TELEFUNKEN
COMPUTER

BCPL
Sprachbeschreibung

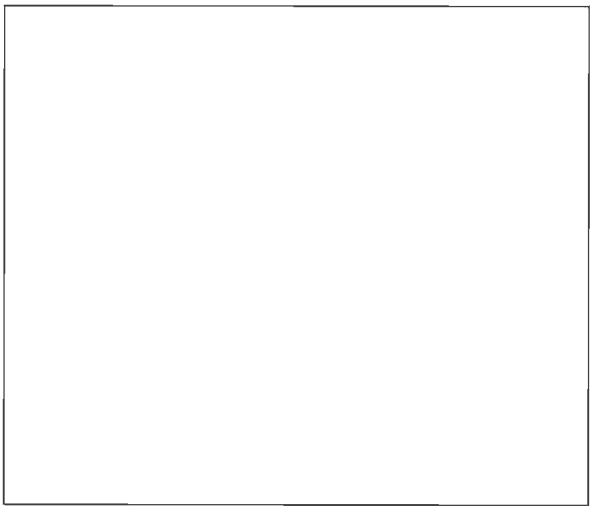
SYSTEM TR 440

SYSTEM TR 440

BCPL Sprachbeschreibung

Änderungsstand

1 März 74	2 Mai 77											



2. Nachtrag zur BCPL-Sprachbeschreibung

Lfd. Nr.	Datum	Abschnitt	Seite	Berichtigung
2	Mai 77	K		Gesamten Abschnitt K austauschen

EINFÜHRUNG

BCPL (Basic Combined Programming Language) ist ein Subset von CPL und wurde am MIT und den Universitäten Cambridge und London entwickelt. Die Sprache war ursprünglich zum Schreiben von Compilern konzipiert worden, sie eignet sich auch zur Lösung anderer nichtnumerischer Probleme.

A

BCPL läßt sich leicht lernen, die Programme sind gut lesbar und besitzen einen hohen Dokumentationswert. Der BCPL-Compiler selbst ist in BCPL geschrieben. Der Syntax- und Semantiklauf des Compilers wurden von Dr. Martin Richards von der Universität London geschrieben, diese Teile sind so konzipiert, daß sie sich leicht zu einer Codegenerierung erweitern und auf eine andere Maschine übertragen lassen.
Dieses Handbuch beschreibt den am TR 440 implementierten Sprachumfang (Abschnitt B - G).
Prozeduren für die Ein- oder Ausgabe sind in Abschnitt H beschrieben.

ALLGEMEINE FESTLEGUNGEN

ALLGEMEINE FESTLEGUNGEN

0. Beschreibung der Metasprache

Im vorliegenden Handbuch wurden bei der Erklärung der Begriffe und Anweisungen zwei Beschreibungsformen gewählt.
Wenn aus der Semantik oder Syntax eine Begriffserläuterung aufgrund von Beispielen oder Beschreibungen eindeutig zu verstehen ist, wurde auf die allgemeine Form einer metasprachlichen Beschreibung verzichtet.

B

Z.B. WHILE E DO C wird beschrieben in der Form

E ist ein beliebiger Ausdruck
C ist eine beliebige Anweisung

In den Fällen, wo einem Begriff mehrere Bedeutungen zukommen oder wo die oben genannten Erläuterungen nicht eindeutig sind, wurde die metasprachliche Beschreibung verwendet.

a) Werden Begriffe, Sonderzeichen oder Ausdrücke in geschweifte Klammern eingeschlossen, so ist genau ein Element auszuwählen.

Z.B. $\left\{\begin{matrix}\text{EQV}\\\text{NEQV}\end{matrix}\right\}$ oder $\left\{\begin{matrix}/\\+\\-\\*\end{matrix}\right\}$

b) Steht ein Ausdruck oder Begriff in eckigen Klammern, so ist seine Verwendung optional

Z.B. FOR N = E1 TO E2 [BY E3] DO
oder E [REP K]

c) Ein metasprachlicher Wiederholungsoperator ist

1) ein hochgestellter Index
2) eine Folge von Punkten

Der hochgestellte Index gibt an, wie oft der metasprachliche Ausdruck wiederholt werden kann. Das Zeichen ∞ bedeutet: beliebig oft wiederholbar.

Aufeinanderfolgende Punkte haben die Bedeutung einer möglichen Fortsetzung gleichartiger Elemente

Z.B. $"[\text{Alphazeichen}]_0^{255}"$

oder $E_{L_1}, E_{L_2}, \ldots E_{L_n} := E_{R_1}, E_{R_2}, \ldots E_{R_n}$

d) Gleichrangige, aber in der Bedeutung unterschiedliche Darstellungen, die einem Ausdruck oder Begriff zugeordnet werden können, werden durch einen senkrechten Strich getrennt:

Z.B. Zahl ::= 0|1|2|3|...|9|

⟨Ausdruck⟩ ::= ⟨einfacher Ausdruck⟩ | ⟨monadischer Ausdruck⟩ | ⟨dyadischer Ausdruck⟩ | ⟨bedingter Ausdruck⟩

e) Eine metasprachliche Variable ist eine durch ⟨ eingeleitete und durch ⟩ abgeschlossene Zeichenfolge, z.B. ⟨table⟩.

f) Eine metasprachliche Variable wird durch eine Deklaration folgender Form definiert: metasprachliche Variable ::= ...

1. Grundsymbole

Ein BCPL-Programm besteht aus einer Folge von syntaktischen Grundsymbolen und Kommentaren.

Grundsymbole:

- BCPL Schlüsselworte
- gewisse Sonderzeichen (-kombinationen)
- Namen
- Blockklammern
- Zahlen
- Zeichenkonstante
- Stringkonstante

Regel:

Für alle Grundsymbole (außer Strings) und Kommentare gilt das Karten- bzw. Zeilenende als Trenner.

2. Zeichenvorrat

Der Zeichenvorrat für die BCPL-Grundsymbole setzt sich aus folgenden Zeichenklassen zusammen:

- Buchstaben	::=	A|B|C|... |Y|Z
- Ziffern	::=	0|1| ... 8|9
- Sonderzeichen	::=	"|'|$|*| | |+|-|/|=|<|>|(|) |.|, |:|;|!|␣|&
- Alphazeichen	::=	alle Zeichen des Zentralcodes*
- Oktalziffer	::=	0|1|... |7
- Sedezimal-Ziffern	::=	0|1|... |9|A|B|C|D|E|F

*
Zentralcode siehe TR 440 GROSSE BEFEHLSLISTE

3. Benennungen

Definition:

Eine Benennung ist eine beliebige Folge von Buchstaben, Ziffern und "."

3.1. Namen

Definition:

Ein Name ist eine mit einem Buchstaben beginnende Benennung aus max. 50 Zeichen. Der Name darf nicht mit einem Schlüsselwort identisch sein.

Beispiel:

A, A.1, A.WERT.1, JGAM7, F3, S...B

3.2. Blockklammern

Definition:

Durch ein Paar öffnender und schließender Blockklammern wird eine Folge von Deklarationen (siehe E) und Anweisungen (siehe F) zu einem Block zusammengefaßt. Blockklammernpaare dienen ferner zur eindeutigen Begrenzung von Deklarationen.

öffnende Blockklammern

unbenannt: $(

benannt : $(⟨id⟩ *

schließende Blockklammern

unbenannt: $)

benannt : $)⟨id⟩ *

* ⟨id⟩ ist eine beliebige Benennung mit max. 50 Zeichen

Alternativdarstellungen für öffnende und schließende Blockklammern siehe O.

Regeln:

a) Wird eine Benennung ⟨id⟩ verwandt, darf zwischen ihr und der öffnenden Blockklammer (bzw. der schließenden) kein Blank stehen, da der Block sonst als unbenannt angesehen wird.
Umgekehrt müssen Namen und Zahlen von den Blockklammern mindestens durch einen Zwischenraum getrennt werden, da sie sonst als Blockbenennung aufgefaßt werden.

b) Zuordnung von öffnenden und schließenden Klammern:

b.1) Eine benannte Blockklammer schließt die vorhergehende gleichbenannte öffnende Blockklammer und alle noch öffnenden Blockklammern (benannt oder unbenannt), die zwischen dem benannten Paar stehen.

b.2) Eine unbenannte schließende BK schließt die vorhergehende öffnende BK, die unbenannt sein muß.

Merke:

Eine implizite Schließung von öffnenden Blockklammern kann also nur durch eine benannte Blockklammer erfolgen.

c) Eine einmal vergebene Blockbenennung kann erneut verwandt werden, wenn zum Zeitpunkt des Wiedergebrauchs jeder vorhergehende gleichbenannte Block bereits geschlossen ist.

Beispiele:

RICHTIG

```
a)    $(1   UNTIL I=10  DO
            $(2     TEST I=0  THEN WRITE(6,"KEIN ERGEBNIS", 0)
                    OR    $(3  SUM:= SUM+C/I
                               C:= -C     $)3
                                  I:= I+1    $)2    $)1
```

```
b)   $(1   UNTIL I=10  DO
           $(   TEST I=0  THEN  WRITE(6,"KEIN ERGEBNIS", 0)
                      OR $(      SUM:= SUM + C/I
                                 C:= -C      $)
                                      I:= I + 1    $)1
```

Beispiel a) und b) sind äquivalent

FALSCH

```
c)   $(    UNTIL I=10  DO
             $(1   TEST  I=0  THEN WRITE (6,"KEIN ERGEBNIS", 0)
                     OR    $(2    SUM:= SUM + C/I
                                  C:= -C      $)2
                                       I:= I + 1   $)
```

```
d)   $(␣1  I:= I + J
           J:= I       $)1
```

Beispiel c) ist falsch, weil die letzte schließende Klammer zunächst den Block mit der Benennung 1 schließen muß und demnach nicht unbenannt sein darf.
Beispiel d) ist falsch, da zwischen öffnender Klammer und Benennung ein Blank steht.

4. Weglassen von Semikolon

Zwischen zwei Anweisungen muß genau dann ein Semikolon stehen, wenn einerseits Anweisungsende und andererseits der folgende Anweisungsanfang nicht eindeutig aus dem Kontext hervorgehen. Es kann weggelassen werden, wenn die syntaktische Eindeutigkeit gewährleistet ist.
Im Zweifelsfall ist zwischen zwei Anweisungen ein Semikolon zu setzen.

Regel:

Mehrere Anweisungen sollten durch Semikolon getrennt werden.

Beispiele:

a) R ( ) (B→F,G) ( )

Diese Anweisung wird interpretiert wie:

(R ( ) (B→F,G)) ( )

Ergebnis ist ein Routineaufruf (siehe F 2).

R ( ); (B→F,G) ( )

Durch das Semikolon entstehen zwei neue Anweisungen (2 Routineaufrufe unter völlig anderen Bedingungen).

b) K: = 10 * I

ERG: = F (K) + 99

Semikolon kann entfallen

c) K: = 10 * I; ERG: = F (K) + 99

Semikolon kann entfallen

d) A: = B;

-C = 5→A1, A2: = 0

Hinter der 1. Anweisung muß ein Semikolon stehen, da sonst die syntaktisch falsche Anweisung

A: = B -C = 5→A1, A2: = 0

entsteht.

