B00001101).

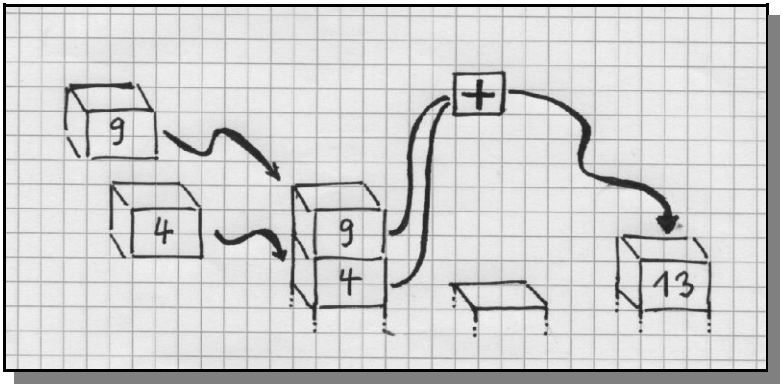


Abb. 4 (9)

Wir haben bereits gelernt, was hier im Einzelnen geschieht: Zuerst werden die Zahlen 4 und 9 auf den Stack gelegt; dann holt das Wort "+" diese beiden Zahlen vom Stack, addiert sie und legt das Ergebnis wieder auf den Stack. Das nächste Wort "." holt sich diese Zahl 13 vom Stack und gibt sie auf dem Port B aus.

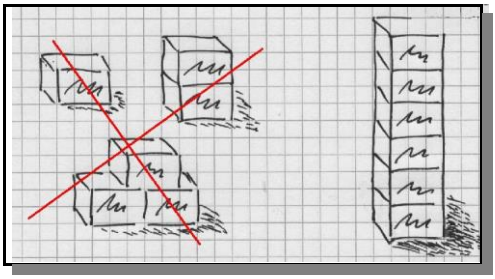


Abb. 5 (10)

Wir sehen: Der Stack ist eine Art Marktplatz, auf dem die einzelnen Wörter Zahlen holen oder auch abgeben können. Im Gegensatz zu einem echten Marktplatz können hier allerdings nur Zahlen gehandelt werden; außerdem gibt es hier nur einen einzigen Stand und an diesem Stand liegen die Zahlen nicht irgendwie nebeneinander, sondern ordentlich übereinander auf einem einzigen Stapel.

An dieser Stelle sei schon verraten: FORTH stellt noch weitere Möglichkeiten zum Austausch von Daten zwischen den Wörtern zur Verfügung, z. B. sogenannte Variablen. Der Austausch über den Stack ist aber die wichtigste Methode. Deswegen wollen wir den Umgang mit dem Stack noch etwas üben.

Wie man einfache Rechenaufgaben mit FORTH lösen kann, haben wir schon kennen gelernt. Wie aber sieht es mit komplexen Termen aus?

1. Beispiel:

Term: (3 + 5) * 2

FORTH: `3 5 + 2 *`

2. Beispiel:

Term: 120 - 5 * 20

FORTH: `120 5 20 * -`

Bei dem letzten Beispiel könnten die Zahlen 120, 5 und 20 schon fertig auf dem Stack liegen; um das Ergebnis des Terms zu erhalten, müssten nur noch die Worte "*" und "-" hintereinander ausgeführt werden. Ginge das auch beim ersten Beispiel? Nein, auf keinen Fall wäre das so einfach wie im Beispiel 2: Da die Zahl 2 auf dem TOS liegt, würde jede Operation sich auf jeden Fall (auch) auf diese Zahl 2 beziehen. Die

Klammern im Term verlangen aber, dass zunächst die Zahlen 3 und 5 verarbeitet (addiert) werden.
Es gibt aber eine Reihe von FORTH-Wörtern, die die Zahlen auf dem Stack manipulieren (vertauschen, entfernen oder verdoppeln) können:

5 Port-Befehle

MikroForth stellt folgende Port-Befehle zur Verfügung:

Wort	Typ	Kommentar	Stack
.	A	gibt TOS auf Port B aus; (Datenrichtungsbits von Port B werden zuvor alle auf 1 gesetzt.)	(n –)
`blink`	F	`b hp blink` gibt das Bitmuster von b auf Port B aus, wartet hp Millisekunden, gibt 0 auf Port B aus und wartet wieder hp Millisekunden.	(b hp –)
`DDBitB`	A	`bit flag DDBitB` setzt den Anschluss bit des Ports B als Ausgang, wenn flag = 1, sonst als Eingang.	(bit flag –)
`DDBitD`	A	`bit flag DDBitD` setzt den Anschluss bit des Ports D als Ausgang, wenn flag = 1, sonst als Eingang.	(bit flag –)
`DDRB`	A	schreibt b in das Datenrichtungsregister des Ports B.	(b –)
`DDRD`	A	schreibt d in das Datenrichtungsregister des Ports D.	(d –)

`InPortB`	A	`bit InPortB` liest den Eingang bit des Ports B und legt 1/0 auf den Stack, wenn er high/low ist. Vgl. `DDRB` und `DDBitB`	(bit – flag)
`InPortD`	A	`bit InPortD` liest den Eingang bit des Ports D und legt 1/0 auf den Stack, wenn er high/low ist. Vgl. `DDRD` und `DDBitD`	(bit – flag)
Wort	**Typ**	**Kommentar**	**Stack**
`outPortB`	A	`bit flag outPortB` setzt den Ausgang bit des Ports B auf high/low, wenn flag = 1/0 ist. Vgl. `DDRB` und `DDBitB`	(bit flag –)
`outPortD`	A	`bit flag outPortD` setzt den Ausgang bit des Ports D auf high/low, wenn flag = 1/0 ist. Vgl. `DDRD` und `DDBitD`	(bit flag –)
`Ta0?`	F	Legt 1/0 auf Stack, wenn Taster Ta0 offen/geschlossen (D2=1/0) PortD.2 wird automatisch konfiguriert.	(– bit)
`Ta1?`	F	Legt 1/0 auf Stack, wenn Taster Ta1 offen/geschlossen (D3=1/0) PortD.3 wird automatisch konfiguriert.	

Exemplarisch werden wir hier die Wörter ".", `blink`, `Ta0?`, `DDBitD`, `OutPortD` und `InPortD` behandeln. Die restlichen Wörter sind in ihrer Bedeutung ganz ähnlich.

Um etwas interessantere Beispiele verwenden zu können, wollen wir allerdings zuvor eine einfache FORTH-Schleifenkonstruktion vorstellen: die BEGIN-UNTIL-Schleife. Diese sieht folgendermaßen aus:

```
begin   Bef1 Bef2 Bef3 ... until
```

Die Befehle Bef1, Bef2, Bef3, ... werden der Reihe nach immer wieder durchlaufen. Allerdings geschieht dies nur solange, wie das Wort `until` auf demTOS eine 0 vorfindet.

Genauer gesagt: Das Wort `until` holt den Wert aus dem TOS und kontrolliert nach, ob er 0 oder 1 ist. Ist er 0 (FALSE), wird die Schleife ein weiteres Mal ausgeführt; ist er 1 (TRUE), so wird die Schleife beendet. Schreibt man also unmittelbar vor das Wort `until` eine 0, so wird eine Endlosschleife gebildet:

```
: endlos Bef1 Bef2 Bef3 ... 0 until ;
```

Kommen wir zu unserem ersten Beispiel: Alle LEDs an Port B sollen im Abstand von 100 ms immer wieder an- und ausgehen. Das Programm dafür ist recht einfach:

```
: main begin 255 . 100 waitms 0 . 100
       waitms 0 until ;
```

Schauen wir uns die Definition von `main` Wort für Wort an. `begin` leitet die Endlosschleife ein, welche durch `0 until` begrenzt wird. Innerhalb der Schleife wird zuerst 255 auf den Stack gelegt. Diese Zahl wird durch das Wort "." sogleich vom Stapel genommen und am Port B

ausgegeben. Dabei wird durch "." Port B automatisch als Ausgang konfiguriert; d. h. DDRB wird auf &B11111111 gesetzt.

Nun sind also alle LEDs an Port B eingeschaltet. Danach wird die Zahl 100 auf den Stack gelegt, um sogleich von dem Wort "waitms" geholt zu werden: Der Mikrocontroller wartet jetzt 100 ms. Anschließend wird die Zahl 0 auf dem Port B ausgegeben; die LEDs werden somit alle ausgeschaltet. Dann wartet der Mikrocontroller abermals 100 ms. Wir sind am Ende eines Schleifendurchlaufs angekommen. Nun beginnt das Ganze wieder von vorne und so weiter und so weiter... Unsere LEDs an Port B blinken also fortwährend.

In unserem zweiten Beispiel soll eine LED an PortD.6 über den Taster Ta0 aus- und eingeschaltet werden. Genauer gesagt soll die LED aus sein, solange der Taster Ta0 gedrückt ist, und leuchten, solange der Taster nicht betätigt ist. Das FORTH-Programm kann durch folgende Zeilen gebildet werden:

```
: schalten begin Ta0? 6 swap
           outPortD 0 until ;
: vorbereiten 6 1 DDBitD ;
: main vorbereiten schalten ;
```

Zunächst wird durch das Wort `vorbereiten` das Bit 6 des Datenrichtungsregisters von D auf 1 gesetzt; Port D.6 wird also als Ausgang konfiguriert. Das Wort `schalten` besteht aus einer Endlosschleife; zu Beginn der Schleife kontrolliert das Wort `Ta0?`, ob der Taster Ta0 gedrückt ist oder nicht. Ist Ta0 gedrückt, legt es den Wert 0 auf den Stack, sonst den Wert 1. Ähnlich wie schon bei dem Wort "." wird der zugehörige Eingang Port D.2 von dem Wort `Ta0?` automatisch konfiguriert (Eingang und Pull-up).

Anschließend wird die Zahl 6 auf den Stack gelegt; `swap` tauscht diesen Wert 6 mit dem von `Ta0?` gelieferten Ein-Aus-Wert aus. Nun liegen der Bit-Wert 6 und der Ein-Aus-Wert genau in der Reihenfolge auf dem

Stapel, wie sie von `OutPortD` benötigt werden: unten der Bit-Wert und oben der An-Aus-Wert (im Vokabular als Flag bezeichnet). `6 1 OutPortD` schaltet z. B. die LED an PortD.6 an; `6 0 OutPortD` schaltet sie aus.

> ***Beachten Sie***: Nur bei den Wörtern ".", `Ta0?`, `Ta1?` und `blink` erfolgt eine automatische Konfigurierung der Ports; bei allen anderen Port-Befehlen müssen die Datenrichtungsbytes bzw. -Bits vom Anwender selbst mithilfe der Wörter `DDRB`, `DDRD`, `DDBitB` und `DDBitD` eingestellt werden.

