4

4 Pointer und Felder

Mit Pointern, Feldern, sowie indirektem Adressieren betreten wir die „höheren Sphären“ des Assemblerprogrammierens.

4.1 Pointer

Zur Einführung sei angenommen, ein Ausschnitt aus der Symboltabelle enthalte eine „Variable 1“, eine „Variable 2“, sowie eine weitere zunächst rätselhafte Variable „Pointer“. Diese stehen auf den Tabellenplätzen 112, 113 und 114, heißen V1, V2 sowie pV und haben zunächst irgendwelche nicht weiter interessierenden Symbolwerte 6, 7 und -5. Das sieht in der Symboltabelle wie folgt aus::

Symboladresse	Symbolname	Symbolwert
112	V1	6
113	V2	7
114	pV	-5

Wir wollen Zahlenwert 10 auf V1 speichern. Im Normalfall werden wir hierfür den Befehl „`mov V1 10`“ ausführen. Dieser Zugriff ist der sog. *direkte* Zugriff. Im direkten Zugriff greift nicht nur der Move auf Variablen zu, sondern die meisten RTA-Befehle. Dass sich V1 in der 112. Zeile der Symboltabelle befindet („auf Symboladresse 112“) ist dabei eher uninteressant.

Nun führen wir den Befehl

```
        adrof     pV        V1                  ; pV wird Pointer auf V1
```

aus. Dieser trägt die Adresse von V1 als Symbolwert von pV ein. Das ist nun gerade die 112:

112	V1	6
113	V2	7
114	pV	112

Nun wird folgender Befehl ausgeführt:

```
        put       pV        .         10        ; V1 über Pointer beschreiben
```

(Den Punkt zwischen pV und 10, den sog. Offset, schreiben wir gewissenhaft mit, ohne ihn zunächst weiter zu beachten.) Anschließend bietet sich folgendes Bild:

112	V1	10
113	V2	7
114	pV	112

Das Ergebnis gleicht zunächst dem des Move-Befehls von oben. Der Zugriff erfolgte allerdings nicht mehr über den Symbolnamen „V1“. Statt dessen gibt es nun das Symbol „pV“ mit dem Symbolwert 112. Dies ist nun die Symboladresse von V1, was letztendlich beschrieben wird. Die Variable pV „zeigt“ somit gleichsam auf V1 und gibt sich somit als Zeiger oder Pointer auf V1 zu erkennen. Wir sprechen vom *indirekten Zugriff oder Pointerzugriff.*

Mit indirektem Lesen und Schreiben können nun ungleich vielfältigere Zugriffe auf Variablen ausgeführt werden als in direkter Art. Beim direkten Zugriff muss der Programmierer den Namen der Variablen zum Zeitpunkt der Programmentwicklung kennen. Pointer hingegen kann man auch vom Programm aus lesen, verändern und berechnen. Das eröffnet viele Möglichkeiten.

Wir erhöhen den Pointer um Eins:

```
        add     pV      1                   ; pV wird um 1 erhöht
```

Nun zeigt pV nicht mehr auf die Variable V1, sondern auf den nächsten Symboltabellenplatz. Das ist nun die Variable V2:

112	V1	10
113	V2	7
114	pV	113

Der Pointer wurde „um 1 weitergestellt“. Wenn man jetzt den gleichen Befehl wie oben

```
        put     pV      .       10
```

nochmals ausführt, wird nicht mehr V1, sondern V2 beschrieben:

112	V1	10
113	V2	10
114	pV	113

Eine solche „Pointer-Weiterstellung“ ist mit dem direkten Zugriff nicht möglich. Nun bleibt noch die Bedeutung des Offsets zu erläutern: Der Offset q wird zum Pointer p hinzuaddiert, bevor der Zugriff erfolgt. So lassen sich solche „Pointerweiterstellungen“ wie soeben von V1 nach V2 sehr elegant programmieren:

```
        adrof   pV      V1
        put     pV      0       10      ; V1 mit 10 beschreiben
        put     pV      1       10      ; V2 mit 10 beschreiben
```