5. Kommentare

Man unterscheidet zwei Arten von Kommentarbegrenzungen.
Wird der Kommentar durch zwei aufeinanderfolgende "/" eingeleitet, dann gilt das Zeilenende als Kommentarende. Die zwischen Kommentaranfang und- ende stehende Zeichenfolge kann beliebig sein.
Wird der Kommentar durch die Zeichenfolge /* eingeleitet, dann muß der Kommentar durch die Zeichenfolge */ abgeschlossen werden.

Beispiele:

```
// HAUPT PROGRAMM STDHP
   EXTERNAL A
   GLOBAL  $(  START :1 $)
START: $(
         :
         :          $)
```

a)
```
AL(B, V!1, LV EQ.1)  // ROUTINE-Aufruf
                     // NAME = AL
A := Q
```

b)
```
LET R( ) BE  // DIESE ROUTINE BESETZT
             // DEN VEKTOR SYMB
$( FOR I=1 TO 100 DO
     READ (5, "I3", 1, LV SYMB!I) $)
```

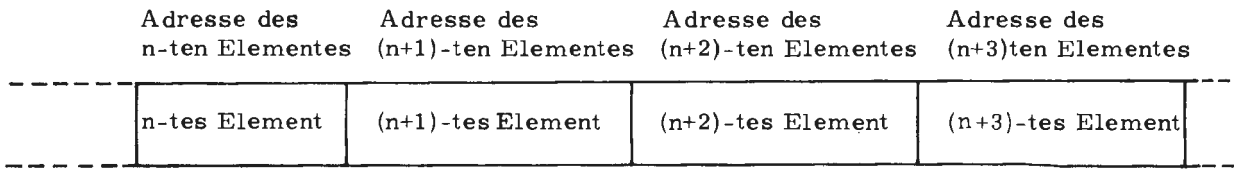
c)
```
LET R( ) BE  /* DIESE ROUTINE BESETZT
                DEN VEKTOR SYMB */
$( FOR I = 1 TO 100 DO
   READ (5, "I3", 1, LV SYMB! I) $)
```

BCPL-GRUNDBEGRIFFE

BCPL-GRUNDBEGRIFFE

1. Struktur der BCPL-Objekt-Maschine

Das wesentliche Merkmal der BCPL-Objekt-Maschine ist ihr Arbeitsspeicher. Er besteht aus einer Reihe von fortlaufend numerierten BCPL-Elementen. Jedes BCPL-Element entspricht einer Speicherzelle, die linear so angeordnet sind, daß sich die BCPL- Adressen aufeinanderfolgender BCPL-Elemente um eins unterscheiden.



Arbeitsspeicher der BCPL-Objekt-Maschine

Implementierung TR 440: Jedes BCPL-Element entspricht einem TR 440-Halbwort von der Länge 24 Bits

Jedes BCPL-Element läßt sich durch zwei Begriffe beschreiben, den L-Wert und den R-Wert.
Der L-Wert eines BCPL-Elementes entspricht der Adresse, der R-Wert dem Inhalt der Speicherzelle. Näheres siehe C 6.2 und C 6.3

2. Datentypen

Bei Konstanten, Variablen und Ausdrücken wird unterschieden zwischen:

a) dem sprachinternen Typ
b) dem Bedeutungstyp

Es gibt in BCPL nur einen internen Typ, eben den R-Wert, im Gegensatz zu ALGOL, FORTRAN etc... Der Compiler läßt also z.B. zu, daß mit einer Variablen, die mit dem logischen Wert TRUE initialisiert ist, multipliziert wird.

Der Programmierer selbst ordnet auf Grund der Operationen die er mit R-Werten durchführt, dem Bitfeld eine ganz bestimmte Bedeutung zu.
Bei allen Operationen sollte beachtet werden, daß der für einen R-Wert gewählte Bedeutungstyp mit der Operationsart vereinbar ist.
Bei arithmetischen Operationen werden die R-Werte der Operanden als ganze Zahlen aufgefaßt.

Die Vorteile einer Sprache ohne unterschiedliche Datentypen sind:

- Es sind keine Typen-Deklarationen notwendig. Da Typ-Prüfungen entfallen, vereinfacht sich die Behandlung von aktuellen Parametern bei Prozeduraufrufen, sowie von EXTERNAL, - und GLOBAL-Variablen, die in verschiedenen, getrennt übersetzten Quellen vorkommen.

- Da alle Bedeutungstypen auf denselben sprachinternen Typ abgebildet werden, hat man die Vorteile einer Sprache mit dynamisch veränderlichen Typen, ohne daß dadurch das Programm belastet wird. Ferner können beliebige Strukturen aufgebaut werden; z.B. kann ein (Teil-) Vektor nebeneinander enthalten: Ganze Zahlen, Zeiger auf weitere Strukturen.

Da eine Typ-Prüfung durch den Compiler entfällt, ist es allerdings möglich, unsinnige Programme zu schreiben, die der Compiler nicht beanstandet.

Implementierung TR440: Die interne Darstellung der Zahlen ist implementierungsabhängig. Deshalb ist es meistens nicht sinnvoll, arithmetische Operanden mit nichtnumerischen Operationen zu verarbeiten oder Bitmanipulationen vorzunehmen.

3. Variable

Eine Variable ist eine Zuordnung eines BCPL-Elementes zu einem Namen. Jede Variable hat einen Wert. Dieser Wert ist der in der Zelle enthaltene R-Wert, der während des Programmlaufs durch Zuweisungen an die Variable verändert werden kann.

In BCPL werden Variable entweder explizit oder implizit deklariert. Die explizite Variablendeklaration muß in einem eigens für Deklarationen vorgesehenen Deklarationsteil erfolgen.

4. MANIFEST-Konstante

Über eine MANIFEST-Deklaration werden Namen deklariert, denen in der Deklaration ein konstanter Wert zugeordnet wird.
Diese einmalig vergebene, feste Zuordnung eines R-Wertes zu einem Namen gilt für den gesamten Programmlauf.
Da bereits zur Compile-Zeit die über eine MANIFEST-Deklaration vereinbarten Namen durch den ihnen zugeordneten konstanten Wert ersetzt werden, dürfen im Programm an MANIFEST-Konstante keine Wertzuweisungen erfolgen.
Durch die Verwendung von MANIFEST-Konstanten wird die Selbstdokumentation von Programmen erhöht (siehe E 2.), ohne die Objektlaufzeit zu erhöhen.

5. Vektoren

Die Bearbeitung eines zusammenhängenden Speicherbereichs, der aus mehreren aufeinanderfolgenden BCPL-Elementen besteht, kann durch die Einführung des Vektorbegriffes und eines Vektoroperators vereinfacht werden.

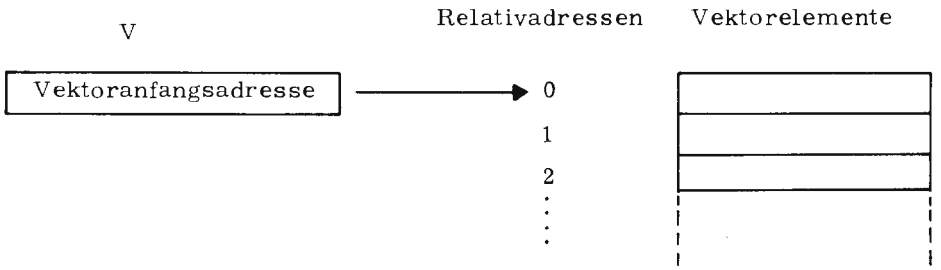
Definition eines Vektors

Ein Vektor ist ein Speicherbereich, dessen aufeinanderfolgende BCPL-Elemente eine BCPL-Adressendifferenz von 1 aufweisen. Die Vektoranfangsadresse liegt in einem weiteren BCPL-Element; über sie kann auf den Vektor und seine einzelnen Elemente zugegriffen werden.

Implementierung TR 440: Jedes Vektor-Element entspricht einem TR 440-Halbwort. Die Adressendifferenz zweier aufeinanderfolgender Vektorelemente ist deshalb 1.

5.1. Zugriff auf Vektorelemente über Vektoroperator

Es sei der R-Wert einer Variablen V gleich der Adresse (L-Wert), die auf das erste der aufeinanderfolgenden Vektorelemente weist. Alle folgenden Vektorelemente können über Relativadressen, bezogen auf den Vektoranfang, angesprochen werden.



Der Zugriff auf die einzelnen Vektorelemente erfolgt über den Vektoroperator " ! ".

Allgemeine Form:

⟨exp 1⟩ !⟨exp 2⟩

Bezeichnung:

⟨exp 1⟩ und ⟨exp 2⟩ sind einfache Ausdrücke. Dies kann immer erreicht werden, indem die Ausdrücke geklammert werden (siehe G 4.6).

Wirkung:

Voraussetzung:

R-Wert von V → L-Wert der 1. Speicherzelle
(Relativadresse 0)

Dann gilt:

R-Wert von V ! 0 → R-Wert (Inhalt) der 1. Speicherzelle
(Relativadresse 0)

R-Wert von V ! 1 → R-Wert der 2. Speicherzelle
(Relativadresse 1)

R-Wert von V ! 2 → R-Wert der 3. Speicherzelle
(Relativadresse 2)

⋮

R-Wert von V ! n → R-Wert der (n+1)-ten Speicherzelle
(Relativadresse n)

Entsprechend gilt:

L-Wert von V ! 0 → L-Wert (Adresse) der 1. Speicherzelle
(= V)

L-Wert von V ! 1 → L-Wert der 2. Speicherzelle

⋮

Merke:

Da die Relativadressierung bei "0" beginnt entspricht der Inhalt (R-Wert) von V ! n dem Inhalt des (n+1)-ten Vektorelements.

Beispiele:

a) V ! (I+2) := V ! I + V ! (I+1)

Die rechte Seite wird im R-Modus berechnet; die R-Werte der Vektorelemente V ! I und V ! (I+1) werden addiert. Das Ergebnis (ein R-Wert) ersetzt den Inhalt jener Speicherzelle, deren Adresse der L-Wert V ! (I+2) ist (siehe C 6).

6. Berechnungsmodi bei einfachen Zuweisungen

6.1. Allgemeine Beschreibung

⟨exp 1⟩: = ⟨exp 2⟩

⟨exp 1⟩ und ⟨exp 2⟩ sind Ausdrücke

Die Wirkung einer einfachen Zuweisung beruht auf der Berechnung des rechts vom Zuweisungszeichen stehenden Ausdrucks und der Abspeicherung des Ergebnisses in eine Speicherzelle, deren Adresse durch den Ausdruck links des Zuweisungszeichens eindeutig gekennzeichnet sein muß.
Die Berechnungsmodi der Ausdrücke ⟨exp 1⟩, ⟨exp 2⟩ und speziell ihre einzelnen Operanden werden durch folgende Begriffe beschrieben.