Im dritten Beispiel soll eine Blink-Schleife über den Taster Ta0 abgebrochen werden. Genauer gesagt soll das Bitmuster 01010101 solange ein- und ausgeschaltet werden, bis der Taster Ta0 gedrückt wird. Das zugehörige Programm ist auch wieder sehr kurz und sieht so aus:

```
: main begin 85 100 blink Ta0? not until ;
```

Innerhalb der BEGIN-UNTIL-Schleife wird zunächst das `blink`-Wort mit dem Bitmuster &B01010101 = 85 und der halben Periodendauer 100 ms ausgeführt. Anschließend wird mit `Ta0?` der Zustand des Tasters Ta0 abgefragt; ist dieser Taster gedrückt, wird eine 0 auf den Stapel gelegt, sonst eine 1.

Ohne das folgende Wort `not` würde dieser Ein-Aus-Wert des Tasters direkt von `until` ausgewertet: der Zustandswert 0 (Taster gedrückt) würde zu einem weiteren Schleifendurchlauf führen und der Zustandswert 1 (Taster offen) würde die Schleife abbrechen. Die Schleife würde also durch ein Öffnen und nicht - wie gefordert - durch ein Schließen des Tasters beendet.

Um zu einem korrekt funktionierenden Programm zu gelangen, muss also der Zustandswert umgekehrt werden: Aus dem Wert 1 muss eine 0

und aus dem Zustandswert 0 muss eine 1 gemacht werden. Dies kann man mit dem Wort `not` erreichen: Dieses Wort holt den Zustandswert vom Stapel und ersetzt ihn durch sein logisches Komplement. Die Wortfolge `Ta0? not` liefert jetzt wie gewünscht auf dem TOS den Wert 0, wenn der Taster offen ist, und den Wert 1, wenn der Taster geschlossen ist.

Aufgabe 1

Ein Blick in den FORTH-Editor zeigt, wie das Wort `Ta0?` definiert ist.

```
: Ta0? 2 0 DDBitD 2 1 outPortD 2 InPortD ;
```

Erläutern Sie diese Definition.

Aufgabe 2

Wie ließe sich das Programm zum ersten Beispiel mithilfe des `blink`-Wortes vereinfachen?

Aufgabe 3

Ändern Sie das Programm des zweiten Beispiels so ab, dass die LED leuchtet, wenn der Taster gedrückt ist, und sonst nicht.

Aufgabe 4

Der Attiny soll die Anzahl der Tastendrücke beim Taster Ta0 an PortB anzeigen; PortB muss entsprechend mit 8 LED bestückt werden. Beachten Sie: Der Tastendruck dauert immer eine gewisse Zeit; selbst bei flotten Menschen bleibt der Taster für mehrere Millisekunden geschlossen, für den Attiny ist das aber eine halbe Ewigkeit!

6 Schleifen und Verzweigungen

Einen ersten Schleifentyp haben Sie im Kapitel über die Portbefehle schon kennen gelernt, die BEGIN-UNTIL-Schleife. Sie hat folgende Form:

```
begin Bef1 Bef2 Bef3 ... until
```

Durch diese Konstruktion wird die Befehlsfolge zwischen `begin` und `until` solange ausgeführt, bis das Wort `until` auf dem TOS eine 1 vorfindet. 1 wird allgemein als Wahrheitswert TRUE interpretiert, 0 als FALSE.

Die Schleife wird also solange ausgeführt, bis auf dem Stack der Wahrheitswert TRUE vorliegt. Zu beachten ist, dass `until` den Wert auch aus dem TOS holt; es muss also bei jedem Schleifendurchlauf dafür gesorgt werden, dass für das Wort `until` ein passender Wahrheitswert auf den Stack gelegt wird.

Da im Kapitel über die Portbefehle schon einige praktische Beispiele für die BEGIN-UNTILSchleife vorgestellt worden sind, wollen wir uns gleich dem nächsten Schleifentyp zuwenden, der Zählschleife. In FORTH sieht sie folgendermaßen aus:

```
ew sw do Bef1 Bef2 Bef3 ... loop
```

Die Bezeichner `ew` (Endwert) und `sw` (Startwert) stehen hier für den Wert des Schleifenindex beim letzten bzw. ersten Schleifendurchlauf. Innerhalb der Zählschleife, also zwischen den Wörtern `do` und `loop`, kann man auf den Schleifenindex mithilfe des Wortes `I` zurückgreifen: `I` legt den aktuellen Schleifenindex auf den Stapel. Schauen wir uns ein einfaches Beispiel dazu an:

```
: zählen 25 10 do I . 100 waitms loop ;
```

Bei diesem Wort startet die Zählschleife mit dem Index 10. Diese Zahl wird zunächst durch das Wort `I` auf den Stapel gelegt und mit dem Wort . am Port B ausgegeben. Nach 100 Millisekunden Wartezeit wird der Schleifenindex automatisch erhöht und die Schleife ein weiters mal durchlaufen. Die Schleife wird ein letztes Mal durchlaufen, wenn der Schleifenindex den Wert 25 hat. Unser Programmzählt also im Zehntelsekunden-Rhythmus von 10 bis 25 und hört dann auf.

Man beachte bei der Angabe der Werte für den Schleifenindex die Reihenfolge: Zuerst wird der Endwert und dann der Startwert angegeben.

Als weiteres Beispiel schauen wir uns die FORTH-Definition der Multiplikation an:

```
: * 0 swap 1 do swap dup rot + loop ;
```

Lautet das Hauptprogramm z. B.

```
: main 12 7 * . ;
```

so wird die Zahl 12 insgesamt 7 mal zur 0 addiert; die Multiplikation wird also auf eine Mehrfachaddition zurückgeführt. Wie das im Detail abläuft, sollte der Leser einmal selbst überlegen, indem er für jeden einzelnen Schritt den Inhalt des Stacks notiert.

Unser MikroForth besitzt nur einen einzigen Verzweigungstyp, die `skipIf`-Anweisung. Dieses Wort wertet zunächst den TOS aus; liegt auf dem TOS der Wert 1 (TRUE), wird die nächste Anweisung übersprungen. Liegt auf dem TOS der Wert 0 (FALSE), wird einfach mit dem nächsten Befehl (Wort) weitergearbeitet.

```
1 skipIf Bef1 Bef2 Bef3 ...
```

Hier wird nach dem Wort `skipIf` das Wort `Bef1` übersprungen und sofort mit demWort `Bef2` weitergearbeitet.

Es folgt das Wort `Bef3` usw.

```
0 skipIf Bef1 Bef2 Bef3 ...
```

Hier wird nach dem Wort `skipIf` mit dem Wort `Bef1` weitergearbeitet. Es folgen die Worte `Bef2` und `Bef3` usw.

Häufig ergeben sich dabei die Wahrheitswerte 0 und 1 als Ergebnisse von Vergleichen. Hier kommen Vergleichsoperatoren zum Einsatz. Ähnlich wie die Rechenoperatoren +, *, - und / werden sie bei FORTH auch in der Postfix-Schreibweise benutzt. Durch

```
7 2 >
```

wird also überprüft, ob 7 > 2 gilt. Da das in diesem Fall wahr ist, wird als Ergebnis dieser Vergleichsoperation der Wert 1 (TRUE) auf den Stapel gelegt. Weitere Vergleichsoperatoren sind < und =.

Ein Beispiel soll erläutern, wie Vergleichsoperatoren und Verzweigungen sinnvoll eingesetzt werden können. Ein Messprozess möge bereits zwei Messwerte (z. B. Temperaturwerte) auf den Stapel gelegt haben. Das Wort `unterschied` soll - wie der Name schon sagt - den Unterschied der beiden Zahlen bestimmen; es könnte z. B. erforderlich sein, vom Mikrocontroller bestimmte Maßnahmen einleiten zu lassen, wenn dieser Unterschied zu groß ist.

Kümmern wir uns zunächst um Berechnung und Ausgabe des Unterschieds. Auf den ersten Blick scheint dieses Problem recht einfach zu lösen zu sein:

```
: unterschied - . ;
```

Zu Testzwecken geben wir bei unserem Forth-Compiler ein:

```
: main 7 2 unterschied ;
```

Nach dem Interpretieren, Compilieren und Übertragen zeigt unser Mikrocontroller den Wert 5 an - wie erwartet! Nun geben wir die Messwerte aber einmal in umgekehrter Reihenfolge ein:

```
: main 2 7 unterschied ;
```

Nun zeigt der Mikrocontroller amPort B den Wert 251(!) an. Wie lässt sich dieses offensichtlich unsinnige Ergebnis erklären, und - mindestens genau so wichtig - wie lässt sich unser Programm verbessern?

Zunächst zur Erklärung: Bei der Subtraktion 2 - 7 gelangt der Mikrocontroller in den Bereich unter 0. Er arbeitet dabei wie ein Kilometerzähler: Wenn man ausgehend vom Kilometerstand 0002 nun 7 km rückwärts fährt, kommt man beim Stand von 9995 aus. Der Kilometerstand springt nämlich beim Rückwärtszählen von 0000 auf 9999. Ganz ähnlich arbeitet der Mikrocontroller: Er springt beim Rückwärtszählen von 000 auf 255.

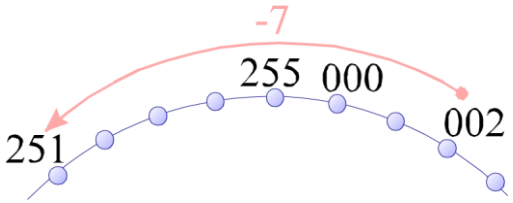


Abbildung 1 (11)

Das Problem liegt also offensichtlich in der Reihenfolge der beiden Messwerte. Um das Programm zu verbessern, müssen wir dafür sorgen, dass die Messwerte ausgetauscht werden, wenn der erste Messwert kleiner als der zweite ist. Hier kann unser `skipIf`-Wort zum Einsatz kommen; für den zugehörigen Vergleich müssen die Messwerte allerdings vorher noch kopiert werden.

```
: unterschied over over > skipIf swap - . ;
```

Das folgende Stapelbild macht deutlich, was beider Ausführung von `unterschied` geschieht.

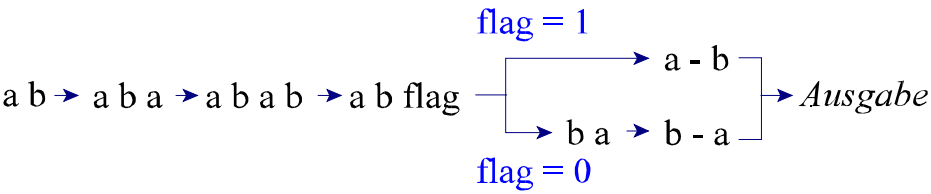


Abbildung 2 (12)

Aufgabe 1

Wenn der Unterschied der beiden Messwerte auf dem Stapel größer als 3 ist, dann soll ein Warnton über D.6 und den Beeper ausgegeben werden.

Aufgabe 2

Schreiben Sie eine Definition für das FORTH-Wort “<=”.