4.2 Felder

Der Pointerzugriff ist insbesondere ein mächtiger Mechanismus zum Arbeiten mit Feldern. Was sind Felder? Felder sind Datenstrukturen, in denen mehrere Zahlen gespeichert werden können. Sie müssen zunächst mit dem Pseudobefehl **_dim** deklariert werden, um sie anzulegen. Der Befehl

```
        _dim    Q       4                   ; Feld Q mit 4 Elementen
```

legt z. B. ein Feld „Q“ mit einer **Feldlänge** 4 an.

Was bedeutet dies im Einzelnen? Der _dim-Befehl erzeugt in der Symboltabelle zunächst ein Symbol Q, sowie weitere 5 Symbole, die die Symbolnamen Q(0), Q(1), Q(2), Q(3) und Q(4) erhalten. Dann wird die Symboladresse von Q(0) auf den Symbolwert von Q geschrieben. Dadurch wird Q zum **Feldpointer.** Q kann nun so wie der ober erwähnte Pointer pV benutzt werden. Die Symbole Q(0) bis Q(4) sind die **Feldelemente.** Die Zahlen 0, 1, 2, 3, 4 sind die **Feldindizees.** Mit dem Feldpointer und dem „nullten Element“ belegt ein Feld in der Symboltabelle immer zwei Einträge mehr, als die Feldlänge angibt. Es entsteht folgender Symboltabellenausschnitt:

Symboladresse	Symbolname	Symbolwert
36	Q	37
37	Q(0)	0
38	Q(1)	0
39	Q(2)	0
40	Q(3)	0
41	Q(4)	0

Natürlich sind auch beliebig andere Feldlängen möglich. Allerdings muss die Feldlänge immer eine Zahl sein. Eine Deklaration per „`_dim Q c`“ würde nicht funktionieren. Hier würde der Wert *c* immer als Feldlänge 0 interpretiert werden.

Nun stehe die Beispielaufgabe, das Feld Q mit Quadratzahlen zu füllen. Dies kann mit folgendem Code geschehen:

```
        mov     Q(0)    0
        mov     Q(1)    1
        mov     Q(2)    4
        mov     Q(3)    9
        mov     Q(4)    16
        ...
```

Wenn man nun aber die Quadrate bis 100 berechnen will, wird dies sehr mühsam. Außerdem lautet die Aufgabe ja, dass das Programm die Quadrate *berechnen* soll und nicht, dass der Programmierer die Quadratzahlen *eintippen* soll.

Dieser Code hingegen löst die Aufgabe:

```
        _dim    Q       100             ; Feld Q(0) ... Q(100)
        clr     i                       ; Index i löschen
m:
        mov     r0      i               ; i nach r0
        power   r0      2               ; r0 quadrieren
        put     Q       i       r0      ; r0 nach Q(i) schaffen
        inc     i                       ; i um 1 erhöhen
        cmple   i       100     m       ; Wenn i≤100 weiter bei m
        ...
```

Der _dim-Befehl legt das Feld Q mit den Elementen Q(0) bis Q(100) an.
Dann wird die Indexvariable *i* gelöscht, d. h. auf 0 gesetzt.
Der weitere Code enthält eine Schleife. Diese beginnt bei der Marke *m*. In dieser Schleife passiert folgendes:

`mov r0 i`	schafft den Index i nach Register 0.
`power r0 2`	rechnet in Register 0 das Quadrat von *i* aus.
`put Q i r0`	füllt das *i*-te Feldelement von Q mit Register 0.
`inc i`	erhöht *i* um 1.
`cmple i 100 m`	springt zu *m*, wenn der Code noch nicht 100× abgearbeitet [wurde.

Interessant ist, dass der Index *i* dreifach genutzt wird, nämlich a) als Zahl, die quadriert wird, b) als Offset, um in Q zu adressieren, und c) als Abbruchbedingung in dem cmple-Befehl.