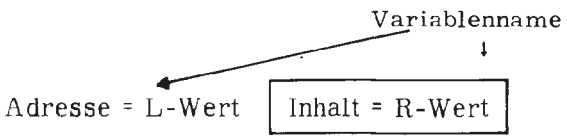
6.2. R-Wert

Ein BCPL-Element ist durch seine Adresse (L-Wert) und seinen Inhalt (R-Wert) eindeutig beschrieben.
Durch R-Werte werden u. a. Zahlen, Strings und die Wahrheitswerte "TRUE" und "FALSE" dargestellt.
Eine ganz spezielle Bedeutung kommt dem R-Wert bei der indirekten Adressierung zu. In diesem Fall ist der Wert eines BCPL-Elementes die Adresse eines weiteren BCPL-Elementes. Für diese Verweis- bzw. Ersetzungsmanipulationen sind im BCPL-Sprachumfang zwei spezielle Operatoren vorhanden (RV-Operator siehe C 6.8, LV-Operator siehe C 6.7).

Implementierung TR 440: Da gemäß Implementierung jedes BCPL-Element einem TR 440-Halbwort entspricht, ist der R-Wert ein Bitfeld von 24 Bits

6.3. L-Wert

Jedes BCPL-Element (Speicherzelle) besitzt eine ganzzahlige Adresse, die L-Wert genannt wird. Da einer Variablen eine Speicherzelle zugeordnet ist, besitzt sie einen L-Wert.
Eine Variable kann also wie folgt dargestellt werden.



Implementierung TR 440: Rein formal wird ein L-Wert durch ein Bitmuster von 24 Bits dargestellt und kann deshalb auch als R-Wert dargestellt werden.

Beispiele:

a) R-Wert der Variablen A → Inhalt der mit A bezeichneten Speicherzelle
   L-Wert der Variablen A → Adresse der mit A bezeichneten Speicherzelle

b) R-Wert der Konstanten 10 → Konstante 10 selbst (siehe G 8).
   L-Wert der Konstanten 10 → gibt es keinen

c) R-Wert von "TRUE" → Wahrheitswert "TRUE" selbst (siehe G 8).
   L-Wert von "TRUE" → gibt es keinen

C

6.4. R-Ausdrücke, L-Ausdrücke

Jedem Ausdruck, gleich welchen Typs, kann ein R-Wert zugeordnet werden; sie heißen R-Ausdrücke.
Zur Berechnung des R-Wertes von Ausdrücken werden die geforderten Operationen mit den R-Werten der einzelnen Operanden ausgeführt.
Einige Ausdrucks- Arten besitzen zusätzlich einen L-Wert; sie heißen L-Ausdrücke.
Einem L-Ausdruck ist ein über seinen L-Wert adressierbares BCPL-Element zugeordnet, das seinen R-Wert enthält.

Bei der Ausführung einfacher Zuweisungen ist auf der linken Seite des Zuweisungszeichens die Existenz einer, für die Abspeicherung erforderlichen Speicherzellenadresse, unumgängliche Voraussetzung.
Deshalb dürfen auf dieser Position nur L-Ausdrücke stehen, womit die Existenz einer Adresse (L-Wert des Ausdruckes) in jedem Fall gewährleistet ist.

Die Bildungsvorschrift für den L-Wert eines L-Ausdruckes ist der Einzelbeschreibung der speziellen Ausdrucksart zu entnehmen.

Merke:

L-Ausdrücke besitzen immer einen R-Wert, da über den zugehörigen L-Wert auch auf den Inhalt der Speicherzelle zugegriffen werden kann.
R-Ausdrücke können einen L-Wert besitzen. Wo dies der Fall ist, ist es, zusammen mit der Bildungsvorschrift, bei den einzelnen Ausdrucksarten angegeben.

Beispiel:

a) V ! 1 : = x

Die Forderung nach einem L-Ausdruck ist erfüllt, da jedes Vektorelement einen L-Wert besitzt.
Der L-Wert wird gebildet (im Beispiel) aus dem Inhalt von V (V muß eine Zeigervariable sein), der um einen Adreßschritt erhöht wird. Auf die so gebildete Adresse wird gespeichert.

b) Es ist darauf zu achten, daß dem L-Wert des L-Ausdruckes auch eine Speicherzelle zugeordnet ist. (Bei indirektem Zugriff, z. B. wird bei 3 ! 0 : = 1 versucht, auf die Absolut-Adresse 3 zu speichern)

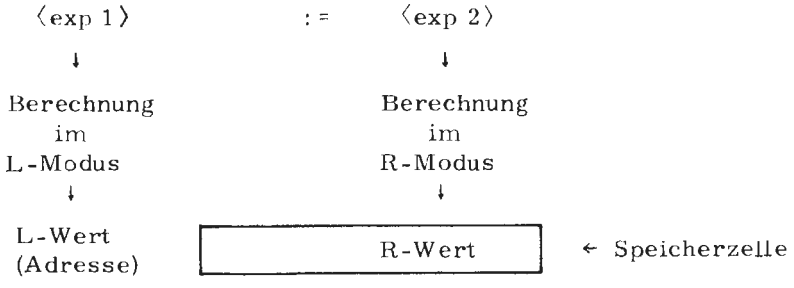
6.5. L-Modus, R-Modus

Bei einer einfachen Zuweisung von der Form:

⟨exp 1⟩ : = ⟨exp 2⟩ ⟨exp 1⟩ und ⟨exp 2⟩ sind Ausdrücke

werden für die Ausdrücke ⟨exp 1⟩ und ⟨exp 2⟩ grundsätzlich zwei verschiedene Berechnungsmodi angewandt.
Der Ausdruck ⟨exp 1⟩ wird im L-Modus (Adressenmodus) berechnet. Das Ergebnis ist der L-Wert von ⟨exp 1⟩. Eine Berechnung im L-Modus setzt als Berechnungsobjekt stets einen L-Ausdruck voraus.
Der Ausdruck ⟨exp 2⟩ wird im R-Modus berechnet. Die geforderten Operationen werden mit den R-Werten der einzelnen Operanden ausgeführt.
Das Ergebnis ist der R-Wert von ⟨exp 2⟩.

Für die einfache Zuweisung gilt infolge dessen:
Der R-Wert von ⟨exp 2⟩ ersetzt den Inhalt der durch den L-Wert von ⟨exp 1⟩ adressierten Speicherzelle.



Beispiele:

a) V ! 1 := X X := V ! 1

Der Ausdruck V ! 1 wird in Abhängigkeit seiner Position bezüglich des Zuweisungszeichen nach zwei verschiedenen Berechnungsmodi interpretiert. Auf der linken Seite wird er im L-Modus berechnet. Ergebnis ist der L-Wert des Vektorelementes V ! 1. Auf der rechten Seite wird er im R-Modus berechnet; Ergebnis ist der R-Wert des Vektorelementes V ! 1. Das gleiche gilt für die einfache Variable X.

b) Der Ausdruck

A LOGOR B (A und B sind Operanden des logischen Operators LOGOR)

kann nur im R-Modus berechnet werden, da logische Ausdrücke keinen L-Wert besitzen (siehe G 6.6).
Deshalb darf dieser Ausdruck nie auf der linken Seite einer Zuweisung stehen.

C

6.6. L-Position, R-Position

Entsprechend dem anzuwendenden Berechnungsmodus werden für Operanden L-Positionen und R-Positionen unterschieden.
Auf einer L-Position dürfen nur L-Ausdrücke stehen. Ein Operand, der auf L-Position steht, ist im L-Modus zu berechnen.
Bei einfachen Zuweisungen steht der Ausdruck links des Zuweisungszeichens auf L-Position, der Ausdruck rechts des Zuweisungszeichens steht auf R-Position.
Die Bestimmung und Bedeutung des Positionstyps (L-Position oder R-Position) ist speziell bei der Wirkung der folgenden zwei Operatoren zu beachten.

6.7. LV-Operator

R-Werte von BCPL-Elementen können insbesondere Adressen sein. Um Adreßmanipulationen vornehmen zu können, muß die Möglichkeit eines expliziten Zugriffs auf Adressen bestehen.
Der LV-Operator macht den L-Wert (die Adresse) einer Speicherzelle zugänglich.

Syntax:

LV ⟨exp⟩

⟨exp⟩ muß ein L-Ausdruck sein.

Wirkung:

⟨exp⟩ steht auf L-Position, d.h.

⟨exp⟩ wird im L-Modus berechnet, muß also ein L-Ausdruck sein. Das Ergebnis (R-Wert) der Operation ist eine Adresse, der L-Wert von ⟨exp⟩. Der LV-Operator bewirkt also, daß der nachfolgende Operand im L-Modus (d.h. dessen Adresse) berechnet wird.

LV-Ausdrücke sind R-Ausdrücke, die keinen L-Wert besitzen.

Regel:

Außer in LV-Ausdrücken und auf den linken Seiten von Zuweisungen werden Ausdrücke immer im R-Modus berechnet.

Beispiel:

```
X := LV V ! 10
        ↓
     L-Ausdruck (L-Position)
     Berechnung
     im
     L-Modus
        ↓
Ergebnis: L-Wert (Adresse)
          von
          V ! 10
```

6.8. RV-Operator

Der RV-Operator ermöglicht die Manipulation von Vektoren und Strukturen. Er bewirkt in jedem der Bearbeitungsmodi eine einfache Ersetzung.

Syntax:

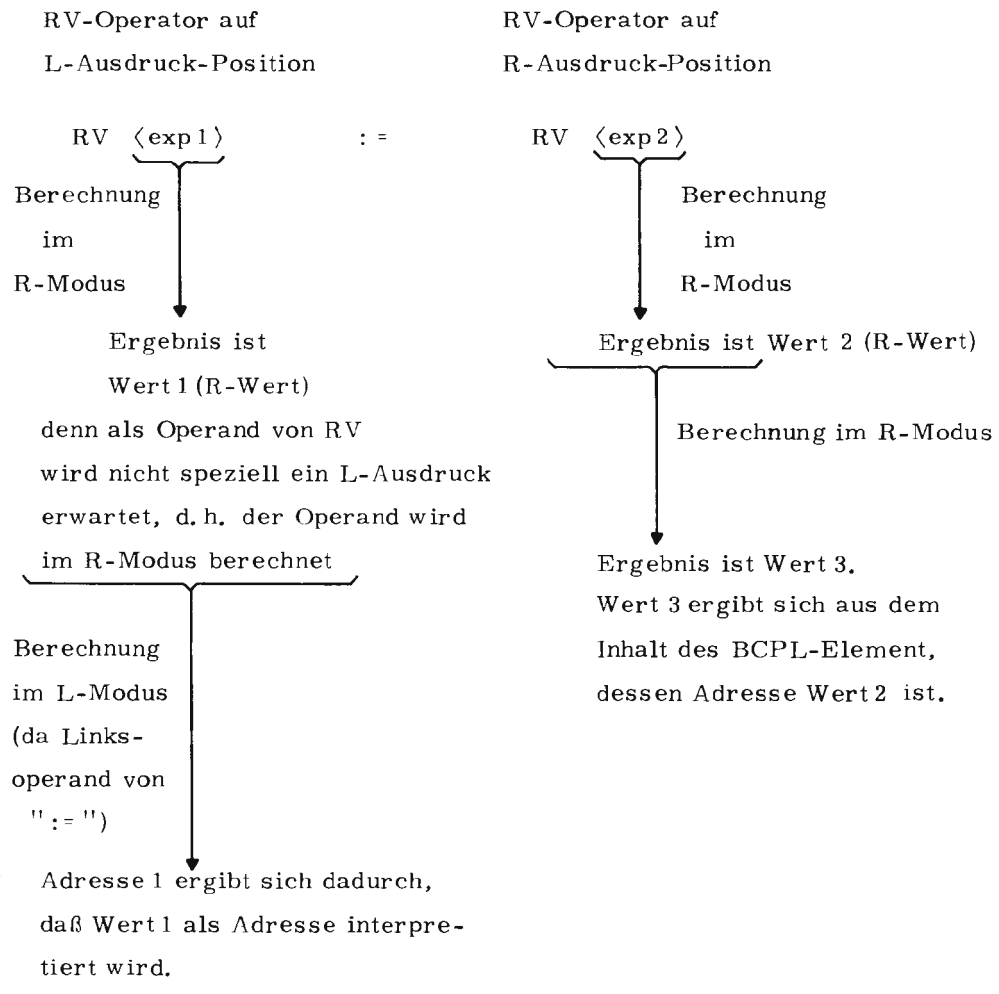
RV ⟨exp⟩

Wirkung:

RV ⟨exp⟩ ist ein L-Ausdruck, d. h. er besitzt eine Adresse oder L-Wert. Zur Berechnung des L-Wertes (L-Modus) wird der R-Wert von ⟨exp⟩ berechnet. Dieser R-Wert wird als Adresse aufgefaßt.

