

# Beschreibung der Assemblersprache RT

RTA ist eine Programmiersprache, die lediglich 97 Befehle kennt. Dies genügt, um turingvollständig zu sein. Damit kann man in RT sämtliche mit einer mathematischen Formel beschreibbaren Algorithmen implementieren.

## Die Befehlsliste

### Einfache Befehle

mov	b a	a nach b schaffen (Transportbefehl)
clr	a	a auf 0 setzen (Löschbefehl)
inc	a	$a = a + 1$ (Inkrement)
dec	a	$a = a - 1$ (Dekrement)

### Grundrechenarten

add	a b	$a = a + b$
sub	a b	$a = a - b$
mul	a b	$a = a * b$
div	a b	$a = a / b$

### Höhere Rechenarten

power	a b	$a = a \text{ hoch } b$
root	a b	$a = b\text{-te Wurzel aus } a$

### Exponential- und Logarithmusfunktionen

exp	a	$a = e \text{ hoch } a$
exp10	a	$a = 10 \text{ hoch } a$
exp2	a	$a = 2 \text{ hoch } a$
expx	a b	$a = b \text{ hoch } a$
log	a	$a = \text{natürlicher Logarithmus aus } a \text{ (Basis } e)$
log10	a	$a = \text{dekadischer Logarithmus aus } a \text{ (Basis } 10)$
log2	a	$a = \text{dyadischer Logarithmus aus } a \text{ (Basis } 2)$
logx	a b	$a = \text{beliebiger Logarithmus aus } a \text{ (Basis } b)$

### Winkel- und Arkusfunktionen

sin	a	$a = \sin(a)$
cos	a	$a = \cos(a)$
tan	a	$a = \tan(a)$
cot	a	$a = \cot(a)$
sec	a	$a = \sec(a) = 1 / \cos(a)$
csc	a	$a = \operatorname{cosec}(a) = 1 / \sin(a)$
asin	a	$a = \arcsin(a)$
acos	a	$a = \arccos(a)$
atan	a	$a = \arctan(a)$
acot	a	$a = \operatorname{arccot}(a)$
asec	a	$a = \operatorname{arcsec}(a)$
acsc	a	$a = \operatorname{arccosec}(a)$

### Hyperbel- und Areafunktionen

sinh	a	$a = \text{sinus hyperbolicus } (a)$
cosh	a	$a = \text{cosinus hyperbolicus } (a)$
tanh	a	$a = \text{tangens hyperbolicus } (a)$
coth	a	$a = \text{cotangens hyperbolicus } (a)$

sech	a	a = secans hyperbolicus (a)
csch	a	a = cosecans hyperbolicus (a)
asinh	a	a = area sinus hyperbolicus (a)
acosh	a	a = area cosinus hyperbolicus (a)
atanh	a	a = area tangens hyperbolicus (a)
acoth	a	a = area cotangens hyperbolicus (a)
asech	a	a = area secans hyperbolicus (a)
acsch	a	a = area cosecans hyperbolicus (a)

### Logische Befehle

bin	a	a = 1, sofern $a \neq 0$ , sonst 0 (Log. Identität)
not	a	a = 1, sofern $a = 0$ , sonst 0 (Log. Negation)
and	a b	a = 1, sofern $a \neq 0$ und $b \neq 0$ , sonst 0
or	a b	a = 1, sofern $a \neq 0$ oder $b \neq 0$ , sonst 0

### Sonstige Rechenbefehle

abs	a	Absolutbetrag von a bilden
neg	a	Vorzeichen von a umkehren (Negation)
sgn	a	Vorzeichen von a (+1, 0 oder -1)
round	a	a runden
ceil	a	a aufrunden
floor	a	a abrunden
fix	a	Vorkommastellen von a (a zur 0 hin runden)
frac	a	Nachkommastellen von a
clip	a b c	a = a, wenn $a < b$ dann a = b, wenn $a > c$ , dann a = c
cmod	a b c	a mit Modulofunktion (Sägezahn) in b ... c clippen
random	a	a = Zufallszahl zwischen 0 und 1