Aufgabe 3

Wie lautet die FORTH-Definition für das Wort `not ?`

7 Alles unter Kontrolle: COM, I²C und EEPROM

Mikrocontroller werden häufig als Herzstück autonomer Messstationen eingesetzt. Folgende Voraussetzungen sollten sie dazu erfüllen:

1. Sie müssen einen Speicher besitzen, der die Messdaten sicher verwahren kann - möglichst auch dann noch, wenn die elektrische Versorgung des Mikrocontrollers ausfällt.

2. Sie müssen gängige Kommunikationsschnittstellen zu Sensoren besitzen.

3. Sie müssen gängige Kommunikationsschnittstellen zu Terminals besitzen, damit die Daten problemlos zur weiteren Auswertung auf Computer übertragen werden können.

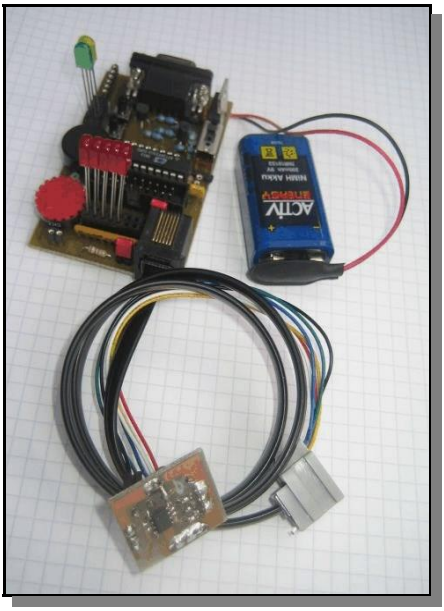


Abbildung 1 (13)

All dies kann unser Attiny2313 mit MikroForth leisten: Er besitzt ein EEPROM, welches Daten auch ohne elektrische Quelle dauerhaft speichern kann. Über den I²C-Bus kann er mit Sensoren und anderen Geräten kommunizieren und über die COM-Schnittstelle kann er die gespeicherten Daten an ein Terminal senden. Für genauere Erläuterungen zu EEPROM, I²C und COM-Schnittstelle sei auf die entsprechenden Abschnitte verwiesen. Hier soll anhand eines einfachen Beispiels betrachtet werden, wie MikroForth zum Anfertigen eines Temperaturmessprotokolls eingesetzt werden kann.

Der Aufbau ist einfach: An die I²C-Buchse der Attiny-Platine wird ein Temperatursensor LM75 mit der I²C-Adresse 157 angeschlossen (Abb. 1). Nicht vergessen sollten Sie, die Jumper zu setzen, die zum Pull-Up der beiden Leitungen SDA und SCL erforderlich sind. Der Datenaustausch mit dem PC erfolgt über dasselbe Kabel wie die Programmierung.
Folgende FORTH-Wörter stellt MikroForth für EEPROM, I²C und COM zur Verfügung:

Wort	**Typ**	**Kommentar**	**Stack**
`>com`	A	sendet TOS an die COM-Schnittstelle. Vorher muss die COM-Schnittstelle mit `INITCOM` initialisiert worden sein.	(n –)
`>eprom`	A	schreibt den Wert w in die Adresse a des EEPROMs. Vgl. `eprom>`	(w a –)
Wort	**Typ**	**Kommentar**	**Stack**

com>	A	legt über COM-Schnittstelle empfangenes Byte auf den Stack. Vgl. >com.	(– n)
eprom>	A	liest den Wert w aus der EEPROMAdresse a und legt ihn auf den Stack Vgl. >eprom	(a – w)
i2cread	A	Ein Wert wird vom Slave gelesen; wenn ACK = 0 ist, wird ein Acknowledge-Signal gegeben.	(ACK – Wert)
i2cstart	A	Startsignal für I²C-Bus wird gesendet (SDA von 1 auf 0; dann SCL von 1 auf 0)	(–)
i2cstop	A	initialisiert den I²C-Bus (SCL und SDA auf 1); Datenrichtungsbits für SDA (PortB.5) und SCL (PortB.7) werden gesetzt.	(–)
i2cwrite	A	Ein einzelnes Byte (Wert oder Adresse) wird an den Slave gesendet; das Acknowledge-Bit wird auf den Stack gelegt.	(Wert/Adr – ACK)

`initCom`	A	initialisiert die COM-Schnittstelle: D0 = RxD D1 = TxD Baudate = 9600 8 Bit kein Paritätsbit	(–)

Zunächst schauen wir uns den Messprozess an: Der Attiny soll im Sekundenabstand 20 Temperaturwerte im EEPROM aufzeichnen. Dies leisten die beiden Wörter `messung` und `main`:

```
: messung i2cstop i2cstart 1 wait
          157 i2cwrite 1 i2cread ;
: main 20 1 do messung I >eprom loop ;
```

Schauen wir uns zunächst die Definition von `messung` an: Mit `i2cstop` wird die I²C-Schnittstelle initialisiert (SCL und SDA auf high), mit `i2cstart` wird das Startsignal gegeben (SDA wechselt von high auf low), mit `157 i2cwrite` wird unser Temperatursensor adressiert.

Schließlich wird mit `1 I2cread` ein Temperaturwert vom LM75 abgefragt und auf den Stack gelegt; der Parameter 1 sorgt dafür, dass kein Acknowledge-Signal gegeben wird. Ein Acknowledge würde nämlich den LM75 dazu veranlassen, als nächstes die Nachkommastelle zu senden. Man beachte ferner, dass der LM75 bis zu 300 ms für eine einzige Temperaturmessung benötigt. Eine längere Pause (hier 1 Sekunde) zwischen den einzelnen Messungen ist also unabdingbar!

Das Wort `main` besteht im Wesentlichen aus einer Zählschleife, bei der die einzelnen Messwerte imEEPROM abgelegt werden. Der Schleifenindex `I` gibt jeweils die Adressnummer des EEPROMs an.

Nun müssen wir die gemessenen Werte noch an den PC übertragen. Dazu setzen wir das Wort `eprom2com` ein:

```
: eprom2com eprom> >com 1 waitms ;
: main initCom 20 1 do I eprom2com loop ;
```

Bevor die Daten Byte für Byte über die COM-Schnittstelle übertragen werden können, muss diese initialisiert werden. Dies geschieht mit dem Wort `initCom`. BeimSenden der Bytes muss das Timing beachtet werden: Das Übertragen eines Bytes über die serielle Schnittstelle dauert eine gewisse Zeit. Während des Sendevorgangs arbeitet der Mikrocontroller aber schon sein Programmweiter ab. Würde man auf den Befehl `1 waitms` verzichten, würde der Mikrocontroller die nächste Übertragung starten wollen, bevor das letzte Byte vollständig übertragen wäre.

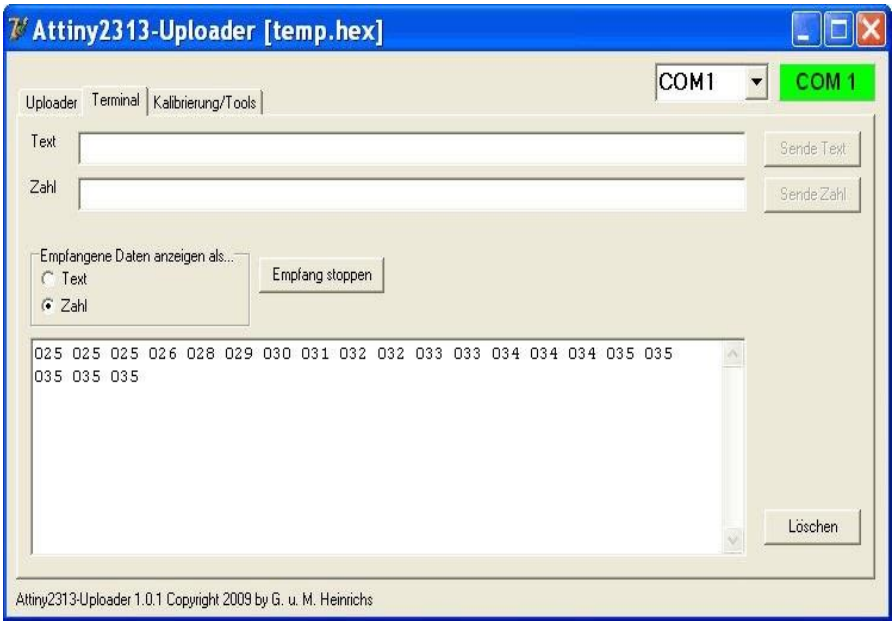


Abbildung 2 (14)

Wenn wir nun das Terminalprogramm von UPLOADER laufen lassen, werden sämtliche Messwerte in weniger als 1 Sekunde an den Computer übertragen. Abb. 2 zeigt das Ergebnis einer solchen Übertragung. Deutlich erkennt man, dass die Temperaturwerte zunächst konstant bleiben und dann rasch ansteigen. Wie ist das zustande gekommen?

Während des Messvorgangs wurde eine Lampe direkt über den LM75 gehalten; dadurch wurde dieser stark erwärmt. In Abb. 3 sind die Ergebnisse mit einem Tabellenkalkulationsprogramm graphisch dargestellt worden.

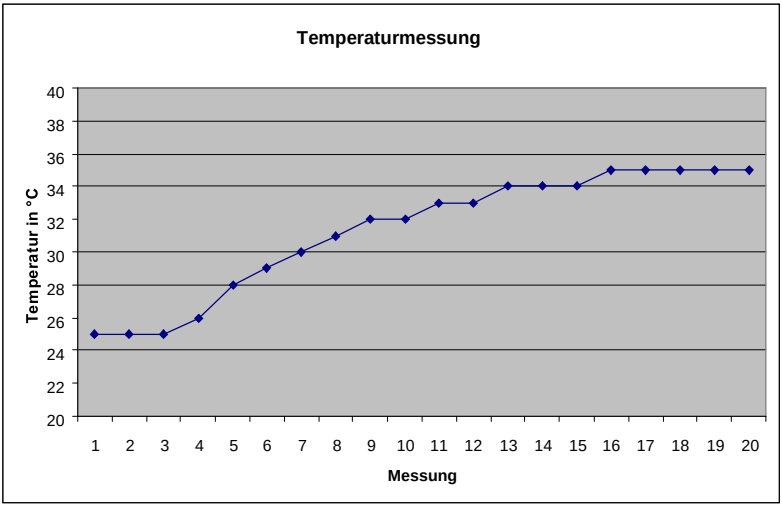


Abbildung 3 (15)

8 MikroForth-Variablen

Zum kurzfristigen Speichern und zur Übergabe von Werten von einem Wort zum anderen ist der Stack bestens geeignet. Zum langfristigen Speichern bietet sich eher das EEPROM des Attiny an.

Allerdings ist es recht mühselig, sich die unterschiedlichen Adressen für die zu speichernden Werte zu merken. Abhilfe schaffen hier **Variable**. Sie werden in MikroForth folgendermaßen deklariert:

```
variable <Variablenname>
```

Die Variablendeklaration kann nicht innerhalb einer Doppelpunktdefinition stehen; das Schlüsselwort `variable` muss am Anfang einer Zeile stehen. Pro Zeile kann jeweils immer nur eine einzige Variable deklariert werden und am Ende der Zeile braucht kein Semikolon stehen.