Den R-Wert von RV ⟨exp⟩ erhält man, indem man den R-Wert von ⟨exp⟩ als L-Wert eines BCPL-Elements auffaßt; dessen Inhalt (R-Wert) ist der Wert des RV-Ausdrucks.

Beispiele:



Nach der Zuweisung enthält die mit Adresse 1 bezeichnete Speicherzelle Wert 3.

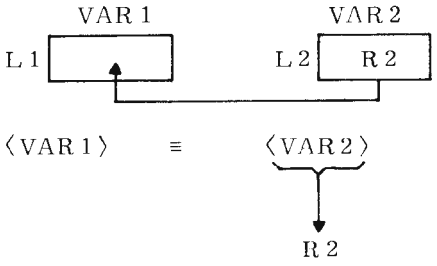
6. 9. Beispiele für die Berechnungsmodi bei einfachen Zuweisungen

Zur Erläuterung der Beispiele dienen folgende Erklärungen:

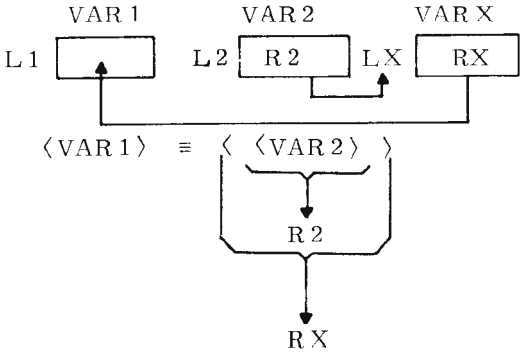
Variablenname	:	VAR 1, VAR 2, VAR X, VAR Y
zugehöriger R-Wert (Inhalt)	:	R 1, R 2, R X, R Y
zugehöriger L-Wert (Adresse)	:	L 1, L 2, L X, L Y
→	:	Anfangs- und Endpunkt des Pfeiles weisen auf identische Bitmuster; *)
⟨VARn⟩	:	Inhalt von VARn (R-Wert)
⟩VARn⟨	:	Adresse von VARn (L-Wert)

*)eine auf eine Adresse deutende Pfeilspitze weist diese Adresse als Inhalt (R-Wert) des zugehörigen Ausgangselementes aus.

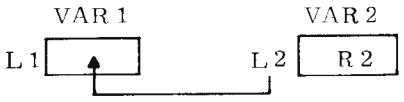
a) VAR 1 := VAR 2



b) VAR 1 := RV VAR 2

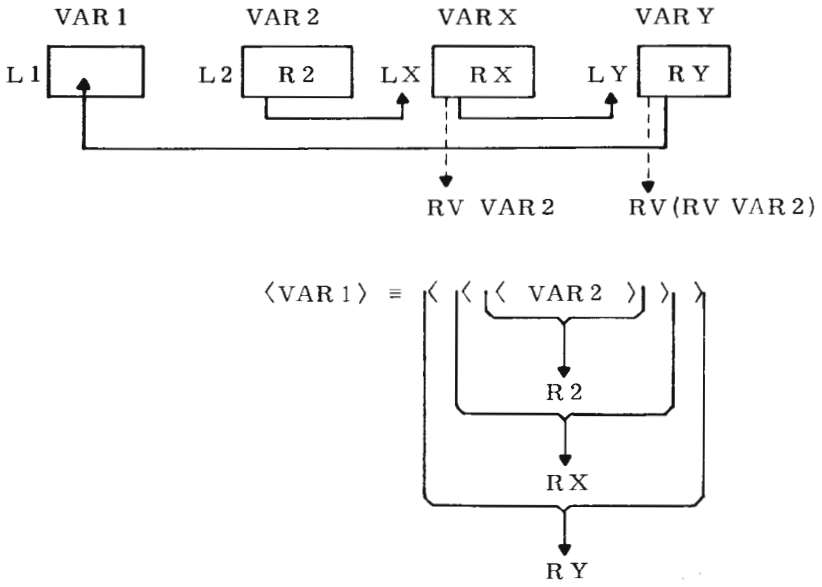


c) VAR 1 := LV VAR 2

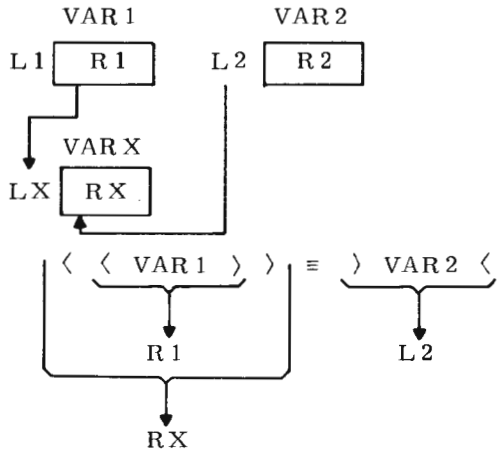


⟨VAR 1⟩ ≡ ⟩VAR 2⟨

d) VAR 1 := RV (RV VAR 2)



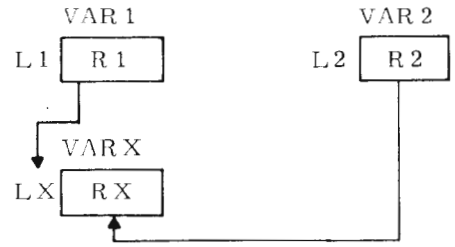
e) RV VAR 1 := LV VAR 2

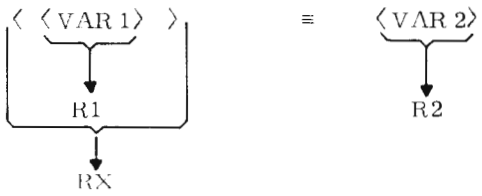


f) LV VAR 1 := VAR 2

Diese Anwendung des LV-Operators ist unzulässig, denn die Adresse von VAR 1 besitzt keinen L-Wert.

g) RV VAR 1 := VAR 2





AUFBAU EINES BCPL-PROGRAMMS

D

AUFBAU EINES BCPL-PROGRAMMES

1. Allgemeiner Aufbau

Ein BCPL-Programm ist gekennzeichnet durch eine algolähnliche Blockstruktur und hat folgenden Aufbau:

⟨Deklarationsteil⟩
⟨Anweisungsteil⟩

- Blocköffnendes Element ("$(") und blockschließendes Element ("$)") sind hier optional und dürfen benannt oder unbenannt sein.

Im Deklarationsteil werden Variable und Namen deklariert, deren Gültigkeitsbereich und Existenzdauer sich auf den gesamten folgenden Anweisungsteil erstrecken.
Im Anweisungsteil kann jede Anweisung wieder ein Block sein von der Form:

$(⟨Deklarationsteil⟩
⟨Anweisungsteil⟩ $)

- Blocköffnendes Element ("$(") und blockschließendes Element ("$)") müssen vorhanden sein (nicht optional)
- Der Deklarationsteil von Blöcken kann leer sein.

Da innerhalb eines Blockes jede Anweisung wieder ein Block sein kann, können Blöcke beliebig verschachtelt sein.
Enthält ein Block Deklarationen, ist die Gültigkeit der darin vereinbarten Namen und Variablen auf diesen Block und alle eventuell vorhandenen Unterblöcke beschränkt.

Ein BCPL-Programm kann aus mehreren Teilprogrammen (Montageobjekte) bestehen, die bei einer Montage zu einem startfähigen Programm (Operator) zusammenmontiert werden.

In genau einem der Teilprogramme muß eine Startmarke vereinbart sein.
Die Programmausführung beginnt bei der Anweisung, die im (Teil-) Programm mit dieser Marke markiert ist.
Das Ende der Programmausführung wird durch die dynamisch zuerst durchlaufene FINISH-Anweisung (siehe F 12.) oder das Erreichen des statischen Programmendes erreicht.

Implementierung TR 440: Die Start-Marke muß mit der Global-Adresse 1 vereinbart sein.

2. Blöcke

Syntax:

```
$(  ⟨Deklarationsteil⟩ ;
    ⟨Anweisungsteil⟩  $)

⟨Deklarationsteil⟩  ::= leer | ⟨Deklaration⟩ |
                        ⟨Deklarationsteil⟩ ⟨Deklaration⟩
⟨Anweisungsteil⟩    ::= leer | ⟨Anweisung⟩ |
                        ⟨Anweisungsteil⟩ [;] ⟨Anweisung⟩
```

- Deklarations- oder Anweisungsteil können in Abhängigkeit der Blockverwendung entfallen.
- Die Semikolons können im Allgemeinen entfallen (siehe Ξ 4).
- Äquivalente Darstellungen für Blockanfangs- und Blockendeklammern (siehe O).

Beschreibung

Die Blockausführung beginnt bei Existenz eines Deklarationsteils zunächst mit der Abarbeitung der Deklarationen. Alle an dieser Stelle vereinbarten Namen und Variablen sind lokale Größen und können deshalb nur in diesem Block verwandt werden. Derselbe Name darf innerhalb ein und derselben Deklaration nur einmal definiert werden.

Da bei einer Vereinbarung eines Namens die Gültigkeit von Deklarationen deselben Namens in übergeordneten Blöcken außer Kraft gesetzt wird, ist es möglich, innerhalb einer Blockverschachtelung Namen mehrfach für verschiedene Zwecke zu vergeben. Bei Verlassen des Blockes verliert die lokale Größe ihre Gültigkeit und die gleichnamige Variable des übergeordneten Blockes tritt wieder in Kraft.