### Sprungbefehle

cmpgt	a b m	wenn $a > b$ , Sprung nach m („compare greather than“)
cmpge	a b m	wenn $a \geq b$ , Sprung nach m („compare greather equal“)
cmplt	a b m	wenn $a < b$ , Sprung nach m („compare less than“)
cmple	a b m	wenn $a \leq b$ , Sprung nach m („compare less equal“)
cmpeq	a b m	wenn $a = b$ , Sprung nach m („compare equal“)
cmpne	a b m	wenn $a \neq b$ , Sprung nach m („compare not equal“)
tstgt	a m	wenn $a \geq 0$ , Sprung nach m („test greather than“)
tstge	a m	wenn $a > 0$ , Sprung nach m („test greather equal“)
tstlt	a m	wenn $a < 0$ , Sprung nach m („test less than“)
tstle	a m	wenn $a \leq 0$ , Sprung nach m („text less equal“)
tsteq	a m	wenn $a = 0$ , Sprung nach m („test equal“)
tstne	a m	wenn $a \neq 0$ , Sprung nach m („test not equal“)
jump	m	Unbedingter Sprung nach m

### Ein-/Ausgabebefehle

input	a s	fordert a mit Text s im Dialog an (mit Pause)
output	a s	zeigt a mit Text s im Dialog an (mit Pause)
pause	s	Programmpause mit Dialogtext s (mit Pause)
proof	a s	Testausgabe von a mit Mitteilung s (ohne Pause)
info	s	Mitteilung s (ohne Pause)
cls		Löschen des Ausgabertextes
printn	a b c	Ausgabe der Zahl a mit b Vor- und c Nachkommastellen
prints	s	Ausgabe der Zeichenkette s in den Ausgabertext
save	s	Speichern des Ausgabertextes in Textdatei s
read	a b	Zahl a (b=0) oder Feld a (b=Länge-1) aus Datei lesen
write	a b	Zahl a (b=0) oder Feld a (b=Länge-1) in Datei speichern

### Indirekt- oder Pointerbefehle

adrof	p a	Adresse von a nach p schaffen. p wird Pointer auf a
get	a p q	Das Symbol mit der Adresse (p+q) nach a schaffen
put	p q a	a auf das Symbol mit der Adresse (p+q) schaffen

### Systembefehle

init		Erste Zeile eines Programms (wird automatisch erzeugt)
nop		Nullbefehl. Dieser Befehl macht nichts
mode	a	a=0: 'Ohne Halt', 1: 'Fehlerhalt', 2: 'Schrittweise'
halt		Hält das Programm an
errcode	a	Fehlercode nach a. 0: Kein Fehler.
errjump	m	Bei Fehler Sprung nach m
exit		Programmbeendigung

### Pseudobefehle

Pseudobefehle sind Befehle, die nicht in fertigen Programm („zur Laufzeit“), sondern vom Assembler („zur Assemblerzeit“) abgearbeitet werden.

_name	s	Dem Programm den Namen s geben
_var	a	Variable a deklarieren
_dim	a i	Feld a mit den Feldelementen a, a(0) ... a(i) deklarieren
_lab	m	Marke m definieren
_config	i	Virtuelle Maschine mit Parameter i konfigurieren
_end		Programmende

Für den markendefinierenden Befehl „\_lab xyz“ gibt es auch die Kurzschreibweise „xyz:“.

(Die Operanden bedeuten:

a, b, c, p, q	Zahlen oder Variablen
i	nur Zahlen. Variablen gelten als Null.
m	Marken
s	Zeichenketten)

## Das Programm

Ein Programm besteht aus Befehlen. Jeder Befehl steht auf einer Zeile. Ein Befehl kann bis zu 3 Operanden haben:

```
befehl operand1 operand2 operand3
```

Die Befehle werden in ihrer Reihenfolge abgearbeitet, z. B.:

```
mov    a    b    ; Transportiere b nach a
div    a    10   ; Dividiere a durch 10
sin    a    ; Berechne nun den Sinus von a
```

Auf diese Art wird z. B.  $a = \sin(b / 10)$  berechnet.

Befehle und Operanden werden durch ein oder mehrere Leerzeichen/Tabulatoren getrennt.