In diesem Code kann man nun einfach die „100“ in _dim und cmple gegen eine „10000“ ändern und schon werden die Quadratzahlen bis 10000 berechnet.

Anmerkung: Eine Notation „Q(i)“ als Variablenname (so wie etwa in höheren Programmiersprachen) ist in Assembler nicht zulässig. Mit etwa einem „`mov Q(i) a`“ würde in RTA lediglich eine Variable „Q(i)“ neu definiert und dann mit a beschrieben, keinesfalls aber das „*i*-te“ Feldelement von *Q*.

4.3 Die Befehle read und write in Felder

Die Befehle write und read sind so konzipiert, dass sich damit auch Felder in Dateien lesen und schreiben lassen. Hierbei wird der Längenparameter der Befehle benutzt. Dabei ist zunächst ist zu berücksichtigen, dass ein Feld der Feldlänge *n* nicht nur *n* Feldelemente 1 … *n* enthält, sondern 2 Werte mehr: Auch Feldpointer und das Feldelement Nr. 0 sind zu übertragen.

Nun übertragen read und write immer einen Wert mehr, als der Längenparameter angibt. Das stellt sicher, dass read und write auch bei Einzelvariablen korrekt funktionieren. Der hier fehlende Längenparameter wird als Null angenommen, es wird genau eine Variable übertragen.

Um nun die für Felder erforderlichen *n*+2 Symbolwerte zu übertragen, führt Längenparameter *n*+1 zum richtigen Transfer. Bei Länge *n*+1 werden *n*+2 Symbolwerte übertragen: Feldpointer, Element Nr. 0 und die Elemente 1 bis *n.*

Um das Feld Q mit der Feldlänge 4 Q zu schreiben, nutze man also den Befehl

```
        write     Q         5
```

Dieser erzeugt die Datei „q.dat“ mit den 6 Textzeilen

```
37¶
 0¶
 1¶
 4¶
 9¶
16¶
```

Die erste Zeile enthält den Feldpointer, der uninteressant ist, es folgen die Feldelemente Q(0), Q(1), Q(2), Q(3) und Q(4).

Das Gegenstück zum write bildet der read-Befehl, mit dem ein Feld aus einer Datei in die Symboltabelle eingelesen wird.

```
        read      Q         4
        adrof     Q         Q(0)
```

Wichtig: Beim Einlesen eines Feldes aus einer Datei per read-Befehl muss der Feldpointer wieder richtig einstellt werden. Das muss *immer* mit einem dem read unmittelbar folgenden adrof-Befehl geschehen. Ohne diesen adrof würde der Feldpointer auf einen falschen Bereich der Symboltabelle zeigen. Dies hätte dann zur Folge, dass nachfolgende get falsch lesen und nachfolgende put sogar die Symboltabelle zerstören können!

Wir fassen zusammen

1. Ein Feld *feldname* der Feldlänge *n* wird mit „`_dim` ***`feldname`*** `{`***`feldlänge`***`}`“ angelegt.

2. Die Anzahl der tatsächlich vorhandenen Feldelemente ist incl. „nulltem Feldelement“ *n*+1.

3. Die Anzahl der Speicherplätze, die das Feld in der Symboltabelle belegt ist incl. Feldpointer und „nulltem Feldelement“ *n*+2.

4. Der in read und write anzugebende Längenparameter ist mit *n*+1 anzugeben. Tatsächlich werden *n*+2 Symbolwerte übertragen. Ein Feld wird also mit einem

`write`	***`feldname`***	`{`***`feldlänge`***`+1}`	geschrieben und mit einem
`read`	***`feldname`***	`{`***`feldlänge`***`+1}`	gelesen.

5. Nach dem ein Feld per read-Befehl gelesen worden ist, immer einen adrof-Befehl angeben:

```
read      feldname  {feldlänge+1}
adrof     feldname   feldname(0)
```