

Sprachbeschreibung der Assemblersprache RT

RTA („Reduced Transliteration Assembler“) ist eine Programmiersprache, die lediglich 96 Befehle kennt. Dies ist für einen Assembler sehr wenig, genügt aber, um turingvollständig zu sein. Damit kann man in RTA sämtliche mit einer mathematischen Formel beschreibbaren Algorithmen programmieren.

1. Die Befehlsliste

Rechenbefehle

mov	b a	$b \leftarrow a$	(Transportbefehl)
clr	a	$a \leftarrow 0$	(Löschbefehl)
inc	a	$a \leftarrow a+1$	(Inkrement)
dec	a	$a \leftarrow a-1$	(Dekrement)
add	a b	$a \leftarrow a+b$	
sub	a b	$a \leftarrow a-b$	
mul	a b	$a \leftarrow a*b$	
div	a b	$a \leftarrow a/b$	
power	a b	$a \leftarrow a^b$	
root	a b	$a \leftarrow b\text{-te Wurzel aus } a$	
exp	a	$a \leftarrow e^a$	
exp10	a	$a \leftarrow 10^a$	
exp2	a	$a \leftarrow 2^a$	
expx	a b	$a \leftarrow b^a$	
log	a	$a \leftarrow \log_e(a)$	
log10	a	$a \leftarrow \log_{10}(a)$	
log2	a	$a \leftarrow \log_2(a)$	
logx	a b	$a \leftarrow \log_b(a)$	
sin	a	$a \leftarrow \sin(a)$	
cos	a	$a \leftarrow \cos(a)$	
tan	a	$a \leftarrow \tan(a)$	
cot	a	$a \leftarrow \cot(a)$	
sec	a	$a \leftarrow \sec(a) = 1/\cos(a)$	
csc	a	$a \leftarrow \operatorname{cosec}(a) = 1/\sin(a)$	
asin	a b	$a \leftarrow \arcsin(a), \cos^*$	[b: Indem b das Vorzeichen
acos	a b	$a \leftarrow \arccos(a), \sin^*$	[der xxx*)-Funktion angibt,
atan	a b	$a \leftarrow \arctan(a), \cos^*$	[kann der Befehl auch einen
acot	a b	$a \leftarrow \operatorname{arccot}(a), \sin^*$	[auf $-\pi \dots +\pi$ erweiterten
asec	a b	$a \leftarrow \operatorname{arcsec}(a), \operatorname{csc}^*$	[Wertebereich abdecken.
acsc	a b	$a \leftarrow \operatorname{arccosec}(a), \operatorname{sec}^*$	[(„360-Grad-Eindeutigkeit“)
sinh	a	$a \leftarrow \sinus \text{ hyperbolicus } (a)$	
cosh	a	$a \leftarrow \cosinus \text{ hyperbolicus } (a)$	
tanh	a	$a \leftarrow \text{tangens hyperbolicus } (a)$	
coth	a	$a \leftarrow \text{cotangens hyperbolicus } (a)$	
sech	a	$a \leftarrow \text{secans hyperbolicus } (a)$	
csch	a	$a \leftarrow \text{cosecans hyperbolicus } (a)$	

asinh	a	$a \leftarrow \text{area sinus hyperbolicus } (a)$
acosh	a	$a \leftarrow \text{area cosinus hyperbolicus } (a)$
atanh	a	$a \leftarrow \text{area tangens hyperbolicus } (a)$
acoth	a	$a \leftarrow \text{area cotangens hyperbolicus } (a)$
asech	a	$a \leftarrow \text{area secans hyperbolicus } (a)$
acsch	a	$a \leftarrow \text{area cosecans hyperbolicus } (a)$
bin	a	$a \leftarrow 1$, sofern $a \neq 0$, sonst 0 (Log. Identität)
not	a	$a \leftarrow 1$, sofern $a = 0$, sonst 0 (Log. Negation)
and	a b	$a \leftarrow 1$, sofern $a \neq 0$ und $b \neq 0$, sonst 0
or	a b	$a \leftarrow 1$, sofern $a \neq 0$ oder $b \neq 0$, sonst 0
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	$a \leftarrow a$; wenn $a < b$ dann $a \leftarrow b$; wenn $a > c$ dann $a \leftarrow c$
cmod	a b c	a mit Modulofunktion (Sägezahn) in b ... c clippen
random	a	$a \leftarrow$ Zufallszahl zwischen 0 und 1