Nach den Deklarationen wird die für diesen Block geltende Anweisungsfolge durchlaufen. Bestimmte Blöcke brauchen nicht am statischen Anfang betreten zu werden. Dies ist der Fall, wenn in Blöcken mit leerem (!) Deklarationsteil (Block darf nicht Bestandteil einer FOR-Schleife, Routine Definition oder eines VALOF-Blocks sein) Marken vereinbart werden. In diesem Fall sind Sprünge auf diese Marken aus übergeordneten Blöcken mit leerem Deklarationsteil oder Sprünge aus dem nächst umgebenden Block mit nichtleerem Deklarationsteil erlaubt.

Blöcke mit leerem Deklarationsteil werden i. A. dazu verwandt, um die Zugehörigkeit einer Anweisungsfolge zu bestimmten Sprachelementen zu kennzeichnen.

Beispiel:

Vorgegebene Syntax für die IF-Anweisung:

```
IF  E  DO
    C                    C ist eine Anweisung
```

Ist an dieser Stelle einer Anweisung eine ganze Anweisungsfolge erforderlich, muß sie als Block gekennzeichnet sein.

```
IF  E  DO
$(  ⟨Anweisung⟩
      .
      .
      .
            $)
```

3. Existenzdauer und Gültigkeitsbereiche von Namen

Jeder in einem BCPL-Programm vorkommende Name muß deklariert werden. Art und Zeitpunkt einer Deklaration bestimmen auch die Existenzdauer der deklarierten Namen.
Die Existenzdauer eines Namens ist die Zeit, während der einem Namen eine feste Speicherzelle zugeordnet ist.
Der Gültigkeitsbereich eines Namens ist der Teil des Programms, in dem über den Namen auf den R-Wert und gegebenenfalls den L-Wert der zugeordneten Speicherzelle zugegriffen werden kann.
Existenzdauer und Gültigkeitsbereich eines Namens können sich durchaus unterscheiden.
So kann zu einem definierten Zeitpunkt zwar die einem Namen zugeordnete Speicherzelle existieren, ein Zugriff auf diese Speicherzelle aber nicht möglich sein. Dieser Fall tritt z. B. ein, wenn im Deklarationsteil eines Blocks ein Name deklariert wird, der bereits in einem übergeordneten Block existiert.
Während der Ausführungsphase des untergeordneten Blocks ist die Gültigkeit der in einem übergeordneten Block deklarierten, gleichnamigen Größen außer Kraft gesetzt.

Namen können als Variable oder Konstante (MANIFEST-Konstante siehe **E** 2). deklariert werden. Bei Variablen wird zwischen einer expliziten und einer impliziten Deklaration unterschieden. Folgende Variablennamen werden implizit deklariert:

- formale Parameter bei Funktions- und Routinedefinition
- Marken
- Lauf-Variable bei FOR-Schleifen

Alle anderen Namen werden explizit deklariert.

4. Typ und Ablagebereiche von Variablen

4.1. Ablagebereiche

Während der Programmausführung können Variable in drei verschiedenen Speicherbereichen angeordnet sein.

a) im Globalvektor (Reihenfolge festgelegt)
b) im Arbeitsspeicher
c) in statischen Speicherzellen (Reihenfolge nicht festgelegt)

Der Globalvektor entspricht einem COMMON-Block in FORTRAN. Die einzelnen Speicherzellen eines Globalvektors können von allen Montageobjekten eines Programms gemeinsam benutzt werden. Sie dienen daher dem Informationsaustausch zwischen verschiedenen Montageobjekten. Bestimmten Zellen (-nummern) werden Namen zugeordnet. Die so definierten Variablen können mit den R-Werten (Einsprungadressen) von Routinen und Funktionen initialisiert werden.

Der Arbeitsspeicher wird für den dynamisch anfallenden Speicherbedarf benötigt. Er enthält die dynamisch angelegten BCPL-Elemente.

4.2. Dynamische Variable

Dynamische Variable werden deklariert durch:

- LET-Deklaration
- FOR-Anweisung (Laufvariable)
- formale Parameter einer Funktions- oder Routinedefinition.

Dynamische Variable existieren vom Zeitpunkt ihrer Deklaration bis zum dynamischen Verlassen des Blockes (einschließlich Unterblöcken), indem sie deklariert werden. Bei Prozeduraufrufen gilt: wird eine Prozedur rekursiv aufgerufen, dann wird bei jedem Aufruf eine neue Generation ihrer dynamischen Variablen angelegt. Dadurch lassen sich dynamische Variable zur komfortablen Kellerung von Information benutzen. Dynamische Variablen liegen stets in Stack.

Z.B.
```
LET F(A) BE
$( ... F(A+1) ... $)
:
:
F(1)
```

Der jeweilige Parameter A gibt in jeder Generation die aktuelle Verschachtelungstiefe der Aufrufe von F an.

4.3. Statische Variable

Statische Variable werden deklariert durch:

- Funktions- oder Routinedeklarationen
- STATIC-Deklarationen
- GLOBAL-Deklarationen
- Marken, die im Programm gesetzt werden.

Statische Variable existieren während des gesamten Programmlaufs.
Vor Beginn der Programmausführung wird einer statischen Variablen eine Speicherzelle zugeordnet, die erst bei Programmende wieder aufgegeben wird.

DEKALRATIONEN

DEKLARATIONEN

1. LET-Deklarationen

Syntax:

LET D [AND D]$_0^\infty$.

D bedeutet jeweils eine Variablendeklaration.

Beschreibung:

Eine LET-Deklaration kann in der Deklarationsfolge eines Blockes auftreten. Sie wird benutzt um einfache Variable, Vektoren, Funktionen und Routinen zu definieren.

Der Geltungsbereich dieser Variablendeklarationen erstreckt sich auf die LET-Deklaration selbst und auf alle folgenden Deklarationen und Anweisungen, die zum Block gehören.

Die durch die LET-Deklaration definierten statischen Variablen (Funktions- und Routinevariablen) werden bereits vor dem Programmlauf initialisiert, so daß LET-Deklarationen zur Definition wechselseitig rekursiver Routinen und Funktionen benutzt werden können.

Die Initialisierung dynamischer Variablen erfolgt in der Definition beim dynamischen Erreichen der Deklaration.

1.1. Deklaration einfacher Variablen

Syntax:

N1 N2, ..., Nn = E1, E2, ... En

N1, ... Nn sind untereinander verschiedene Namen

E1, ... En sind Ausdrücke oder das Schlüsselwort NIL. Die Liste darf auch Replikatoren enthalten (vgl. G 10).

Beschreibung:

Es werden dynamische Variable mit den Namen N1, ... Nn deklariert. Ihnen wird bei Abarbeitung der LET-Deklaration zunächst je eine Speicherzelle zugeordnet, die anschließend für jede Variable mit dem ihr zugeordneten Wert von E_i initialisiert wird.

Ist E_i das Schlüsselwort NIL, wird die Variable nicht initialisiert. Ihr Wert ist zum Zeitpunkt der Deklaration undefiniert. Erst durch eine Wertzuweisung im Anweisungsteil wird ihr Wert definiert.

Beispiele:

```
LET  A = 10
AND  B,C,D, = NIL, TRUE, $8772231
```

Der Variablen B ist zwar eine Speicherzelle zugeordnet, ihr Wert aber ist undefiniert. Die Variable C wird mit der logischen Größe TRUE (jedes Bit gesetzt), und die Variable D mit der angegebenen Oktalzahl initialisiert.

1.2. Vektor-Definitionen

Syntax:

```
N = VEC  K
```

N ist ein Name

K ist eine Konstante

Beschreibung:

Mit der Vektordefinition werden Namen zu Vektornamen erklärt und die max. Länge des Vektors bestimmt.

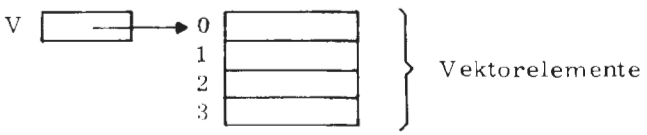
Die dynamische Variable N (Vektorname) wird mit der Anfangsadresse des Vektorspeichers initialisiert. Die Länge des Vektors ist durch den Wert der Konstanten K in der Vektordefinition festgelegt. Ihr Wert K wird zur Compilezeit berechnet. Er wird als ganze Zahl interpretiert und muß ≥0 sein.

Die Anzahl der Vektorelemente (je ein BCPL-Element) beträgt K + 1, da die Relativadressierung für die einzelnen Elemente mit '0' beginnt.

Eine Vektordefinition von der Form:

```
LET  V = VEC  3
```

wird wie folgt interpretiert.



Während bei einer Vektordefinition die Variable N mit der Anfangsadresse der Vektorelemente initialisiert wird, bleiben die R-Werte der einzelnen Vektorelemente undefiniert.

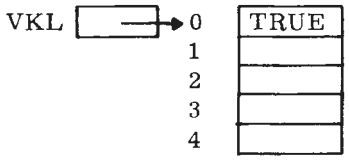
Die Vektorvariable und die Vektorelemente sind dynamische Größen. Wird also der Geltungsbereich von N verlassen, werden auch die Vektorelemente aufgegeben.

Auf die Elemente des Vektors kann z. B. mit Hilfe des Vektoroperators (siehe C 5.1) zugegriffen werden. V ! 0, V ! 1 ... bezeichnen die Vektorelemente mit den Relativadressen 0, 1 usw.

Implementierung TR 440: Jedes Vektorelement belegt ein TR 440-Halbwort, Vektoren werden ab einer Ganzwortadresse abgelegt

Beispiel:

a) LET VKL = VEC 4
VKL ! 0 := TRUE



b) LET V1 = VEC 10 AND V2 = V1 ! 3

Die Initialisierung von V2 erfolgt mit dem R-Wert des Vektorelementes V1 ! 3. Da das Vektorelement V1 ! 3 zu diesem Zeitpunkt noch undefiniert ist, wird auch der R-Wert von V2 auf undefiniert gesetzt.

1.3. Funktions- und Routine-Definitionen

Syntax:

⟨Funktions Definition⟩ ::= N(N1, N2, ... Nn) = E
⟨Routine Definition⟩ ::= N(N1, N2, ... Nn) BE C

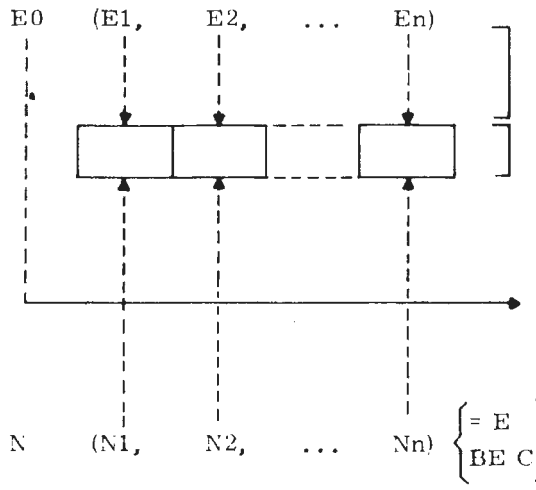
N, N1, N2, ..., Nn sind Namen
E ist ein Ausdruck
C ist eine Anweisung
Die Liste der Namen heißt formale Parameterliste. Sie kann leer sein.
E bzw. C heißen der Rumpf der Funktion bzw. der Routine.