Eine Zeile kann auch leer sein. Nach Semikolon kann Kommentar folgen. Wenn eine Zeile mit dem Zeichen „`␣`“ (Zeichencode Alt/182) endet, so setzt die folgende Zeile diese Zeile fort.

## Operanden

Alle Operanden werden in einer Symboltabelle verzeichnet und haben dort einen *Symbolnamen*, einen *Symbolwert* und eine *Symboladresse*. Der Symbolname ist die Zeichenkette, die im Programm steht. Auf dem Symbolwert stehen die Zahlen, mit dem der Befehl rechnet. Die Zeilennummer eines Symbols in der Symboltabelle heißt Symboladresse. Ein Symbol kann als *Variable*, *Zahl*, *Zeichenkette* oder *Marke* genutzt werden.

**a) Variablen:** Auf jedem Symbol kann ein beliebiger Zahlenwert gespeichert werden. Symbolnamen wie z. B.

```
a x1 alpha b11 aber auch ??? (° 055$ oder [PM343]
```

lassen sich so als Variablen nutzen. RTA unterscheidet Groß- und Kleinschreibung. „A“ und „a“ sind also zwei verschiedene Variablen.

**b) Zahlen:** Wenn sich der Symbolname als Zahlenwert interpretieren lässt, so wird der Symbolwert beim Programmstart mit diesem Zahlenwert gefüllt. So kann man mit Symbolnamen wie z. B.

```
1 2 +1.5 -3.3E6 -2.3E-2
```

die Zahlen 1, 2, 1.5, -3300000 -0.023 usw. erzeugen. Das Dezimaltrennzeichen ist der Punkt, niemals das Komma. Ein E oder e wird als Exponentialkennzeichen aufgefasst.

**c) Marken:** Sprungbefehle benötigen Marken als Ziele. Dies sind Symbole deren Symbolwert als Programmzeilennummer aufgefasst wird. Derartige Marken können mit Markendefinitionsbefehlen wie „\_lab markel“ (bzw. „markel:“) gesetzt werden.

**d) Zeichenketten:** Manche Befehle benötigen Zeichenketten als Operanden z. B. für Dialogtexte. Hier wird der Symbolname als Zeichenkette verwendet. In Zeichenketten gilt „~“ als Leerzeichen und „\“ als Zeilenumbruchzeichen. Zeichenketten dürfen bis zu 1024 Zeichen lang sein. Beispiel:

```
\Beginn~der~Berechnung\\
```

Variablen, Zahlen, Marken und Zeichenketten sind lediglich verschiedene Interpretationen von Symbolen, keinesfalls etwa verschiedene Datentypen. RTA kennt keine unterschiedlichen Datentypen. Es ist eine ganz einfache Sprache. Jeder Operand ist immer nur ein Symbol.

## Vordefinierte Variablen

Bei jedem Programmstart werden bestimmte vordefinierte Symbole automatisch bereitgestellt:

.	Das Leersymbol. Wird immer mit 0 gelesen.
..	Der Befehlszeiger. Hier steht immer die Adresse des aktuellen Befehls.
pi	$\pi$
pi/2	$1/2 \pi$
pi/4	$1/4 \pi$
e	Die Eulersche Zahl e
®	Erdradius am Äquator in Metern (R-Zeichen=Alt/0174)
®f	Abplattung des Erdellipsoides. (Den Polradius errechne man mit $\text{®} - (\text{®f} * \text{®})$ )
° (	Faktor, der Gradmaß in Bogenmaß umrechnet (Gradzeichen=Alt/0176))
(°	Faktor, der Bogenmaß in Gradmaß umrechnet (Gradzeichen=Alt/0176)
eps	Kleinste, von Null verschiedene Zahl
max	Größte zulässige Zahl
r0 ... r7	8 Register zur allgemeinen Verwendung
x y	Gegebene, zu transformierende Koordinaten
x' y'	Gesuchte, transformierte Koordinaten
z	eine Testvariable