Verzweigungsbefehle

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 $a > 0$, Sprung nach m („test greather than“)
tstge	a m	wenn $a \geq 0$, 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 („test 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; b=0: Ausgabe mit variabler Länge, c=0: Ganzzahlausgabe
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

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

Steuerbefehle

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
err	a b	Fehlercode nach a, im Fehlerfall 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

2. Das Programm

Das Grundprinzip aller Assembler ist ganz einfach: Ein Programm besteht aus Befehlen. Jeder Befehl steht auf einer Zeile, wobei hinter einem Befehl bis zu 3 Operanden stehen können:

befehl op1 op2 op3

Befehle und Operanden werden normalerweise durch Tabulatoren getrennt, die alle 8 Zeichen stehen. Der Befehl steht dabei normalerweise hinter einem ersten Tabulator, also in der 2. Spalte. Leerzeichen trennen aber auch.

Die Befehle werden nun einfach 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.

Weitere Regeln: Nach Semikolon kann Kommentar folgen, normalerweise ab der 5. Spalte. Eine Zeile kann auch nur Kommentar enthalten, das Semikolon steht dann meist ganz am Anfang („Kommentarzeile“). Sie kann auch völlig leer sein („Leerzeile“). Wenn eine Zeile mit dem Zeichen „¶“ (Zeichencode Alt/182) endet, so setzt die folgende Zeile diese Zeile fort, wobei dann führende Leerzeichen und Tabulatoren ignoriert werden („Fortsetzungszeile“).

3. 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 denen der Befehl rechnet. Die Zeilennummer eines Symbols in der Symboltabelle heißt Symboladresse. Ein Symbol kann als *Variable*, *Zahl*, *Zeichenkette* oder *Marke* genutzt werden.

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

```
a x1 alpha b11 aber auch ??9 (° 055$ oder [PM343]
```

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

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.

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. Dabei steht die Form „markel:“ in der ersten Spalte, die deshalb auch Markenspalte heißt.

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.

4. 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	π
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 @ - (@f * @))
° (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
z'	eine Testvariable
Rx Ry	ein Erd-Äquatorialradius in einem Quellbild-Geokoordinatenmaß (in x- bzw. y-Richtung)
Rx' Ry'	ein Erd-Äquatorialradius in einem Zielbild-Geokoordinatenmaß (x-/y-Richtung)
Cx Cy	ein Bildmittelpunkt in einem Quellbild-Geokoordinatenmaß (x-/y-Richtung)
Cx' Cy'	ein Bildmittelpunkt in einem Zielbild-Geokoordinatenmaß (x-/y-Richtung)

5. Ein- und Ausgabe

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

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`.

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.

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 Text in einer Dateizeile gespeichert wird. Der Dateiname ist der Variablenname.

Der 2. Operand ist ein Längenparameter, mit Hilfe dessen auch mehr als ein einzelner Symbolwert übertragen werden kann. Dabei kommt jede Zahl in eine Dateizeile. Es wird immer *ein Wert mehr* übertragen, als angegeben. Im Normalfall ist der Längenparameter leer, so dass er mit Null angenommen wird. Folglich wird *eine Variable* übertragen. Ist der Längenparameter mit einem Wert n belegt, so werden $n+1$ fortlaufende Werte aus der Symboltabelle übertragen. Dies kann für Felder genutzt werden (s. u.).

6. Felder und Pointer

(Für Fortgeschrittene.)

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 der `_dim` auf a die Symboladresse, also die Symboltabelle-Zeilenummer von $a(0)$, ein. Damit wird a zu einem Pointer auf das Feld („Feldpointer“).

Der Pointerzugriff: Die Befehle `put` und `get` realisieren den Pointerzugriff. Hier wird ein Symbol nicht direkt über seinen Symbolnamen, sondern indirekt über seine Symboladresse angesprochen. 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: Beim `write` und `read` nutze man den 2. Operanden, den Längenparameter. Zunächst ist hier zu beachten, dass ein Feld der Länge n nicht nur die n Feldelemente $1 \dots n$ enthält, sondern 2 Werte mehr: den Feldpointer und das Feldelement Nr. 0. Weiterhin zu berücksichtigen, dass der Längenparameter immer ein Element weniger angibt, als tatsächlich übertragen wird. Somit ist also nun nicht $n+2$, sondern lediglich $n+1$ im Längenparameter anzugeben. Genau $n+1$ führt zum richtigen Transfer: Es werden übertragen: Feldpointer, Element Nr. 0 und die Elemente 1 bis n .