Beschreibung:

Durch eine Funktions- oder Routinedefinition wird eine statische Variable N erklärt, die in Prozeduraufrufen als Funktions- oder Routinenamen verwandt werden kann. Die statische Variable N wird durch die Prozedurdefinition mit der Einsprungadresse der Routine oder Funktion vorbesetzt.

Der Kopf der Prozedur besteht aus dem Prozedurnamen, dem eine, möglicherweise leere, in Klammern eingeschlossene formale Parameterliste folgt.
Durch die formale Parameterliste werden dynamische Variable erklärt, deren Geltungsbereiche der Rumpf der zugehörigen Prozedur ist.
Den formalen Parametern werden erst zum Zeitpunkt des Prozeduraufrufes Speicherzellen zugeordnet, die mit den entsprechenden Werten der aktuellen Parameter initialisiert werden. Die Parameterübergabe ist also eine reine value-Übergabe, so daß Variable, die als aktuelle Parameter stehen, auch bei Wertzuweisungen an den entsprechenden formalen Parameter im Prozedurrumpf nach der Prozedurausführung nicht verändert sind. Dies läßt sich jedoch erreichen, wenn auf aktueller Parameterposition die Adresse (LV-Operator) eines BCPL-Elementes übergeben wird (CALL BY REFERENCE).
Über den RV-Operator kann das entsprechende BCPL-Element verändert werden.

Der Aufruf einer Prozedur kann wie folgt dargestellt werden:



1. Aufruf der Prozedur
2. Berechnung der R-Werte der aktuellen Parameter
3. Ihre R-Werte werden in aufeinanderfolgenden BCPL-Elementen abgelegt.
4. Aufruf der Prozedur, deren Einsprungadresse mit dem R-Wert von E0 übereinstimmt.
5. Die in 3. belegten BCPL-Elemente werden der Reihe nach den Formalparametern zugeordnet.
6. Routine- oder Funktionsausführung mit den aktuellen Parametern
7a. Routine-Aufruf: Rückkehr ins aufrufende Programm auf die dem Routineaufruf folgende Anweisung
7b. Funktions-Aufruf: Ergebnis ist der R-Wert des Expressions

Regeln:

a) Alle Namen, die in Funktions- oder Routinerümpfen vorkommen, müssen entweder dort deklariert sein, oder es müssen statische Variable oder MANIFEST-Konstanten sein.

b) Die als statische Variablen festgelegten Routine- oder Funktionsvariablen können durch eine GLOBAL- oder EXTERNAL-Deklaration zu globalen oder externen Variablen gemacht werden. Dann können diese Prozeduren auch aus anderen Montageobjekten eines Programms aufgerufen werden.

c) Falls Prozeduren nicht als nichtrekursiv deklariert sind, werden sie als rekursive Prozeduren angelegt.

d) Die Anzahl formaler und aktueller Parameter braucht nicht übereinzustimmen
Ist die aktuelle Parameterliste kürzer, wird nur die entsprechende Anzahl formaler Parameter initialisiert.
Ist die aktuelle Parameterliste länger, wird der Teil aktueller Parameter ignoriert, dem keine formalen Parameter zugeordnet werden können.

Beispiele:

a)
```
LET  NODE (X) = VALOF
     $ (  LET P = FREELIST          // NACH PEGELERHOEHUNG
          FREELIST := LV P ! 3      // EINTRAG IN LISTE
          P ! 0, P ! 1, P ! 2 := X, 0, 0   // AB NEUEM PEGEL
          RESULTIS P   $)           //

AND  PUT (X, T)  BE
     $ (  IF  T ! 0 = X RETURN              // SUCHWORT GEFUNDEN
          T := T ! 0 < X → LV T ! 1, LV T ! 2 // ERHOEHEN VERGLEICHS-
                                                WORT-ADRESSE
          TEST  RV T = 0                    // VERGLEICHSWORT = 0
                                                → AUFRUF
            THEN  RV T := NODE (X)          // NODE(X) sonst
            OR    PUT (X, RV T)    $)       // erneuter Aufruf PUT
```

Die Variable FREELIST in der Funktionsdefinition muß entweder als statische Variable oder als globale Variable deklariert sein.

b) LET A, B = 1, 2
   LET F(X) = A * X + B

Dieses Beispiel ist falsch, da A und B weder als statische Variablen noch als MANIFEST-Konstante vereinbart sind.

Richtig:

```
   STATIC  $( A = 1,  B = 2  $)
  LET  F(X) = A * X + B
```

c) LET S(X, Y) BE
        RV X: = Y
   LET A, B = 0, 1

Aufruf: S (LV A, B)

Als erster Aktualparameter steht die Adresse einer zuvor deklarierten Variablen.
Im Prozedurrumpf wird über den RV-Operator der Inhalt der so bezeichneten Speicherzelle verändert. Der aktuelle Parameter A hat also nach der Routineausführung den ihm über den RV-Operator zugewiesenen Wert 1.

2. MANIFEST-Deklarationen

Syntax:

MANIFEST ⟨ Deklarationsrumpf⟩

⟨Deklarationsrumpf⟩ :: = $(⟨C - DEF⟩ $\left[[;] ⟨C - DEF⟩ \right]_0^\infty$ $)

wobei

⟨C - DEF⟩ :: = N = K | N : K

N ist ein Name

K ist eine Konstante

Beschreibung:

Eine MANIFEST-Deklaration ordnet den im Rumpf erklärten Namen direkt die R-Werte von Konstanten zu. Die Zuordnung findet bereits zur Übersetzungszeit statt und kann im Objektlauf nicht mehr geändert werden. Die so deklarierten Namen sind keine Variablen und dürfen, da sie keinen L-Wert besitzen, nicht auf der linken Seite einer Zuweisung stehen. MANIFEST-Konstante dienen der besseren Selbstdokumentation eines Programms.

Außerdem ermöglichen sie die Parametrisierung eines Programms, sodaß bei Übergang auf eine andere Anlage nur die anlagenspezifischen Größen in den MANIFEST-Anweisungen geändert werden müssen (Beispiel d).

```
a) MANIFEST   $(  EINS = 1;  ZWEI = 2;  DREI = 3   $)
b) MANIFEST   $(      S.LET = 74
                      S.SEQ = 73
                      S.COMMA = 38   $)
c) MANIFEST   $(  PLUS = 90;  MINUS = 91   $)
   MANIFEST   $(  PLUSMALMINUS = MINUS   $)
d) MANIFEST   $(  BITSPERCHAR = 8;  CHARPERWORD = 3   $)
   MANIFEST   $(  BITSPERWORD = BITSPERCHAR * CHARPERWORD   $)
   MANIFEST   $(  CHARANZ = 30   $)
   MANIFEST   $(  LISTEND = (CHARANZ + (CHARPERWORD - 1)) /
                                        CHARPERWORD   $)

   LET  LIST = VEC  LISTEND - 1  AND
    LISTELEM = VEC CHARANZ - 1 AND
    VAR = 0
   :
   FOR I = 0  TO LISTEND - 1
   FOR J = 1  TO CHARPERWORD
  $(  VAR :=  LIST ! I
   LISTELEM !(CHARPERWORD * I + J - 1) :=
    (SLCT BITSPERCHAR : (BITSPERWORD - BITSPERCHAR)) OF  LV VAR
   IF  (CHARPERWORD * I + J - 1) = (CHARANZ - 1)   BREAK
   LIST ! I :=  LIST ! I  LSHIFT  BITSPERCHAR      $)
```

In Abhängigkeit der Wortstruktur (Bits pro Zeichen, Zeichen pro Wort), Anzahl zu verarbeitender Zeichen (CHARANZ) und Länge des die Zeichen in gepackter Darstellung enthaltenen Vektors wird ein neuer Vektor gefüllt, der die Zeichen ungepackt (ein Zeichen pro Vektorelement) enthält.

3. STATIC-Deklarationen

Syntax:

STATIC $ (⟨S-DEF⟩ [[;] ⟨S-DEF⟩]$_0^\infty$ $)

wobei

⟨S-DEF⟩ ::= N = { K | ⟨Stringkonstante⟩ | ⟨table⟩ }

N ist ein Name

K ist eine Konstante

Beschreibung:

Es werden statische Variable erklärt (siehe D 4.3), die mit den zugehörigen Werten im Deklarationsrumpf initialisiert werden. Die Zuordnung der Speicherzellen und ihre Initialisierung erfolgen bereits vor der Programmausführung. In STATIC-Deklarationen dürfen nur einfache Variable erklärt und vorbesetzt werden.

Der Initialwert dieser Variablen ist der Konstantenwert im Falle einer Konstanten, der String- oder Tablezeiger im Falle eines Strings oder Tables.

Bemerkung:

Table oder String dürfen auch in Klammernpaare eingeschlossen sein.

Beispiele:

```
a) STATIC   $(  P = 0; Q = 0
                REPORTMAX = 10   $)

b) STATIC   $(  S = "STRING"; T = TABLE 0,1,2   $)
```

4. GLOBAL- Deklarationen

Syntax:

GLOBAL ⟨Deklarationsrumpf⟩

⟨Deklarationsrumpf⟩ siehe E 2.

Beschreibung:

Mit der GLOBAL-Deklaration werden statische Variable erklärt, deren Speicherzellen im Globalvektor liegen (siehe D 4.1).
Dazu werden den Variablen im Globalvektor Konstanten zugeordnet, die die Relativadressen der zugehörigen Speicherzelle innerhalb des Globalvektors angeben.
Die gleiche globale Speicherzelle kann den Variablen verschiedener Montageobjekte zugeordnet sein, und kann deshalb dazu benutzt werden, Informationen zwischen den Montageobjekten auszutauschen. Die Zuordnung erfolgt dabei über die Adresse im Globalvektor, nicht über den Namen.
Nach folgender Regel können globale Variable vor der Programmausführung vorbesetzt werden. Tritt im Geltungsbereich einer globalen Variablen eine Funktions- oder Routinedefinition gleichen Namens auf oder steht dort eine gleichbenannte Marke, so werden diesen Variablen jene Speicherzellen zugeordnet, die auch für die globalen Variablen gleichen Namens im Globalvektor festgelegt sind. Die globale Variable wird in diesem Fall vor Programmausführung mit der Vorbesetzung der Funktions- Routine- oder Markenvariablen initialisiert.

Regel:

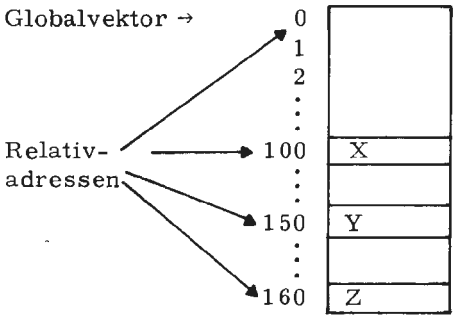
Jede Globalzelle darf in allen Montageobjekten eines Programms höchstens einmal vorbesetzt werden.