<code>z'</code>		eine Testvariable
<code>Rx</code>	<code>Ry</code>	ein Erd-Äquatorialradius in einem Quellbild-Geokoordinatenmaß (in x- bzw. y-Richtung)
<code>Rx'</code>	<code>Ry'</code>	ein Erd-Äquatorialradius in einem Zielbild-Geokoordinatenmaß (x-/y-Richtung)
<code>Cx</code>	<code>Cy</code>	ein Bildmittelpunkt in einem Quellbild-Geokoordinatenmaß (x-/y-Richtung)
<code>Cx'</code>	<code>Cy'</code>	ein Bildmittelpunkt in einem Zielbild-Geokoordinatenmaß (x-/y-Richtung)

## Ein- und Ausgabe

Es gibt drei Möglichkeiten, Zahlenwerte in ein Assemblerprogramm ein- und auszugeben: *Dialoge*, einen *globalen Ausgabertext* und *Dateien*.

**a) Dialoge:** Mit den Befehlen `input` und `output` können Variablen über Textfenster eingegeben und angezeigt werden. In diesen Gruppe gehören auch die Befehle `pause`, `proof` und `info`.

**b) Globaler Ausgabertext:** Es gibt einen globalen Ausgabertext, in den beliebige Texte mit den Befehlen `printn` und `prints` geschrieben werden können. Der Ausgabertext kann mit dem `cls`-Befehl gelöscht und mit dem `save`-Befehl als Datei gespeichert werden.

**c) Dateien:** Mit den Befehlen `write` und `read` lassen sich Variablen in Dateien abspeichern und wieder einlesen. Diese Dateien sind gewöhnliche Textdateien mit dem Dateityp „.dat“, wobei ein Symbolwert als Zahl in einer Zeile der Datei als Text gespeichert wird. Der Dateiname ist der Variablenname. Wenn der 2. Operand `b` mit einem Wert  $> 0$  belegt wird, so übertragen die Befehle nicht nur einen einzelnen Symbolwert, sondern  $b+1$  in der Symboltabelle aufeinanderfolgende Symbolwerte. Dies kann für Felder genutzt werden (s. u.).

## Felder und Pointer

*Felder:* Der Befehl „`__dim a 10`“ definiert zunächst ein gewöhnliches Symbol `a`. Weiterhin werden 11 weitere Symbole mit den Namen `a(0)`, `a(1)`, `a(2)`, `a(3)` ... `a(10)` erzeugt. Dies sind die Feldelemente. Schließlich trägt `__dim` auf `a` die Nummer der Zeile, auf welcher `a(0)` in der Symboltabelle steht, ein. Damit wird `a` zu einem Pointer auf das Feld („Feldpointer“).

*Der Pointerzugriff:* Die Befehle `put` und `get` realisieren den Pointerzugriff, bei dem ein Symbol nicht direkt über seinen Namen, sondern indirekt angesprochen wird. Diese Befehle enthalten als Operanden einen Pointer `p` sowie einen Offset `q`. Pointer und Offset ergeben zusammengerechnet die Adresse des Symbols, welches geschrieben/gelesen wird. Auf diese Art liefert der Befehl „`get z a i`“ das Feldelement Nr. `i` des Feldes `a` nach `z`. Der Befehl „`put b 7 22`“ schreibt eine 22 auf das 7. Element eines Feldes `b`. Dieser Befehl macht folglich dasselbe, wie der Befehl „`mov b(7) 22`“. Der Unterschied ist, dass die 7 beim `mov` eine feste Zahl sein muss, während beim `put` auch auf eine Variable als Offset möglich ist. — Pointer sind nicht unbedingt an Felder gebunden. Man kann mit Ihnen auch auf gewöhnliche Symbole zugreifen. Die Symboladresse stellt dann der Befehl `adrof` bereit.

Mit den Befehlen `__dim`, `put`, `get` und `adrof` können nicht nur Felder, sondern auch Stacks, Matrizen, Listen und ähnliche Datenstrukturen organisiert werden. Wenn der Offset `q` bei `put` oder `get` nicht benötigt wird, so setze man ihn einfach auf das Leersymbol „.“

*Die Befehle write und read in Felder:* Mit den Befehlen `write` und `read` kann man auch Felder aus Dateien lesen bzw. in Dateien schreiben. Hierfür ist als 2. Operand der Index des letzten zu schreibenden Feldelementes *plus 1* anzugeben – auf der ersten Dateizeile steht der Feldpointer, auf der zweiten Dateizeile das nullte Feldelement. Der Feldpointer ist nach einem `write`-Befehl in der Datei im allgemeinen unbrauchbar belegt. Nach einem `read` in ein Feld steht der Feldpointer daher falsch und muss vor seiner Verwendung mit dem `adrof`-Befehl richtig eingestellt werden, z. B. mit „`adrof f f(0)`“.