Der Compiler vergibt für jede Variable eine Nummer zwischen 0 und 126, die als EEPROM-Adresse dient; die EEPROM-Zelle mit der Adresse 127 ist für den OSCCAL-Wert reserviert. Zudem erzeugt der Compiler für jede Variable ein Wort mit gleichem Namen. Dieses Wort hat nur eine einzige Aufgabe: Es legt die zugehörige Adresse auf den Stack.

An einem Beispiel wollen wir uns die Benutzung von Variablen anschauen:

```
variable KontoNr
: ablegen 129 KontoNr >eprom ;
: holen KontoNr eprom> ;
: main ablegen holen . ;
```

In der ersten Zeile wird die Variable `KontoNr` deklariert. In der zweiten Zeile wird durch das Wort `KontoNr` die zugehörige Adresse auf den Stack gelegt. Wenn `KontoNr` als erste Variable deklariert worden ist,

dann ist diese Adresse $00. Durch das nachfolgende Wort `>eprom` wird die Zahl 129 unter der EEPROM-Adresse $00 gespeichert. In der dritten Zeile wird die Zahl 129 wieder aus der EEPROM-Zelle $00 geholt und auf den Stack gelegt. Das Wort `KontoNr` kann hier wieder als Stellvertreter für die zugehörige EEPROM-Adresse angesehen werden.

Durch die Einführung von Variablen ändert sich der Quellcode im Wesentlichen nicht; er wird aber vielübersichtlicher. Man beachte: Da bei einer EEPROM-Zelle nur eine beschränkte Anzahl von Schreibvorgängen durchgeführt werden kann, sollte man das Wort `holen` sparsam einsetzen. Von daher sollten die MikroForth-Variablen eher als Konstanten-Speicher angesehen werden.

Im Prinzip könnten die Variablen auch zur Adressierung von SRAM- oder auch Flash-Registern benutzt werden. Davon ist aber dringend abzuraten. Der Compiler vergibt nämlich die Adressen der Reihe nach, bei 0 beginnend. So ist es fast sicher, dass unkontrolliert wichtige Statusregister oder auch Teile des Programms auf diese Weise überschrieben würden. Die Folgen wären fatal.

Aufgabe 1

Geben Sie das obige Beispiel ein und testen Sie es. Schauen Sie sich mit dem Vokabular-Editor auch das Wort `KontoNr` an.

9 Der Compiler von MikroForth

FORTH ist von der Struktur her eine einfache Sprache; deswegen ist es auch nicht schwer, die Funktionsweise unseres Forth-Compilers nachzuvollziehen. Ausgangspunkt unserer Betrachtungen soll ein kleines FORTH-Programm sein, das wir schon im Kapitel über die Portbefehle kennen gelernt haben, die Datei “schalten.frth”:

```
: schalten begin Ta0? 6 swap outPortD 0 until ;
: vorbereiten 6 1 DDBitD ;
: main vorbereiten schalten ;
```

Durch dieses Programm wird eine Leuchtdiode an Port D.6 durch den Taster Ta0 ein- und ausgeschaltet.

Wir öffnen diese Datei und betätigen die Interpretieren-Schaltfläche. Dadurch werden die neuen Wörter der Doppelpunktdefinitionen in das Vokabular eingetragen. Dies können wir leicht überprüfen, indem wir oben links auf die Lasche “Vokabular editieren” klicken.

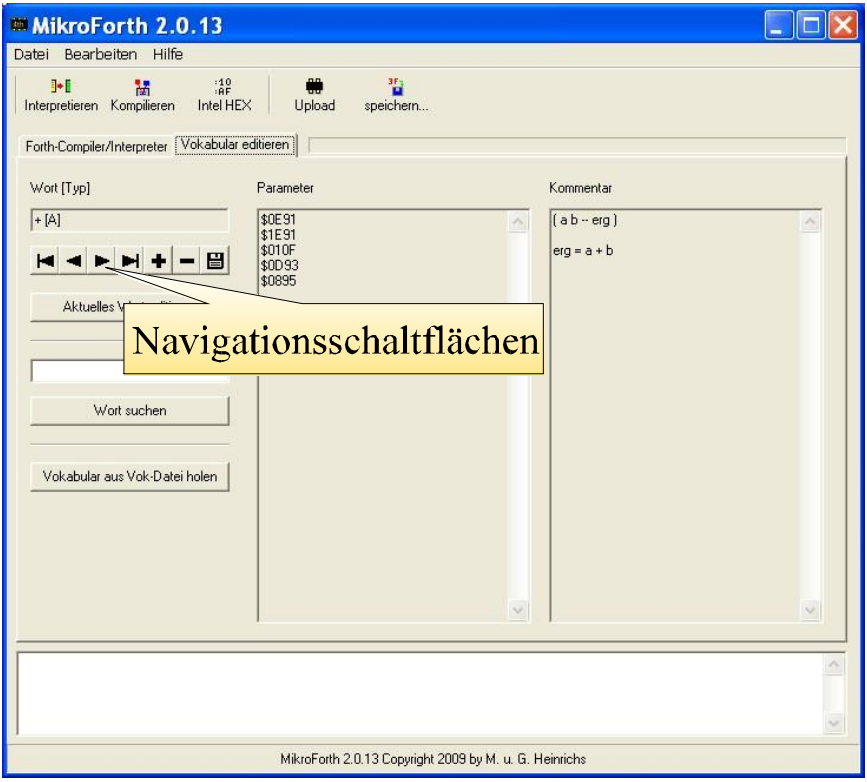


Abb. 1 (16)

In diesem Editor können wir alle Wörter des Vokabulars anschauen; darüber hinausgehend können wir hier bestehende Wörter auch abändern oder sogar auch neue Wörter erzeugen, aber darauf wollen wir erst ineinem späteren Kapitel zu sprechen kommen. Betätigen wir nun die ▶| Schaltfläche, gelangen wir zum letzten Wort des Vokabulars; hier finden wir unser Wort "vorbereiten". Im Parameterfeld entdecken wir die Worte, durch die `vorbereiten` im Rahmen der Doppelpunkt-definition beschrieben worden ist. Beim **Interpretieren** wird also im Wesentlichen nur umstrukturiert: aus einer Textzeile werden das definierte Wort und die zugehörigen Parameter herausgeschält.

Auch das neue Wort `schalten` können wir uns anschauen, dazu müssen wir nur die Schaltfläche ◄ betätigen; dadurch gelangen wir zu dem vorletzten Wort des Vokabulars. Indem wir diese Schaltfläche immer wieder betätigen, können wir uns alle Wörter des Vokabulars anschauen. Dabei fällt auf, dass es zwei Typen von Wörtern gibt:

1. **F-Wörter**: Ihre Parameter bestehen selbst wieder aus Wörtern. Sie sind aus Doppelpunktdefinitionen hervorgegangen.

2. **A-Wörter**: Ihre Parameter bestehen aus Maschinencode. Dieser Maschinencode wurde mithilfe eines Assemblers erzeugt.

Da unser Mikrocontroller nur Maschinencode verarbeiten kann, muss der gesamte FORTH-Quellcode auf solche A-Wörter zurückgeführt werden. Das ist die Aufgabe des **Compilers**. Wie er dabei vorgeht, das schauen wir uns nun anhand des obigen Beispiels etwas genauer an. Dazu klicken wir erst einmal auf die Lasche “Compiler/Interpreter” und gelangen so wieder in die gewohnte Betriebsart von MikroForth.

Ausgangspunkt unserer Betrachtungen ist zunächst das Wort `main`. Mit diesem Wort soll ja auch der Mikrocontroller seine Arbeit beginnen. Das Wort `main` ruft die Wörter `vorbereiten` und `schalten` auf.

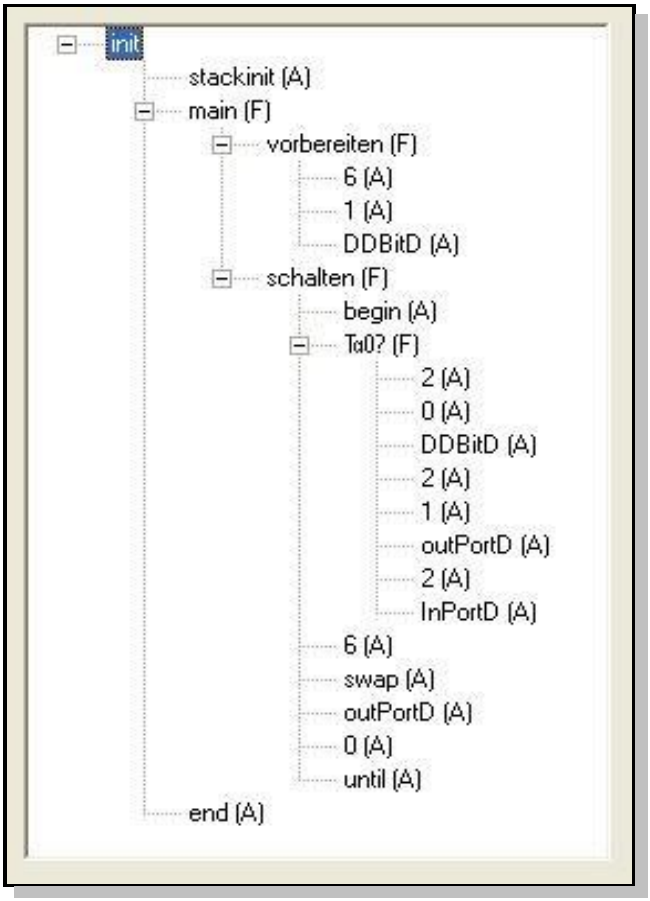


Abb. 2 (17)

Die Parameter von `vorbereiten` sind sämtlich A-Wörter; dagegen taucht unter den Parametern von `schalten` noch das F-Wort `Ta0?` auf. Auch dieses muss wieder analysiert werden; es besteht nur aus A-Wörtern.

Derartige Analysen können in Form eines Baums verdeutlicht werden. Abb. 2 zeigt den Baum für unser Programm. Allerdings besteht die Wurzel unseres Baums nicht aus dem Wort `main`, sondern aus dem Wort

`init`. Das hat folgenden Grund: Unabhängig von den speziellen Aufgaben, welches ein FORTH-Programm zu erfüllen hat, gibt es einige Aufgaben, welche immer ausgeführt werden sollen. In unserem Fall muss vor der Ausführung von `main` der Stack eingerichtet werden; dies geschieht durch das Wort `stackinit`. Nach der Ausführung von `main` soll der Mikrocontroller stets in eine Endlosschleife übergehen; dazu dient das Wort `end`. Natürlich hätte man diese Aufgabe auch dem Anwender überlassen können, aber so ist es bequemer und sicherer. Das Wort `init` ruft also zuerst `stackinit` auf, dann das Wort `main` und schließlich das Wort `end`. Bei Bedarf könnte man sogar das Wort `init` um weitere Standardaufgaben ergänzen, indem man das bestehende Wort `init` durch ein neues überschreibt.