Implementierung TR 440:

a) Die Globalzellen 0, 2-9 dürfen nicht benutzt werden
b) Die Globalzelle 1 muß mit der Startadresse vorbesetzt sein d.h. es wird ihr jene Markenvariable zugeordnet, die im Programm den Beginn der Ausführung kennzeichnet

Beispiele:

a) GLOBAL $(X : 100; Y : 150 $)
GLOBAL $(Z : 160 $)



Es wird, auch bei mehreren GLOBAL-deklarationen, nur ein Globalvektor (eine C-Zone in TAS) angelegt

b)
```
GLOBAL  $( SUB : 90; ...          $)
  :           LET SUB(K, L, M)    BE
  :            $( ...             $)

GLOBAL  $( FU : 90, ...           $)
  :
BC := FU (A, B, C)
  :
```

// Definitionen stehen im 1. Teilprogramm

// 2. Teilprogramm

Bei Aufruf von FU (...) wird echt die Routine SUB aufgerufen da der Variablen FU im Globalvektor die gleiche Speicherzelle wie der Routinevariablen SUB zugeordnet wurde.

c)
```
GLOBAL  $( F : 103; G:104   $)
LET   F (A) = G(A)+A * A
AND   G (A) = (3 * A - 1) / 2
```

// Prozedurdefinitionen in Quelle 1

```
GLOBAL  $( F : 103; G : 104   $)
GLOBAL  $( START : 1   $)
START : $(
LET X = 3
LET Y = F (X)  AND  Z = G (X)
                              $)
```

// Prozeduraufrufe in Quelle 2

Über die gleiche Globalnummer werden in Quelle 2 die in Quelle 1 definierten Funktionen angesprochen.

5. EXTERNAL-Deklaration

Syntax:

EXTERNAL M, M, ... M (1)

EXTERNAL M : K, K, K, ... K (2)

EXTERNAL : K, K, ... K (3)

M ist Bezeichner einer Programmeinheit

K ist Bezeichner einer Kontaktvariablen

Wirkung:

Durch die EXTERNAL-Deklaration werden externe Variable definiert, die dazu dienen, Bezüge zwischen getrennt übersetzten Quellprogrammen (= verschiedenen Programmeinheiten) herzustellen, so daß mehrere Programmeinheiten untereinander über EXTERNAL-Variable Information austauschen können.

Genau in einer der Programmeinheiten muß jede externe Variable als STATIC-Variable, Funktions-, Routine- oder Marken-Variable definiert sein.

E

5.1. Systemnamen

In vielen Fällen ist ein Externbezug auf system-interne Programmeinheiten nötig, deren Namen nicht der Syntax für BCPL-Namen entsprechen. Es ist jedoch möglich, jeder externen Variablen einen BCPL-Namen zuzuordnen, unter dem sie in der zugehörigen aktuellen Programmeinheit angesprochen werden kann.

$$\left\{\begin{matrix} M \\ K \end{matrix}\right\} ::= \langle \text{BCPL-Name} \rangle \; [= \langle \text{Systemname} \rangle]$$

Die Programmeinheit bzw. die Kontaktvariable mit dem Namen ⟨Systemname⟩ kann nach dieser Deklaration in der aktuellen Programmeinheit unter dem BCPL-Namen angesprochen werden.

Regeln:

a) Der Systemname darf die Zeichen ":", "," und " ␣" nicht enthalten

b) Es werden nur die ersten 11 Zeichen ausgewertet

c) Wird einer externen Variablen kein Systemname zugeordnet, ergibt sich der Systemname aus dem BCPL-Namen selbst, wobei ein im Namen auftretender Punkt "." auf "&" abgebildet wird.

EXTERNAL N.1 ist gleichbedeutend mit
EXTERNAL N.1 = N&1

Implementierung TR 440: Ein Systemname in M-Position ist ein Montageobjektname.
Ein Systemname in K-Position ist ein Kontaktname.

5.2. M-Variable

Externe Variable, deren Systemnamen auf M-Position stehen, (Deklaration (1) und (2)) müssen innerhalb aller zum Programm gehörenden Programmeinheiten eindeutig sein. Sie dürfen nur einmal unter diesem Systemnamen als statische Variable definiert sein. Es dürfen nicht mehrere M-Variable in verschiedenen Programmeinheiten unter demselben Namen als externe Variable definiert werden.

5.3. K-Variable

Externe Variable, deren Systemnamen auf K-Position stehen müssen nur in der Programmeinheit eindeutig sein, in der sie als statische Variable definiert sind.
Sie beziehen sich also stets nur auf eine Programmeinheit. Mit der Deklaration (3) werden externe K-Variable deklariert, deren Definition als statische Variable in der gleichen Programmeinheit erfolgen muß. Fremde Programmeinheiten können über die Deklaration (2) auf diese K-Variablen zugreifen. Der Systemname auf M-Position bezeichnet die Programmeinheit, in der die K-Variablen definiert sind.
Über die Deklaration externer K-Variable ist es möglich, in verschiedenen Programmeinheiten gleichnamige externe Variable zu definieren. Bei Aufruf dieser K-Variablen von fremden Programmeinheiten ist die eindeutige Zuordnung zu einer Programmeinheit durch die M-Variable in Deklaration (2) (= Teilprogramm) festgelegt.

Regeln:

a) Externe Variable dürfen nicht zugleich global sein.

b) In der Deklaration (3) sind nur solche K-Variable zugelassen, die in der gleichen Programmeinheit als statische Variable definiert sind.

c) Werden in der Deklaration (2) die K-Variablen in der aktuellen Quelle definiert, so erhält die aus der Quelle erstellte Programmeinheit einen zusätzlichen Namen (nämlich den Systemnamen aus dem Bezeichner M).

Implementierung TR440: Dieser Name wird zusätzlich (!) zum Montageobjektnamen aus dem UEBERSETZE-Kommando vergeben.

d) Alle Namen einer EXTERNAL-Deklaration müssen voneinander verschieden sein.

Beispiele:

a)
```
EXTERNAL  A, B, C
STATIC  $(  A = 6     $)          // Quelle 1
LET  B (Z)  =  A * Z              // Definition der externen
LET  C (N) = 3 * N + A            // Variablen A, B und C
```

```
        EXTERNAL  A, B, C
        GLOBAL  $( ANF : 1   $)   // Quelle 2
ANF:    $( LET  X = 0             // Aufruf der externen Prozeduren
        A: =  B (A) + 10          // B und C, Bezug auf die externe
        X: =  C (3) + B (10)   $) // Variable A
```

b)
```
EXTERNAL   ABS = &ABS
```
In der öffentlichen Bibliothek gibt es eine Funktion &ABS, die vom aktuellen BCPL-Programm unter dem Namen ABS aufgerufen werden soll.

c)
```
EXTERNAL  :  Z, Q                 // Montageobjekt P&S&A
EXTERNAL  S.TEST : L1 = F1        // (aus UEB.-Kommando)
STATIC  $( Z = 99  $)             // Definition der Kontaktnamen
LET  Q (B) = B * Z + L1 (Z)       // Z und Q; Aufruf des Kontaktnamen
                                  // F1 (mit BCPL-Namen L1)
```

```
        GLOBAL   $(  BEG : 1   $)          // Montageobjekt S&TEST
        EXTERNAL  K = P&S&A : S1 = Z, Y = Q // Definition des Kontakt-
        EXTERNAL  : F1                     // namens F1.
        STATIC  $( Z2 = 30   $)            // Aufruf der Kontakt-
        LET  F1 (P) = P * (S1 + Z2)        // namen Z und Q (mit
BEG:    Z 2 : = S 1 + Y (3) -F1 (2)        // BCPL-Namen S1 und Y).
```

```
d)      EXTERNAL : A, B                              // Montageobjekt Z
        EXTERNAL  MOS :  AMOS = A, BMOS = B  //  Definition der Kontakt-
        GLOBAL  $(  ANF : 1  $)                      // namen A und B, Aufruf
        STATIC  $(  A = 7 ; B = 70  $)               // der Kontaktnamen A
ANF:    A := A + AMOS(  ) -2 * BMOS(B)               // und B aus Montageobjekt
                                                     // MOS (Mit BCPL-Namen
                                                     // AMOS und BMOS)

        EXTERNAL : A, B                              // Montageobjekt MOS
        EXTERNAL Z :  ZA = A, ZB = B                 // Definition der Kontakt-
        LET  A(  ) = ZA * B (ZA)                     // namen A und B; Aufruf
        AND  B(Q) = Q * ZB                           // Kontaktnamen ZA und
                                                     // ZB (= Kontaktnamen A
                                                     // u. B in Montageobjekt Z)
```

6. NONREC-Deklarationen

Syntax:

NONREC C : N1, N2, ... Nn

C ist eine Konstante zwischen 1 und 99

N1, N2, ... Nn sind Namen

Wirkung:

Funktionen und Routinen können in BCPL als rekursiv oder nicht-rekursiv deklariert werden. Eine Prozedur, die im Programm nicht-rekursiv verwandt wird, sollte aus Laufzeitgründen als NONREC deklariert werden. Wird eine Routine oder Funktion rekursiv benutzt, muß dementsprechend die NONREC-Deklaration entfallen.

Klassenzuordnung

Mit der NONREC-Deklaration wird den durch die Namen N_i bezeichneten Prozeduren eine Klasse C zugeordnet. Durch die Klassenzuordnung wird allen Funktionen und Routinen derselben Klasse C ein gemeinsamer Arbeitsspeicher zugeteilt; seine Länge ergibt sich als Maximum der von den verschiedenen Prozeduren der Klasse C benötigten Längen der Arbeitsspeicher.

Aufrufmöglichkeiten

Der gemeinsame Arbeitsspeicher bedingt, daß in einer aktuellen Aufrufverschachtelung nie zwei nicht-rekursive Prozeduren derselben Klasse C vorkommen dürfen.
Ebenso ist beim Aufruf einer nicht-rekursiven Prozedur verboten, eine Funktion derselben Klasse auf Parameterposition aufzurufen.
Sonst bestehen keine Einschränkungen für die Verschachtelungen von rekursiven und nicht-rekursiven Prozeduren.
Bei der Vergabe der Klassen sollte beachtet werden, daß der benötigte Arbeitsspeicher mit der Zahl der verwendeten NONREC-Klassen steigt.

Aufrufeigenschaft des Hauptprogramms

Das Hauptprogramm, welches die START-Marke enthält, entspricht dem Rumpf einer rekursiven Prozedur, wenn es nicht in einem Prozedurrumpf liegt.
Liegt es in einem Prozedurrumpf und wird dieser als nicht-rekursiv (NONREC) deklariert, so ist zu beachten, daß die für das Hauptprogramm vergebene Klasse nicht mehr verwendet werden darf (siehe Klassenzuordnung).

Geltungsbereich von NONREC-Deklaration

Der Geltungsbereich einer NONREC-Deklaration (NONREC C : N1, N2, ... Nn) beginnt hinter dieser Deklaration und endet, falls vorhanden, am Ende des kleinsten umgebenden Blockes, oder am Ende der Programmeinheit.
Zwei Bereiche sind gegebenenfalls vom Geltungsbereich einer NONREC-Deklaration ausgenommen:

a) Wird in ihrem Geltungsbereich mit einer MANIFEST-Deklaration eine MANIFEST-Konstante gleichen Namens deklariert, so ist im Geltungsbereich dieser MANIFEST-Deklaration die ursprüngliche NONREC-Deklaration außer Kraft gesetzt. Wird der Geltungsbereich der MANIFEST-Deklaration verlassen, so gilt wieder die ursprüngliche NONREC-Deklaration.

b) Wird im Geltungsbereich einer NONREC-Deklaration demselben Namen durch eine weitere NONREC-Deklaration eine andere Klasse zugeordnet, so tritt die dynamisch zuerst definierte Zuordnung erst dann wieder in Kraft, wenn der Geltungsbereich der zweiten NONREC-Deklaration verlassen wird.