## Abarbeiten und Debuggen

Es gibt 3 Programmabarbeitungsmodi: 0=Abarbeitung 'Ohne Halt', 1=Abarbeitung mit 'Fehlerhalt', 2='Schrittweise' Abarbeitung. Den Modus kann man an der Benutzeroberfläche einstellen oder mit dem Befehl `mode` setzen.

Weiterhin gibt es zwei Signale, mit denen man das laufende Programm beeinflussen kann.

Das *Haltsignal* bewirkt eine Programmunterbrechung nach dem laufenden Befehl. Es kann vom Debugger aus oder mit dem Befehl `halt` gesetzt werden.

Das *Abbruchsignal* bewirkt eine sofortige Programmbeendigung. Es wirkt wie der Befehl `exit` und kann auch mit dem Z-Interrupt (Eingabe von Ctrl/Z) ausgelöst werden.

## Hinweise zum Programmieren

*Datentyp:* Der Assembler kennt keine Datentypen. Die Symbolwerte werden mit Double-Gleitkommazahlen gerechnet. Es sind Zahlenbeträge bis  $\pm 1E99$  zulässig, intern sogar bis  $\pm 1E308$ . Die Berechnungen erfolgen auf 14 Dezimalstellen genau. Faustregel: Über 100.000.000.000.000 kann es zu Einschränkungen der Genauigkeit kommen.

*Groß- und Kleinschreibung:* In Symbolnamen werden Groß- und Kleinbuchstaben unterschieden. „A“ und „a“ sind also zwei verschiedene Symbole.

*Sonderzeichen:* Symbolnamen können beliebig Sonderzeichen enthalten, ja ausschließlich aus Sonderzeichen bestehen. So sind z. B. `°` (`°` und `@` gültige Symbole. Man beachte auch, dass „a,“ ein gültiger Symbolname ist und der Befehl `„mov a, b“` keinesfalls eine Variable b nach a schafft, sondern vielmehr ggf. eine Variable mit dem Symbolnamen „a,“ erzeugt und mit b beschreibt.

Die Klammern werden in Symbolnamen allerdings vom `_dim`-Befehl für Feldelemente benutzt und sollten nicht anderweitig genutzt werden. Ebenso ist das Semikolon in Symbolnamen unzulässig: Programmtext ab dem Semikolon ist Kommentar.

In Dateinamen der Befehle `read`, `write` und `save` werden nur die Sonderzeichen `_`, `(`, `)` und `$` geduldet. Andere Zeichen werden in den Unterstrich umgewandelt. Auch werden Großbuchstaben in Kleinbuchstaben umkodiert. Um Schwierigkeiten zu vermeiden, sollte in diesbezüglichen Symbolnamen nur Kleinbuchstaben, Ziffern und die Sonderzeichen `_`, `(`, `)` und `$` verwendet werden.

*Zahlen:* Man beachte, dass Zahlen auch nur ganz gewöhnliche Symbole sind und damit genauso wie Variablen – beschreibbar. Nach Ausführung des Befehls `„mov 1 a“` steht der Wert von a auf dem Symbol mit dem Namen „1“. Nachfolgende Befehle lesen die 1 dann nicht mehr als 1, was irreführend ist. Auch sollte man Symbole wie `pi` oder `e` nicht verändern.

*Das Leersymbol:* Fehlen Operanden (Symbole) in Befehlen, so tritt an ihre Stelle das Leersymbol `.`. So werden fehlende Operanden mit 0 gelesen. Das ist in den meisten Fällen sinnvoll.

*Leersymbol und Befehlszeiger:* Die Symbole `.` und `..` sind nicht beschreibbar. Diese Symbole enthalten immer den Wert 0 bzw. die Zeilennummer des aktuellen Befehls.

