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.

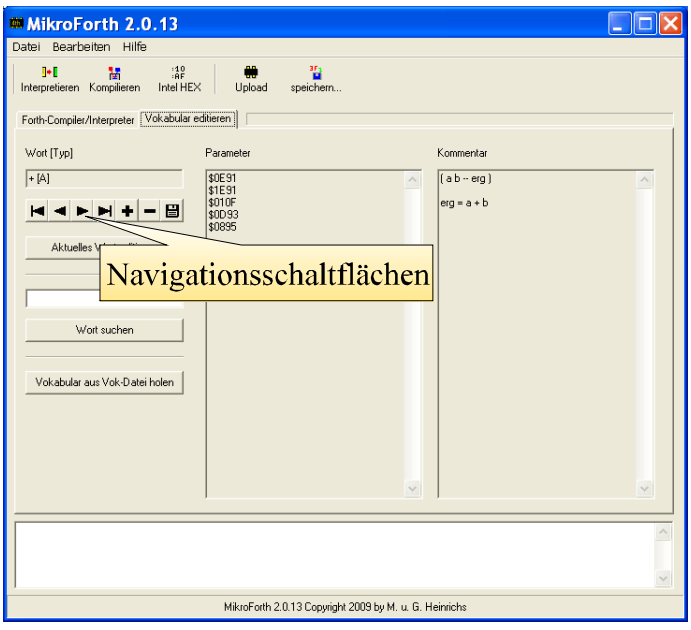

Abb. 1

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 in einem 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 Doppelpunktdefinition beschrieben worden ist. Beim **Interpretieren** wird also im Wesentli-

chen 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 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. 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.

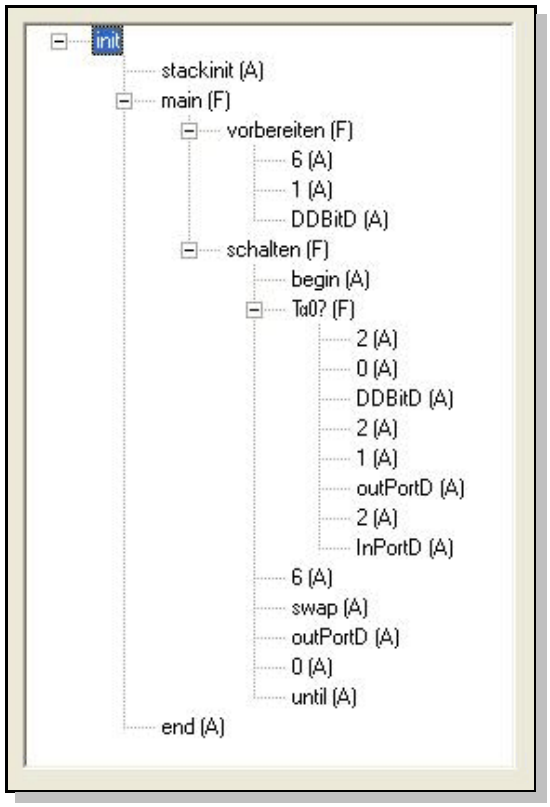

Abb. 2

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 Baum nach 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:

```
:1000000029C018951895189518951895189518954C
:10001000189518951895189518951895189518951895189578
:10002000189518951895189518951895189518951895189578
:100030000D9308950E911E91010F0D9308951FEFDA
:100040000E9117BB08BB0895FFCFF1DFF0DFF2DFA1
:0C005000F6DF0895E9DFF9DFF7DF08951F
: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
```

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. Zwei mal hintereinander wird das Unterprogramm mit 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 nächsten beiden 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.

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.

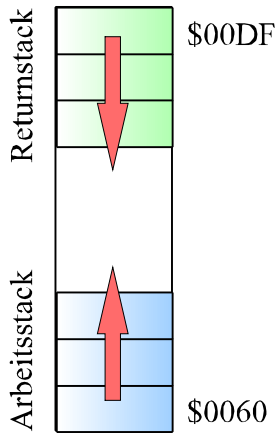

Abb. 3

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 eines 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.