Betätigt man nun die "Kompilieren"-Schaltfläche, wird der Baum aus Abb. 2 rekursiv nach A-Wörtern durchsucht; die zugehörigen Parameter, d. h. die entsprechenden Maschinencodes, werden in die Maschinencode-Tabelle eingetragen. Dabei wird die Startadresse dieser Maschinenprogramme zusätzlich in der **Adresszuweisungstabelle** festgehalten. Mithilfe dieser Zuweisungstabelle kann u. a. kontrolliert werden, ob ein A-Wort schon eingetragen (kompiliert) wurde oder nicht; auf diese Weise wird vermieden, dass ein und dasselbe Wort mehrfach kompiliert wird.

Die Maschinenprogramme der A-Wörter enden alle mit dem `ret`-Befehl (Code $0895). Deswegen können sie als Unterprogramme aufgerufen werden. Beim Kompilieren der F-Wörter werden die als Parameter auftauchenden A-Wörter durch entsprechende Unterprogrammaufrufe ersetzt. Aus den Parametern

```
6
1
DDBitD
```

des Wortes "`vorbereiten`" wird z. B. der Code

```
rcall
<Adresse
von 6>
rcall
<Adresse
von 1>
rcall
<Adresse
von DDBitD>
ret
```

erzeugt, natürlich in bereits assemblierter Form. Auch hier wird wieder von der Adresszuweisungstabelle Gebrauch gemacht. Wie wir sehen, endet dieses Programmteil ebenfalls mit einem `ret`-Befehl; deswegen kann auch dieses Programmteil seinerseits wieder als Unterprogramm aufgerufen werden. Genau diesen Prozess führt der Compiler aus, wenn er in einem zweiten Lauf den Baumnach F-Wörtern durchsucht. Sie werden dann abhängig von Reihenfolge und Suchtiefe in die Adresszuweisungstabelle und die Maschinecodetabelle eingetragen.

So wird z. B. das Wort `Ta0?` vor dem Wort `schalten` eingetragen; es steht zwar hinter dem Wort `schalten`, liegt aber tiefer im Suchbaum.

Von der Maschinencodetabelle zum HEX-Code ist es nur ein kleiner Schritt. Er bedeutet in gewisser Weise nur eine andere Schreibweise. In der Tat muss zum Brennen dieser Schritt sogar wieder rückgängig gemacht werden.

Es ist sehr lehrreich, das Ergebnis eines solchen Kompiliervorgangs einmal detailliert anschauen. Dazu betrachten wir allerdings nicht den Maschinencode selbst, sondern den zugehörigen Assemblercode; diesen lassen wir uns aus dem HEX-Code mithilfe eines so genannten Disassemblers erzeugen. Wir legen allerdings ein etwas kürzeres Beispiel zugrunde:

FORTH-Quelltext:

```
: main 5 5 + . ;
```

HEX-Code:

```
:1000000029C01895189518951895189518951895189518954C
:10001000189518951895189518951895189518951895189578
:10002000189518951895189518951895189518951895189578
:100030000D9308950E911E91010F0D9308951FEFDA
:100040000E9117BB08BB0895FFCFF1DFF0DFF2DFA1
:0C005000F6DF0895E9DFF9DFF7DF08951
F
:00000001FF
```

Assemblercode:

```
            rjmp    avr002A
            reti
            reti
            reti
            reti
            reti
            reti
            reti
            reti
            reti
            reti
            reti
            reti
            reti
            reti
            reti
            reti
            reti
            reti
            reti
avr0014:    ldi     XL, 0x60
            ldi     XH, 0x00
            ret
avr0017:    ldi     r16, 0x05
            st      X+, r16
            ret
avr001A:    ld      r16, -X
            ld      r17, -X
            add     r16, r17
            st      X+, r16
            ret
avr001F:    ldi     r17, 0xFF
            ld      r16, -X
            out     DDRB, r17
            out     PORTB, r16
            ret
avr0024:    rjmp    $
avr0025:    rcall   avr0017
            rcall   avr0017
            rcall   avr001A
            rcall   avr001F
            ret
avr002A:    rcall   avr0014
            rcall   avr0025
            rcall   avr0024
            ret
```

Abbildung 18

Das Programm beginnt mit dem Befehl `rjmp $002A`, einem Sprung zur Adresse $002A. (Die folgenden `reti`-Befehle übergehen wir an dieser Stelle; im Zusammenhang mit den Interrupts werden wir noch ausführlich auf ihre Bedeutung zu sprechen kommen.)
Bei der Adresse $002A beginnt das Unterprogramm zu `init`:

```
rcall $0014          ; stackinit
rcall $0025          ; main
rcall $0024          ; end
```

Als nächstes wird also das Unterprogramm `stackinit` aufgerufen; hier wird der Zeiger für den Stack initialisiert; als Zeiger wird hier das Registerpaar (XH, XL) benutzt. Der Stackzeiger X wird auf den Wert $0060 gesetzt; das ist die unterste Adresse des SRAMs.

Über den `ret`-Befehl springt das Programm wieder zurück zum nächsten Befehl des `init` Unterprogramms; hier wird das `main`-Unterprogramm aufgerufen, das bei der Adresse $0025 beginnt. Da es von einem FORTH-Wort abstammt, setzt es sich seinerseits aus lauter Unterprogrammaufrufen zusammen, abgeschlossen von einem `ret`-Befehl. Zweimal hintereinander wird das Unterprogrammm `it` der Adresse $0017 aufgerufen. Hier wird jeweils die Zahl 5 auf den Stack gelegt. Dazu wird die Zahl 5 zunächst im Register r16 zwischengespeichert. Durch den Befehl `st X+, r16` wird der Inhalt von r16, also unsere Zahl 5, in der Speicherzelle abgelegt, die durch X indiziert wird; anschließend wird X um 1 erhöht, weist danach also auf die nächste Speicherstelle des Stacks.

Durch die nächstenbeiden Unterprogrammaufrufe von `main` wird die Addition ausgeführt (bei Adresse $001A) und das Ergebnis auf Port B ausgegeben (bei $001F). Dabei kann man auch erkennen, wie Zahlen vom Stack geholt werden: Durch den Befehl `ld r16, X-` wird z. B. der Wert aus dem von X indizierten SRAM-Register in das Register r16 geholt und der Zeigerwert um 1 vermindert; damit zeigt X nun auf den darunter liegenden Stackinhalt.

Zu guter letzt wird aus dem Unterprogramm `main` wieder zurückgesprungen zum Unterprogramm `init`. Hier geht es weiter mit

`rcall $0024`; dort wird der Mikrocontroller in eine Endlosschleife geschickt.

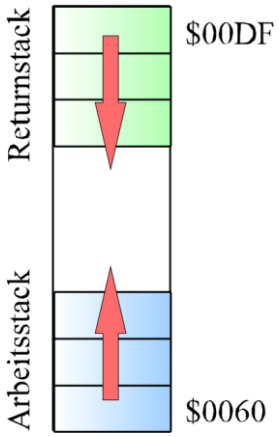


Abb. 3 (19)

Wer sich eingehender mit Assembler-Unterprogrammen beschäftigt hat, der weiß, dass hier ein weiterer Stack benutzt wird, der so genannte **Returnstack**. Der Stack, welchen wir bei der FORTH-Programmierung bislang betrachtet haben, wird zur Unterscheidung oft auch als **Arbeitsstack** bezeichnet. Returnstack- und Arbeitsstack sind beide im SRAM angesiedelt; sie teilen ihn sich:

Während der Arbeitsstack bei $0060 beginnt und dann die darüber liegenden Zellen belegt, fängt der Returnstack bei $00DF an und belegt dann die darunter liegenden Zellen (Abb. 3). Auf diese Weise wird die Gefahr einer Kollision der beiden Stacks möglichst klein gehalten. Im Gegensatz zum Arbeitsstack muss man sich um die Verwaltung des Stackpointers Z beim Returnstack nicht kümmern; dies übernehmen die Befehle `rcall` und `ret` selbstständig.

Wesentliche Idee unseres Forth-Compilers ist also die Verschachtelung von Unterprogrammen. F-Wörter bestehen nur aus Unterprogramm-Aufrufen; diese können auf F- oder auch auf A-Wörter verweisen. Letztlich müssen diese Unterprogrammaufrufe natürlich immer bei A-Wörtern auskommen; denn nur hier findet sich der Maschinencode, der nicht auf ein anderes Wort verweist, sondern tatsächlich "Arbeit verrichtet".

Dieses einfache Konzept führt natürlich zu Einschränkungen. Manche Kontrollstrukturen (IF ELSE-THEN (Kein Versehen bei der Reihenfolge!) oder BEGIN-WHILE-REPEAT lassen sich damit auch nicht realisieren. Dafür bietet dieses Konzept aber die Möglichkeit, recht unkompliziert neue A-Wörter in das Vokabular einzufügen. Wenn Sie daran interessiert sind, sollten Sie gleich das übernächste Kapitel lesen. Im folgenden Kapitel wollen wir uns nämlich etwas eingehender damit beschäftigen, wie überhaupt Kontrollstrukturen realisiert werden können.

10 Funktionsweise der do-loop-Schleife

Wir gehen aus von folgendem Beispiel:

```
10 3 do Bef1 Bef2 loop Bef3
```

Dem entsprechen im Speicher die Unterprogrammaufrufe

```
rcall <10> rcall
<3> rcall <do>
rcall <Bef1>
rcall <Bef2>
rcall <loop>
rcall <Bef3>.
```

Dabei bedeuten die spitzen Klammern “Adresse von...”.

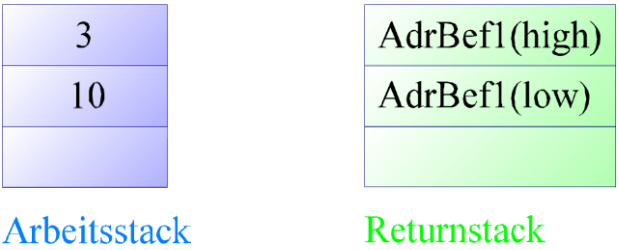


Abbildung 1 (20)

Durch die ersten beiden Unterprogramme werden der Startwert (3) und der Endwert (10) des Schleifenindex auf den Arbeitsstack gelegt. Jedesmal, wenn ein Unterprogramm aufgerufen wird, merkt sich der Mikrocontroller die Adresse des nächsten Befehls, indem er sie auf den Returnstack legt. Wenn also das `do`-Unterprogramm aufgerufen wird, wird die Adresse von `rcall <Bef1>` auf den Returnstack gelegt. Genauer betrachtet besteht diese Adresse aus zwei Bytes; wir bezeichnen

diese Adressen als *AdrBef1(high)* und *AdrBef1(low)*. Unmittelbar nach dem Aufruf des `do`-Unterprogramms sehen unsere beiden Stacks also so aus:

Alles, was das `do`-Unterprogramm leisten muss, ist dafür zu sorgen, dass der Mikrocontroller sich dieses Adresspaar langfristig merkt; nur so kann gewährleistet werden, dass er am Ende eines Schleifendurchlaufs wieder zum Anfang der Schleife kehren kann. Dieses Merken kann nicht über Arbeitsregister wie z. B. r16 geschehen; denn diese könnten z. B. durch das Unterprogramm von `Bef1` oder `Bef2` überschrieben werden.