*Kein Deklarationszwang:* Jedes Symbol wird bei seiner erstmaligen Verwendung automatisch in die Symboltabelle eingetragen. Symbole brauchen also nicht deklariert zu werden. Dessenungeachtet ist eine Deklaration mit dem Befehl `_var` sinnvoll. Die Symboltabelle wird so besser lesbar und Doppeldeklarationen infolge von Tippfehlern sammeln sich „unten“ in der Symboltabelle an. Dort können sie schnell entdeckt werden. Felder müssen allerdings immer (mit `_dim`) deklariert werden. (Sonst weiß der Assembler ja nicht, wie lang sie sein sollen.)

*Felder und Pointer:* Typisch in der Assemblerprogrammierung ist, dass es keine Pointerüberprüfung gibt. Mit falschen Pointern kann man sich sehr schnell die Symboltabelle zerstören, darum bitte

- Felder unbedingt mit `_dim` deklarieren.
- Die Feldlänge unbedingt als Zahl angeben. Variablen gelten als 0.
- Symbolnamen wie „a(64)“ ausschließlich für Feldelemente nutzen.
- Symbolnamen wie „a(b)“ meiden. Hier würde lediglich ein Symbol mit dem Namen `a(b)` in der Symboltabelle aufgesucht, keinesfalls aber das b-te Element eines Feldes `a`, wie in höheren Programmiersprachen.
- Vorsicht auch beim `read`-Befehl in Felder. Hier soll die im Befehl angegebene Länge die im `_dim` deklarierte Feldlänge sein. Ist der Wert größer, so wird die Symboltabelle über das Feldende hinaus geschrieben und damit möglicherweise zerstört.

*Zuse Z17:* Vorläuferimplementation ist die virtuelle Rechenmaschine Zuse Z17, die mit einem dem RTA verwandten Z17-Assembler programmiert wird. Die Z17-Befehle `max`, `eps`, `com`, `sft`, `sftl`, `sfti`, `sftd`, `sftdr`, `sftdl`, `sftp`, `sftpl`, `sftpr`, `int`, `rnd`, `pow`, `sqr`, `toa`, `tob`, `toc`, `tod`, `froma`, `fromb`, `fromc`, `fromd`, und `valid` werden von RTA nicht unterstützt. Z17 hingegen kennt keine höheren Rechenoperationen wie Sinus, Logarithmus etc. Ansonsten arbeiten beide virtuellen Assembler gleich.

## Fehlercodes

101:	Überlauf. Betrag > 9E99	(Die Fehlercodes 101 bis 113 sind Laufzeitfehler, die beim Abarbeiten eines Programmes durch den RT-Prozessor entstehen können.)
102:	Division durch 0	
103:	Potenz 0 hoch 0	
104:	Rationale Potenz von Zahl < 0	
105:	Rationale oder gerade Wurzel aus Zahl < 0	
106:	Wurzelexponent = 0	
107:	Logarithmus aus Zahl < 0	
108:	Logarithmus aus 0	
109:	Logarithmenbasis < 0	
110:	Logarithmenbasis = 0	
111:	Logarithmenbasis = 1	
112:	Funktionswert nicht definiert	
113:	Datei-Ein-/Ausgabefehler	
116:	Unbekannter Befehl	(Die Fehlercodes 116 bis 120 sind Assemblerfehler, die bei dem Übersetzen des Programmes durch den RT-Assembler, das vor dem Programmstart erfolgt, entstehen können.)
117:	Symbol nicht definiert	
118:	Symbol bereits definiert	
119:	Symboltabelle voll	
120:	Symbolname länger als 1024 Zeichen	

Weil RTA eine Parallelprozessorsprache ist, gibt es bei der normalen Abarbeitung keine Fehlerunterbrechungen. Es wird immer mit irgendeinem Wert weitergerechnet. Es wird aber ein Fehlercode erzeugt, der mit dem Befehl `errcode` gelesen oder auf den mit dem Befehl `errjump` reagiert werden kann. Außerdem werden die jeweils letzten Laufzeitfehler eines jeden Befehls in einem Laufzeitprotokoll eingetragen und gezählt.