Der mit einem `write` in der Datei geschriebene Feldpointer ist im Allgemeinen uninteressant. Nach einem `read` in ein Feld ist davon auszugehen, dass der Feldpointer grundsätzlich falsch steht. Daher muss er vor seiner Verwendung mit einem `adrof`-Befehl *unbedingt* richtig eingestellt werden. Dies erfolgt typisch mit z. B. mit „`adrof f f(0)`“ unmittelbar nach dem `read`.

Zusammenfassung: Für ein Feld F der Länge 10 gilt ...

- Das Feld wird mit einem `_dim F 10` angelegt. Der höchste Index ist 10, das letzte Feldelement `F(10)`,
- Da die Zählung mit dem 0. Element beginnt, ist die Anzahl der tatsächlich vorhandenen Feldelemente um 1 höher. Unser Feld hat also tatsächlich 11 Elemente.
- Außerdem kommt auch noch der Feldpointer hinzu. In der Symboltabelle belegt das Feld also 12 Speicherplätze. Ebenso belegt das Feld in einer Datei 12 Zeilen.
- Weil der Längenparameter der Befehle „read“ und „write“ die Anzahl der tatsächlich übertragenen Werte um 1 vermindert angibt, wird das Feld mit einem `write F 11` geschrieben und einem `read F 11` gelesen.
- Nach dem „read“ einen `adrof F F(0)` keinesfalls vergessen!

7. Abarbeiten und Debuggen

Modi: 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.

Signale: Weiterhin gibt es zwei Signale, mit denen man das laufende Programm beeinflussen kann. Das *Halt*signal bewirkt eine Programmunterbrechung nach dem laufenden Befehl. Es kann vom Debugger aus oder mit dem Befehl `halt` gesetzt werden. — Das *Abbruch*signal bewirkt eine sofortige Programmbeendigung. Es wirkt wie der Befehl `exit` und kann auch mit dem Z-Interrupt (Eingabe von Ctrl/Z) ausgelöst werden.

8. Hinweise zum Programmieren

Datentyp: Der RTA-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 aus Betriebssystemgründen nur die Sonderzeichen `_`, `(`, `)` und `$` geduldet. Andere Zeichen werden in den Unterstrich umgewandelt. Großbuchstaben werden in Kleinbuchstaben umkodiert. — Um Schwierigkeiten zu vermeiden, sollte in diesbezüglichen Symbolnamen nur Buchstaben, Ziffern und die Sonderzeichen `_`, `(`, `)` und `$` verwendet werden. Großbuchstabige Symbolnamen sind zulässig, sofern nicht zeitgleich ein verwechselbares kleinbuchstabiges Symbol gespeichert wird – der Dateiname ist immer kleinbuchstabig.

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. Hier eine Prüfcheckliste:

- 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 beim read-Befehl in Felder. Hier soll der im Befehl angegebene Längenparameter die im `_dim` deklarierte Feldlänge + 1 sein. Ist der Wert größer, so wird die Symboltabelle über das Feldende hinaus beschrieben und damit zerstört.
- Nach dem „read“ (`read FELD n`) eines Feldes den „adrof“ (`adrof FELD FELD(0)`) nicht vergessen.

Anhang: Fehlercodes

100:	Allgemeiner Laufzeitfehler
101:	Überlauf. Betrag > 9E99
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
114:	Nichtexistierende Symboladresse
115:	Nichtexistierende Codeadresse
116:	Unbekannter Befehl
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. Die Befehle erzeugen aber Fehlercodes, die mit dem Befehl `errcode` gelesen oder auf den mit dem Befehl `errjump` reagiert werden kann. Der `errjump` rettet den Fehler des jeweils vorherigen Befehls, so dass dieser auch nach dem Sprung noch per `errcode` ermittelt werden kann.

Außerdem werden die jeweils letzten Laufzeitfehler eines jeden Befehls in einem Laufzeitprotokoll eingetragen und gezählt.