Ein sicherer Ort zum langfristigen Merken ist der Returnstack. Hier liegt unser Adresspaar zwar schon, aber am Ende des `do`−Unterprogramms wird dieses Adresspaar durch den ret-Befehl vom Returnstack in den Programmzähler geschoben und verschwindet dabei vom Returnstack.

Damit ist aber auch schon die Lösung in Sicht: Das Adresspaar auf dem Returnstack muss von `do` verdoppelt werden; so steht die Kopie auch nach der Ausführung von `do` noch zur Verfügung. Ebenso müssen auch die beiden Schleifenindizes gesichert werden. Dies leistet der folgende Assemblercode:

```
.def AdrH = r16
.def AdrL = r17

.def A = r18
.def E = r19
ld A,-x                 ; Startindex vom
                        Parameterstack
ld E,-x                 ; Schleifenende vom
                        Parameterstack
                        ; Returnadresse vom
pop AdrH pop AdrL       Returnstack

                        ; und wieder drauf (für
push AdrL push AdrH     LOOP)
```

```
push E push A                ; Schleifenparameter auf
                             Returnstack

push AdrL push AdrH          ; Returnadresse noch einmal
                             auf den Stack

ret                          ; zum nächsten Befehl
```

Unmittelbar vor dem `ret`-Befehl des `do`-Unterprogramms und unmittelbar danach sieht der Returnstack dann so aus:

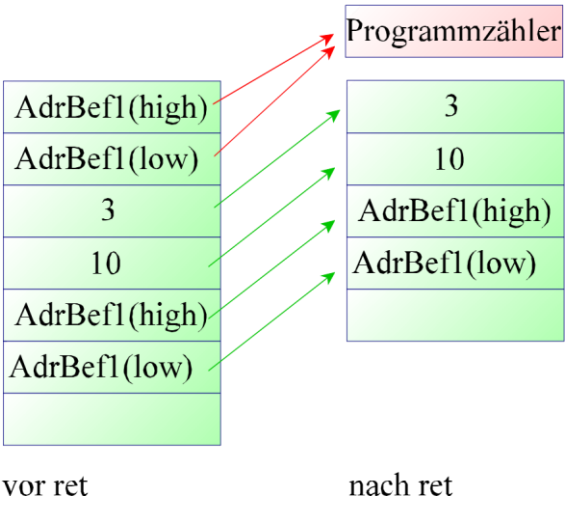


Abbildung 2 (21)

Durch den nun folgenden Unterprogrammaufruf `rcall <Bef1>` wird nun das Adresspaar von `rcall <Bef2>` auf den Returnstack gelegt; beim Rücksprung verschwindet es aber wieder. Und so geht es weiter, bis das folgende `loop`-Unterprogramm aufgerufen wird.

```
.def AdrH = r16      ; Returnstack
.def AdrL = r17
.def A = r18         ; Schleife: aktueller Index
```

```
.def E = r19          ; Endwert des Schleifenindex
.def nextAdrH = r20 ; Adresse von Bef3
.def nextAdrL = r21
pop nextAdrH          ; Adresse von Bef3 retten
pop nextAdrL
                      ;    aktueller    Index    vom
pop A                 Returnstack holen
pop E                 ; Endwert vom Returnstack
                      ; Adresse von Bef1 vom
pop ZH                Returnstack
pop ZL
cp A, E
breq loopende         ; wenn A=E dann nach loopende
                      springen
; sonst (A < E)
push ZL               ; und wieder drauf (für nächstes
push ZH               LOOP)
inc A
push E                ; Schleifenparameter auf
push A                Returnstack
ijmp                  ; Sprung nach Adresse, die in Z
                      steht (s. o.) ; keine zusätzliche
loopende:             Adr auf Stapel!
push nextAdrL
push nextAdrH
ret                   ; zum nächsten Befehl
```

In dieser Situation liegt das Adresspaar von `rcall <Bef3>` auf dem Returnstack. Das wird zunächst einmal vom Returnstack geholt und in Arbeitsregistern (nextAdrH und nextAdrL) zwischengespeichert; die Schleifenindizes 3 und 10 werden ebenso in Arbeitsregistern zwischengelagert (A bzw. E) . Die Werte *AdrBef1(high)* und *AdrBef1(low)* werden in das Registerpaar Z geschoben; dies wird einen indirekten Sprung zumBefehl `rcall <Bef1>` ermöglichen. Nun wird kontrolliert, ob der aktuelle Schleifenindex (3) kleiner als der Endwert

(10) ist. Da dies der Fall ist, wird der Schleifenindex (3) um 1 erhöht, das Adresspaar und die Indizes für einen möglichen weiteren Schleifendurchlauf wieder auf den Returnstack gelegt und der indirekte Sprung `ijmp` ausgeführt. Der bedeutet einen Sprung an die Adresse, welche im Registerpaar Z steht, also zu `rcall <Bef1>`.

Dies wiederholt sich solange, bis der aktuelle Indexwert gleich dem Endwert (10) ist. In diesem Fall wird das zwischengespeicherte Adresspaar von `rcall <Bef3>` wieder auf den Returnstack gelegt; so springt das Programm durch den letzten Befehl von `loop`, nämlich dem `ret`-Befehl, wie gewünscht nicht mehr an den Anfang der Schleife, sondern zum Unterprogrammaufruf von `Bef3`.

Aufgabe

Das Wort `do` kann auch als F-Wort geschrieben werden:

```
: do swap R> R> over over >R
       >R rot >R rot >R >R >R ;
```

Schlagen Sie die Bedeutung von `R>` und `>R` im Vokabular nach und machen Sie sich die Funktionsweise dieser Befehlsfolge anhand von Stackdiagrammen klar.

11 Herstellen von A-Wörtern

Bisher haben wir uns nur darum gekümmert, wie man F-Wörter erzeugt. Dies geschah über die Doppelpunktdefinition im Rahmen des üblichen Interpretier- und Kompiliervorgangs. Für viele Anwendungen ist das auch ausreichend.

Es kann aber vorkommen, dass die im Vokabular zur Verfügung gestellten Wörter nicht ausreichen. In diesem Fall ist es zweckmäßig, das Problem genau zu lokalisieren und dafür ein passendes A-Wort selbst zu erzeugen. Wir wollen dies an einem einfachen Beispiel verdeutlichen.

Die Aufgabe möge darin bestehen, ein Wort zu erzeugen, welches sämtliche Werte auf dem Stack über die serielle Schnittstelle ausgibt; ein solches Wort könnte gut zu Testzwecken eingesetzt werden. Schleifenstrukturen und Wörter für die COM-Ausgabe und das Arbeiten mit dem SRAM stehen im Vokabular schon zur Verfügung. Was noch fehlt, ist ein Wort, dass die Anzahl der Werte auf dem Stack angibt; diese Anzahl bezeichnet man manchmal auch als **Stacktiefe**.

Zur Ermittlung dieser Stacktiefe soll nun ein neues Wort `stackcount` erzeugt werden. Hierzu muss auf den Stackpointer X zurückgegriffen werden; da der bislang durch kein Forth-Wort erfasst ist, muss `stackcount` mit Maschinencode erzeugt werden. Wir müssen also ein A-Wort herstellen.

Der nötige Assemblercode sieht so aus:

```
.def stackcount = r16     ; r0 bis r15
                          reserviert für
                          ;  Interrupts
mov stackcount, XL        ; Stackpointer XL
                          nach stackcount
subi stackcount, $60      ; Stack-Startadresse
                          $60 subtrahieren
```

```
st x+, stackcount          ; Ergebnis auf Stack
ret                        legen
```

Abbildung 1 (22)

Wir assemblieren den Code, zum Beispiel mit Studio 4, und speichern die Hex-Datei unter dem Namen “stackcount.hex” ab. Nun öffnen wir unser Programm MikroForth und klicken auf die Lasche “Vokabular editieren”. Auf der Navigationsleiste klicken wir die +-Schaltfläche an; es öffnet sich ein Fenster zum Editieren von Wörtern; dort tragen wir den Namen des Wortes und den Kommentar so ein wie in Abb. 1 zu sehen.

Nun laden wir den vorbereiteten Hex-Code in das Parameterfeld: Dazu betätigen wir die Schaltfläche “Hex-Datei laden” unterhalb des Parameterfeldes und öffnen unsere Datei “stackcount.hex”. Das Editier-Fenster sieht dann so aus:

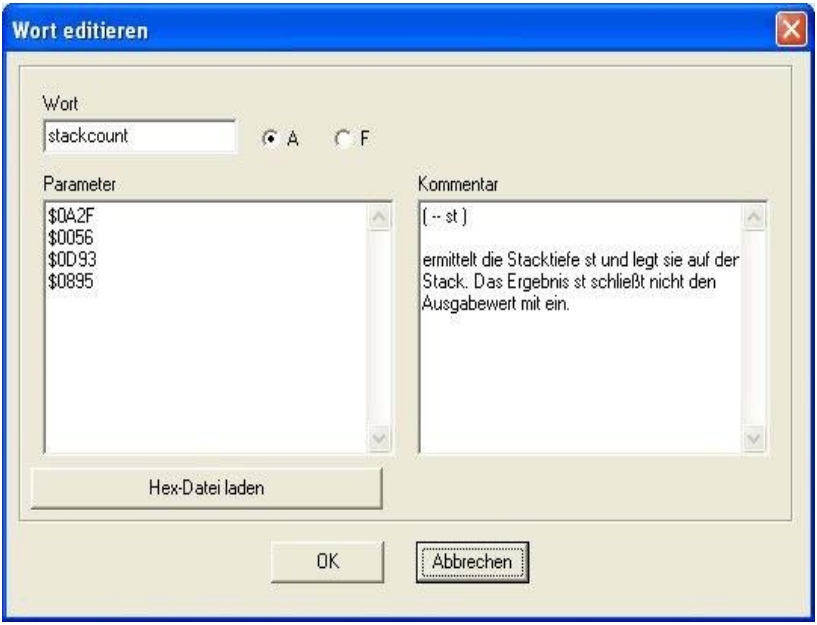


Abbildung 2 (23)

Jetzt bestätigen wir noch die Eingabe mit der OK-Schaltfläche. Damit ist unser neues A-Wort fertig. Es befindet sich allerdings nur im temporären Vokabular. Um es dauerhaft zu speichern, klicken wir abschließend auf die Disketten-Schaltfläche in der Navigationsleiste.

Mithilfe von `stackcount` können wir jetzt leicht das Wort `stack2com` schreiben, welches den Stack über die COM-Schnittstelle ausgibt:

```
: stack2com initcom stackcount
            1 do I 95 + sram> >com 250
            waitms loop ;
```

Was geschieht hier? Zunächst wird die COM-Schnittstelle initialisiert; anschließend wird die Stacktiefe ermittelt und auf den Stack gelegt. Zusammen mit der folgenden Zahl 1 bildet sie End- und Startwert der folgenden `do-loop`-Schleife. Innerhalb der Schleife wird zum

Schleifenindex jeweils die Zahl 95 addiert; das Ergebnis ist die Adresse des jeweiligen Stackregisters im SRAM. Der Inhalt dieses Registers wird dann mit dem Befehl `sram>` auf den Stack gelegt und von `>com` an die serielle Schnittstelle weitergereicht. Eine kurze Wartezeit - wenn auch nicht so lange, wie hier angegeben - ist erforderlich, damit die COM-Übertragung eines Bytes nicht durch die Übertragung des nächsten gestört wird.

Zum Austesten benutzen wir das folgende `main`-Wort:

```
: main 11 22 33 44 55 66 77 88 99
       stack2com ;
```

Hierdurch werden die Zahlen 11, 22 bis 99 auf den Stack gelegt und gleich darauf durch `stack2com` ausgegeben. Das können wir leicht nachkontrollieren, indem wir im Uploader-Programm das Terminal-Programm aktivieren.

Abbildung 3 (24)

12 Rekursion mit MikroForth

Das einfache Konzept unseres FORTH-Compilers bedingt, dass ein Forth-Wort sich nicht direkt selbst aufrufen kann. Lassen wir z. B. das folgende Programm

```
: rektest 1 . rektest ;
: main rektest ;
```

interpretieren, so erhält man die Meldung:

```
Fehler: Das Wort "rektest" wurde im Vokabular
nicht gefunden.
"rektest" wurde nicht im Vokabular eingetragen!
Warnung: Interpretiervorgang abgebrochen...
```

Allerdings können wir die Kenntnisse über die Art und Weise, wie MikroForth arbeitet, ausnutzen, um mit einem Trick doch noch eine Rekursion zu realisieren. Dazu rufen wir aus der Doppelpunktdefinition von `rektest` nicht das Wort `rektest` selbst auf, sondern ein anderes Wort `zu_rektest`, welches seinerseits dafür sorgt, dass `rektest` ausgeführt wird. Allerdings kann das Wort `zu_rektest` das Wort `rektest` nicht wie üblich aufrufen; dies hätte nur eine Fehlermeldung wie oben zur Folge.

Wie kann dieser Aufruf von `rektest` nun anders realisiert werden? Hierzu nutzen wir aus, dass die Adresse des Wortes, das als nächstes auszuführen ist, auf dem Returnstack zu liegen hat. Das Wort `zu_rektest` muss also nichts anderes leisten, als die Adresse von `rektest` auf den Returnstack zu legen.

An einem konkreten Beispiel soll die Vorgehensweise erläutert werden. Die Zahlen von 1 bis 32 sollen ausgegeben werden, zuerst vorwärts zählend und dann rückwärts.

```
: zu_rektest 0 122 >r >r ;
: rektest dup dup . 255 waitms 1 + dup
          32 equal skipIf
          zu_rektest . 255 waitms ;
: main 1 rektest . ;
```

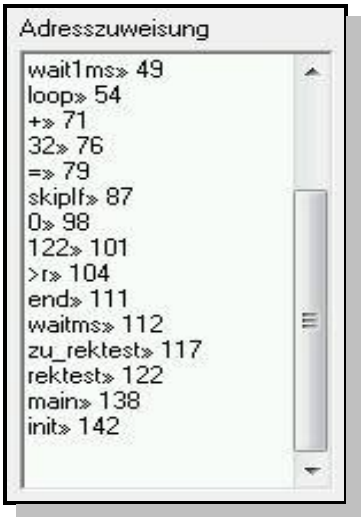


Abbildung 1 (25)

Das Wort `zu_rektest` legt die Adresse von `rektest` auf den Returnstack. Diese Adresse besteht hier aus dem Highbyte 0 und dem Lowbyte 122. Wie gelangt man an diese Adresse? Dazu braucht man nach dem Kompilieren nur die Adresszuweisungstabelle anschauen. Die drittletzte Zeile in Abb. 1 zeigt, dass die gesuchte Adresse 122 ist. Für die Adressfindung ist es nicht gleichgültig, welche Werte für die beiden Adressbytes in der Definition von `zu_rektest` zunächst eingesetzt worden sind. Eine nachträgliche Kontrolle mit der gefundenen Adresse empfiehlt sich also.

Die Rekursionstiefe wird durch die Größe des SRAMs begrenzt.

13 Interrupts

MikroForth unterstützt auch das Interrupt-Konzept des Attiny-Mikrocontrollers. Allerdings gibt es im Standardvokabular nur Wörter für die Interrupts INT0, INT1 und T0OVF (Timer/Couter0 Overflow). Einer Erweiterung des Vokabulars für andere Interrupts steht jedoch nichts imWeg.

Am Beispiel des INT0-Interrupts soll nun dargelegt werden, wie die Interrupt-Programmierung in MikroForth erfolgt. Für grundlegende Fragen zum Interrupt-Konzept wird auf das entsprechende Kapitel verwiesen.

Und das soll unser Programm leisten: Ein an Port D.6 angeschlossener Beeper soll die ganze Zeit über tönen. Eine LED an Port B.0 soll währenddessen über eine fallende Flanke an Ta0 geschaltet werden; ob durch diesen Tastendruck diese LED ein- oder ausgeschaltet wird, soll durch den Zustand von Ta1 festgelegt werden.

Wie üblich muss der INT0-Interrupt zunächst initialisiert werden. Dies geschieht durch das Wort `initInt0`:

```
initInt0        ( f – )
```

Der Wert von f entscheidet darüber, ob der Interrupt durch eine fallende (f = 0) oder steigende Flanke (f = 1) am INT0-Eingang (Port D.2) erfolgt. Durch die Initialisierung mit `initInt0` werden auch folgende Aktionen erledigt:

- Port D.2 als Eingang konfigurieren,
- Port D.2 auf high legen,
- Interrupts freigeben.

In unserem Fall muss f = 0 sein, denn durch das Drücken von Ta0 geht D.2 von high nach low über.

Wird der INT0-Interrupt ausgelöst, wird nun automatisch das Wort `int0` ausgeführt. Die Bezeichnung dieses Wortes ist fest vorgegeben. Der Wortkörper kann jedoch beliebig vom Benutzer programmiert werden; jedoch muss dabei folgende Form eingehalten werden:

```
: int0 pushreg <benutzerdefinierter Teil>
       popreg reti ;
```

Das hat folgenden Grund: Der Interrupt wird in der Regel mitten in der Ausführung eines anderen Wortes erfolgen. Damit die Registerinhalte r16-r29, welche dieses andere Wort womöglich benutzt, nicht verloren gehen, werden sie als erstes durch das Wort `pushreg` gerettet. Dazu kopiert `pushreg` deren Inhalte in die Register r2 bis r15, die bei den A-Wörtern nicht zum Einsatz kommen (dürfen). In dem benutzerdefinierten Bereich von `int0` können jetzt alle Wörter beliebig benutzt werden. Am Ende werden durch `popreg` alle Register r16 bis r29 wieder hergestellt; so kann das in seiner Ausführung unterbrochene Wort korrekt weiterarbeiten.

Beim Auslösen des Interrupts wird der Attiny für weitere Interrupts global gesperrt. Um ihn wieder frei zu geben für erneute Interrupts, muss die Definition von `int0` mit dem Wort `reti` enden.

In unserem Fall sieht die Definition des Wortes `int0` so aus:

```
: int0 pushreg Ta1? . popreg reti ;
```

Der benutzerdefinierte Teil ist sehr kurz: Durch `Ta1?` wird der Zustand des Tasters Ta1 abgefragt. Ist Ta1 gedrückt, wird eine 0 auf den Stapel gelegt und diese dann an Port B ausgegeben; andernfalls wird eine 1 auf

den Stapel gelegt und dann ausgegeben. Die LED wird also, je nachdem ob Ta1 gedrückt ist oder nicht, ein- bzw. ausgeschaltet.

Das gesamte Programm sieht dann so aus:

```
: int0 pushreg Ta1? . popreg reti ;
: BeeperAnAus 6 1 outportD 10 waitms
              6 0 outPortD 10 waitms ;
: main 6 1 DDBitD 0 initInt0
       begin BeeperAnAus 0 until ;
```

Im Hauptprogramm wird durch eine Endlosschleife für eine Schwingung mit der Periodendauer 2 ⍰ 10 ms am Port D.6 gesorgt.

Aufgabe

Testen Sie das Programmaus; betätigen Sie dazu den Taster Ta0 mehrfach bei unterschiedlichen Tasterzuständen von Ta1. Lassen Sie nun den `reti`-Befehl weg und führen Sie den Test erneut durch.

14 MikroForth einstellen

MikroForth erlaubt folgende Anpassungen:

1. Programm zum Uploaden
2. Warnhinweis bei Überschreiben von Wörtern
3. Auswahl des Separators in der Adressenzuweisungstabelle
4. Anzeigen der Adressen im HEX- oder Dezimalformat

Die entsprechenden Einstellungen sind inder Datei `forth2.ini` gespeichert. Sie können dort bei Bedarf mit einem Editor geändert werden.

Programm zum Uploaden

Soll auf das Programm "Uploader.exe" zurückgegriffen werden, besteht der Eintrag in der ini-Datei einfach aus einem Minuszeichen:

```
externuploader=-
```

Ansonsten wird hinter das Gleichheitszeichen der Name des gewünschten Programms mitsamt dem vollständigen Pfad angegeben.

Warnhinweis bei Überschreiben von Wörtern

Häufig müssen einzelne Wörter des aktuellen Vokabulars überschrieben werden. Ist der Eintrag

```
ueberschreiben=1
```

dann gibt MikroForth einen entsprechender Warnhinweis in einem Meldungsfenster. Sie haben dann die Möglichkeit, das Überschreiben zu unterbinden. Wenn Sie den Wert 0 hinter das Gleichheitszeichen

schreiben, dann erfolgt nur ein Hinweis im Statusbereich und das alte Wort wird überschrieben.

Sie können diesen Parameter auch über Bearbeiten - Einstellungen ändern.

Auswahl des Separators in der Adressenzuweisungstabelle

In der Adresszuweisungstabelle befindet sich zwischen dem Forth-Wort und der zugehörigen Adresse ein so genannter **Separator**. Standardmäßig ist dies ein Doppel-Größer-Zeichen mit dem ASCII-Code 187. Sie können dieses Zeichen in der Zeile

```
separator=187
```

ändern. Geben Sie dazu hinter dem Gleichheitszeichen einen anderen Code ein. Dieser Code darf aber nicht zu einem Zeichen gehören, das im Namen eines Forth-Wortes auftaucht. Daher empfiehlt es sich, nur ASCII-Codes oberhalb von 127 zu benutzen.

Anzeigen der Adressen im HEX- oder Dezimalformat

Sie können auswählen, ob die Adressen im Hex- oder im Dezimalformat angezeigt werden sollen. Dazu wird die Zeile

```
hexadressen=1
```

benutzt. Bei dem Eintrag 1 werden die Adressen im HEX-Format angezeigt, bei dem Eintrag 0 in dezimaler Schreibweise.

Sie können diesen Parameter auch über Bearbeiten - Einstellungen ändern.

15 Forth-Vokabular

Stand: 01.11.2012

A: Assembler-Wort **F:** Forth-Wort **C:** Compiler-Wort

Wort	Typ	Kommentar	Stack
.	A	gibt TOS auf Port B aus; (Datenrichtungsbits von Port B werden alle auf 1gesetzt.)	(n –)
_	A	erg = a - b	(a b – erg)
/	A	dividiert a (ganzzahlig) durch b erg = a/b (ohne Rest)	(a b – erg rest)
:	C	leitet Doppelpunktdefinition ein.	
;	C	schließt Doppelpunktdefinition ab.	
<	A	`a b <` legt 1 (TRUE) auf den Stack, wenn a < b ist, sonst 0.	(a b – flag)
>	F	`a b >` legt 1 (TRUE) auf den Stack, wenn a > b ist, sonst 0.	(a b – flag)

`>com`	A	sendet TOS an die COM-Schnittstelle. Vorher muss die COM-Schnittstelle mit INITCOM initialisiert worden sein.	(n –)
`>eprom`	A	schreibt den Wert w in die Adresse a des EEPROMs. Vgl. `eprom>`	(w a –)
`>R`	A	schiebt den TOS auf den Returnstack Vgl. `R>`	(a –)
`>sram`	A	speichert den Wert w in der SRAM-Zelle mit der Adresse a	(w a –)
`1 bis 255`	AC	legt die Zahl auf den Stack.	(– n)
`and`	A	erg = a and b	(a b – erg)

Wort	**Typ**	**Kommentar**	**Stack**
`begin ... until`	A	`begin Bef1 Bef2 ... Befn until` wiederholt die Befehle `Bef1`, `Bef2`, ..., `Befn`, bis `until` auf TOS = 1 stößt. `begin until`	(–) (n –)

`blink`	F	*bitmuster hp* `blink` gibt *bitmuster* auf Port B aus, wartet *hp* Millisekunden, gibt 0 auf Port B aus und wartet wieder *hp* Millisekunden.	(b hp –)
`com>`	A	legt über COM-Schnittstelle empfangenes Byte auf Stack. Vgl. `>com`.	(– n)
`DDBitB`	A	*bit flag* `DDBitB` setzt den Anschluss bit des Ports B als Ausgang, wenn *flag* = 1, sonst als Eingang.	(bit flag –)
`DDBitD`	A*	*bit flag* `DDBitB` setzt den Anschluss *bit* des Ports B als Ausgang, wenn *flag* = 1, sonst als Eingang.	(bit flag –)
`DDRB`	A	schreibt b in das Datenrichtungsregister des Ports B.	(b –)
`DDRD`	A*	schreibt d in das Datenrichtungsregister des Ports D.	(d –)

`do ... loop`	A	*e a* `do Bef1 Bef2 ... Befn loop` wiederholt die Befehle `Bef1`, `Bef2`, ..., `Befn`; die Schleife beginnt mit dem Index *a* und läuft bis *e* (einschließlich). Die Schleife wird mindestens einmal durchlaufen. Innerhalb der Schleife kann durch das Wort `I` auf den Index zurückgegriffen werden. `do loop`	(e a –) (–)
`drop`	A	entfernt den TOS	(n –)
`dup`	A	dupliziert den TOS	(n – n n)

Wort	**Typ**	**Kommentar**	**Stack**
`end`	A	führt eine Endlosleerschleife aus; wird für das Ende eines Programms empfohlen.	(–)
`eprom>`	A	liest den Wert w aus der EEPROMAdresse a und legt ihn auf den Stack Vgl. `>eprom`	(a – w)
`getOSCCAL`	A	legt den OSCCAL-Wert auf den Stack. Vgl. `SetOSCCAL`	(– n)

`I`	A	legt den Schleifenindex einer `do-loop`Schleife auf den Stack. Darf nur zwischen `do` und `loop` auftauchen.	(– n)
`i2cread`	A	Ein Wert wird vom Slave gelesen; wenn ACK = 0 ist, wird ein AcknowledgeSignal gegeben.	(ACK – Wert)
`i2cstart`	A	Startsignal für I2C-Bus wird gesendet (SDA von 1 auf 0; dann SCL von 1 auf 0)	(–)
`i2cstop`	A	initialisiert den I2C-Bus (SCL und SDA auf 1); Datenrichtungsbits für SDA (PortB.5) und SCL (PortB.7) werden gesetzt.	(–)
`i2cwrite`	A	Ein einzelnes Byte wird an den Slave gesendet; das Acknowledge-Signal wird auf den Stack gelegt.	(Wert/Adr – ACK)
`init`	F	Systemwort, darf nicht geändert oder entfernt werden.	
`initCom`	A	initialisiert die COM-Schnittstelle: D0 = RxD D1 = TxD Baudate = 9600 8 Bit	(–)

		kein Paritätsbit	

Wort	Typ	Kommentar	Stack
`initInt0`	A	*signaltyp* `initInt0` konfiguriert INT0 (Port D2) als Interrupteingang und legt diesen auf High. Je nach *signaltyp*-Wert lösen unterschiedliche Eingangssignale den Interrupt aus: 0: fallende Flanke 1: steigende Flanke Interrupts werden generell zugelassen.	(signaltyp –)
`initInt1`	A	wie `initInt0`, jedoch für den Eingang INT1 (Port D3).	(signaltyp –)
`initT0ovf`	A	*typ preset* `initT0ovf` initialisiert den Timer0-Interrupt: *typ* 0: Timer stoppen/deaktivieren 1: Systemtakt/1 2: Systemtakt/8 3: Systemtakt/64 4: Systemtakt/256	(typ preset –)

		5: Systemtakt/1024 6: ext. Takt, fallend an T0 7: ext. Takt, steigend an T0 Timer-Interrupt werden freigegeben alle Interrupts werden freigegeben *preset*-Wert muss in Interruptroutine immer wieder neu gesetzt werden.	
`inPortB`	A	*bit* `InPortB` liest den Eingang *bit* des Ports B und legt 1/0 auf den Stack, wenn er High/Low ist. Vgl. `DDRB` und `DDBitB`	(bit – flag)
`inPortD`	A	*bit* `InPortD` liest den Eingang *bit* des Ports D und legt 1/0 auf den Stack, wenn er High/Low ist. Vgl. `DDRD` und `DDBitD`	(bit – flag)

Wort	**Typ**	**Kommentar**	**Stack**

`int0`	F	Dieses Wort wird aufgerufen wenn das INT0-Interrupt ausgelöst wird.
		Aufbau eines Interruptwortes:
		`: int0 pushreg ...`
		`<beliebige Wörter>`
		`...`
		`popreg reti;`
		Während das Wort `int0` ausgeführt wird, sind sämtliche Interrupts gesperrt.
			(–)

`int1`	F	Dieses Wort wird aufgerufen, wenn das INT1-Interrupt ausgelöst wird. Kann beliebig definiert werden.	(–)
`not`	F	ersetzt *flag* durch sein logisches Komplement.	(*flag* –)
`or`	A	erg = a or b	(a b – erg)
`outPortB`	A	*bit flag* `outPortB` setzt den Ausgang *bit* des Ports B auf High/Low, wenn *flag* = 1/0 ist. Vgl. `DDRB` und `DDBitB`	(bit flag –)
`outPortD`	A	*bit flag* `outPortD` setzt den Ausgang *bit* des Ports D auf High/Low, wenn *flag* = 1/0 ist. Vgl. `DDRD` und `DDBitD`	(bit flag –)

`over`	A	kopiert das zweite Element des Stacks auf den TOS.	(a b – a b a)
`popreg`	A	Sämtliche internen Register r16-r29 werden wiederhergestellt.	(–)
`pushreg`	A	Sämtliche internen Register r16-r29 werden gesichert (in r2-r15).	(–)
`R>`	A	holt das oberste Element des Returnstacks und legt es auf den (Arbeits-) Stack. Vgl. `>R`	(– a)
`reti`	A	Interrupts werden freigegeben.	(–)
`rot`	A	rotiert die obersten drei Zahlen des Stacks.	(a b c – b c a)
`sei`	A	wie `reti`	
`setOSCCAL`	A	schreibt den Wert n in das OSCCALRegister.	(n –)

Wort	**Typ**	**Kommentar**	**Stack**
`setTimer0`	A	setzt den Preset-Wert (TCNT0) von Timer0.	(preset –)

`skipIf`	A	überspringt den nächsten Befehl, wenn TOS gleich 1 (TRUE) ist.	(n –)
`sram>`	A	legt den Wert der SRAM-Zelle a auf den Stack	(a – w)
`stackInit`	A	Systemwort, darf nicht geändert oder entfernt werden	
`swap`	A	vertauscht die beiden obersten Zahlen des Stacks.	(n m – m n)
`T0ovf`	F	Dieses Wort wird aufgerufen, wenn das Timer0-Overflow-Interrupt ausgelöst wird. Zum Aufbau eines Interrupt-Wortes vgl. `int0`. Innerhalb von `T0ovf` muss ggf. der Preset-Wert des Timers mit `setTimer0` gesetzt werden.	(–)
`Ta0?`	F	Legt 1/0 auf Stack, wenn Taster Ta0 offen/geschlossen (D2=1/0) PortD.2 wird automatisch konfiguriert.	(– bit)

`Ta1?`	F	Legt 1/0 auf Stack, wenn Taster Ta1 offen/geschlossen (D3=1/0) PortD.3 wird automatisch konfiguriert.	
`toggleB`	A	toggelt das Register von Port B.	(–)
`VARIABLE`	C	leitet Variablendeklaration ein. Durch `VARIABLE` *abc* wird die Variable *abc* deklariert. Dadurch wird durch den Compiler im EEPROM ein Speicherplatz reserviert. Anschließend wird durch *abc* die Adresse von der zugehörigen Speicherstelle auf den Stapel gelegt.	
`wait`	F	wartet s Sekunden.	(s –)
`wait1ms`	A	wartet 1 Millisekunde.	(–)
`waitms`	F	wartet n Millisekunden.	(n –)

Wort	**Typ**	**Kommentar**	**Stack**

`wdogOff`	A	schaltet den Watchdog aus.	(–)
`wdogOn`	A	schaltet den Watchdog an.	(–)
`xor`	A	erg = a xor b	(a b – erg)
□	A	*a b* = legt 1 (TRUE) auf den Stack, wenn *a* = *b* ist, sonst 0 (FALSE).	(a b – flag)
□	A	erg = a + b	(a b – erg)
□	F	erg = a * b	(a b – erg)

http://www.g-heinrichs.de/wordpress/index.php/attiny/

Exeter 2017_07_01 v8_A5

Notizen: ####