Regeln:

a) Wurde eine Prozedur im Geltungsbereich einer NONREC-Deklaration (für ihren Namen) definiert, so darf sie nur über eine Variable aufgerufen werden, der dieselbe Klasse zugeordnet ist.

b) Sind Aufruf und Definition einer Funktion oder Routine in verschiedenen Programmeinheiten enthalten, müssen die bei der Prozedurdeklaration zugeordneten Aufrufeigenschaften (NONREC der Klasse C oder nicht NONREC) unverändert in die den Prozeduraufruf enthaltende Programmeinheit übernommen werden.

c) Eine nicht-rekursive Prozedur der Klasse C darf keine nicht-rekursive Prozedur der gleichen Klasse aufrufen, da ihre Variablen sonst überschrieben werden. Wird diese Einschränkung nicht beachtet, ist die weitere Programmausführung undefiniert.

d) Die Deklaration `NONREC C: N1, N2, ... Nn`
ist gleichwertig mit
```
NONREC C: N1
NONREC C: N2
  :      :
NONREC C: Nn
```

Implementierung TR 440: Die NONREC-Klasse C muß der Forderung

$$1 \leq C \leq 99$$

genügen.
Die Klassen $1 \leq C \leq 9$, sind für Standardprozeduren (EA u.s.w.) reserviert und dürfen deshalb nicht benutzt werden.

Beispiele:

a) Definierende Quelle:
```
NONREC  10 : A
  :
LET  A (X)  BE
$( ...  $)
```

Aufrufende Quelle:

```
NONREC  10 : A
  :
  :
A (20)
  :
  :
```

b) Definierende Quelle:

```
NONREC  11 :  FUN1, FUN2
LET FUN1 (Y) =  Y * Y + 9
LET FUN2 (Z) =  7 * (Z + 3)
```

Aufrufende Quelle:

```
NONREC  11 :  FUN2
NONREC  11 :  FUN1
  :
  :
Q : =  FUN1 (3)  +  FUN2 (17)
```

c) Falsches Beispiel:

```
NONREC  15 :  SUB
LET  SUB (P, Q)  BE
$(  NONREC  15 :  FUN
    LET  FUN (S) = S * S + 3   AND   K = 0
    K : =  FUN (P) + Q
    :
    :          $)
SUB (13, 2)
```

Dieses Beispiel ist falsch, da bei Aufruf der Routine SUB an der aktuellen Aufrufverschachtelung eine Funktion beteiligt ist, die der gleichen Klasse C wie die aufrufende Routine SUB angehört. Dieses Beispiel wird korrekt, wenn der Funktion FUN eine Klasse ungleich 15 zugeordnet wird.

d) Falsches Beispiel:

```
NONREC  10 :  H
NONREC  20 :  F, G
EXTERNAL  F, G, H, L
F   (G (1))                    // verboten :  F, G haben gleiche Klasse
  :
  :
H : = F
F (20)
H (20)                         // Fehler: Unterschiedliche Klassen-
                               // zuordnung bei Deklaration und
                               // Aufruf
```

e) Falsches Beispiel:

```
NONREC  10:  F.PLUS
NONREC  11 :  F. MINUS
EXTERNAL  F.PLUS, F.MINUS
GLOBAL  $(  OP :  100  $)
  :
  :
(OP = '+' → F.PLUS, F. MINUS) (A, B)  // verboten, da Aufruf über
                                      // Ausdruck, der keine Varia-
                                      // ble ist.
```

f)

```
NONREC 17:  ERG
STATIC  $( SUM = 0  $)
LET  ERG (AN, EN, FAK)  BE
$( FOR  I = AN  TO  EN  DO
SUM :=  SUM + FAK  $)
  :
  :
ERG  (2, 200, 10)
  :
  :
$(N NONREC  18 : ERG
  LET  ERG (AN, EN, FAK)  BE
$ ( FOR I = AN  TO  EN  DO
    SUM :=  SUM * FAK  $)
ERG  (0, 10, 5)
  :
  :
  $)N

ERG  (1, 100, 3)
```

Im Gültigkeitsbereich der zweiten NONREC-Deklaration ist die Klassenzuordnung des umgebenden Blockes aufgehoben. Wird der Bereich der zweiten NONREC-Deklaration verlassen, tritt die Gültigkeit der zuerst definierten Routine ERG wieder in Kraft.

Da im Gültigkeitsbereich der inneren NONREC-Deklaration die Routine auch definiert wird, ist der Aufruf zulässig.

falsch wäre

```
  :
  :
$(N    NONREC  18 :  ERG
       ERG ( 2, 200, 10)
```

ANWEISUNGEN

ANWEISUNGEN

Im Kapitel "Anweisungen" werden häufig die Begriffe R-Wert und L-Wert benutzt. Diese Begriffe sind in Kapitel C 6. ausführlich erläutert.

1. Zuweisungen

1.1. Einfache Zuweisung

Allgemeine Form:

E1 : = E2

Bezeichnung:

E1 ist ein Ausdruck mit L-Wert

E2 ist ein beliebiger Ausdruck

Zuweisungen, bei denen mindestens ein Operand ein Selektorausdruck ist, werden in Kapitel F 1.2 erläutert.

Wirkung:

Es werden der R-Wert von E2 und der L-Wert von E1 berechnet. Der Inhalt des BCPL-Elementes, dessen Adresse sich aus dem L-Wert von E1 ergibt, wird durch den R-Wert von E2 ersetzt.

Beispiele:

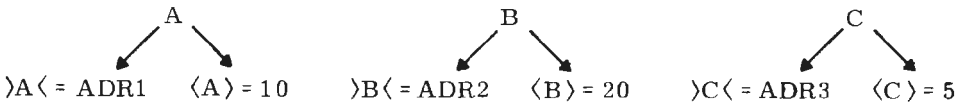
a) A : = B+C

Die R-Werte (Inhalte) der mit B und C bezeicneten BCPL-Elemente werden addiert und ersetzen den Inhalt des mit A bezeichneten BCPL-Elements.

Es bezeichne:

⟨ n ⟩ → den Inhalt n

⟩ n ⟨ → die Adresse des BCPL-Elements n, sei ferner



dann gilt:

L-Wert von A = ADR1
R-Wert von B = 20
R-Wert von C = 5

Ergebnis: (A) = 25

Der Inhalt des BCPL-Elements A (=10) wird durch die addierten R-Werte von B und C (= 25) ersetzt.

b) C → B, D : = F

Abhängig von der Variablen C wird entweder die Zuweisung B : = F (falls C = TRUE) oder D : = F (falls C = FALSE) ausgeführt.

1.2. Teilwortzuweisungen

Allgemeine Form:

E1 : = E2

Bezeichnung:

Die Teilwortzuweisung unterscheidet sich von der einfachen Zuweisung dadurch, daß der Ausdruck E 1 ein Selektorausdruck (siehe G. 6.2) von der Form S OF E ist. S ist dabei ein Selektor, E und E2 sind beliebige Ausdrücke.

Wirkung:

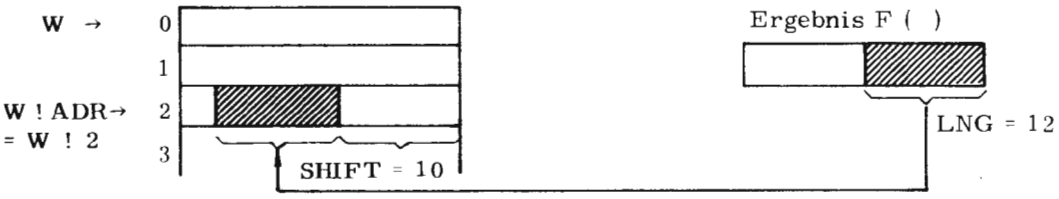
Die Teilwortzuweisung erlaubt es, einzelne Bits und Bitfelder in einem BCPL - Element umzusetzen, ohne den Rest des BCPL-Elements zu verändern.

Ein Selektor S bestimmt die Bitfeldlänge LNG und den rechten Randabstand (im BCPL-Element) SHIFT des Bitfelds, sowie die Relativadresse ADR des BCPL-Elements (zum Wert des Zeiger-Ausdrucks E).

Beispiele:

a) MANIFEST $(LNG = 12; SHIFT = 10; ADR = 2 $)
MANIFEST $(SL = SLCT LNG : SHIFT : ADR $)
LET W = VEC 20
SL OF W : = F( ) .// OF kann gleichbedeutend durch :: ersetzt werden (siehe Alternativdarstellungen O 2).

Mit der ersten MANIFEST-Anweisung werden 3 Selektorgrößen bestimmt, die in der 2. MANIFEST-Anweisung dem Selektor SL zugeordnet werden (siehe auch G 6.2).
Die folgende Zuweisung bewirkt, daß die rechtsbündigen 12 Bits (LNG = 12) des Funktionswertes F ( ) diejenigen 12 Bits von W ! 2 (Relativadresse ADR = 2) ersetzen, die vom rechten Rand einen Abstand von 10 Bits haben (SHIFT = 10).

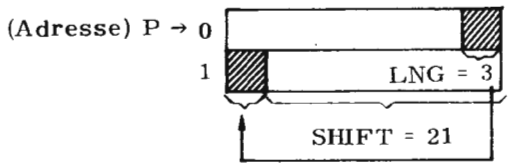


Bit 3 - 14 von W ! 2 werden also durch die Bit 13 - 24 des Ergebnisses des Funktionsaufrufs F ( ) ersetzt.

Ein Selektor auf der rechten Seite wird behandelt wie ein normaler Ausdruck (siehe G 6.2).

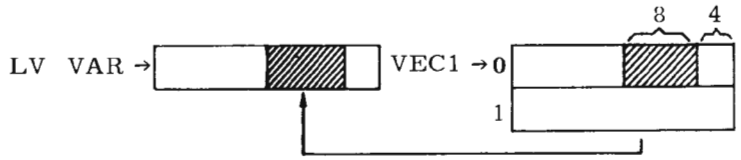
b) Auf beiden Seiten der Zuweisung stehen Selektorausdrücke

```
MANIFEST   $( S0 = SLCT 3 : 21 : 1   $)
LET P = VEC ␣ 1
 S0 OF P := (SLCT 3) OF P
                    // SHIFT = 0;  ADR = 0
```



c) Selektorausdruck in Verbindung mit einer einfachen Variablen

```
MANIFEST   $( W1 = SLCT 8 : 4   $)
LET VAR = NIL  AND  VEC1 = VEC ␣ 1
W1 OF LV VAR := W1 OF VEC1        // Zuweisung auf VAR
```



Bei Selektoren, die auf eine einfache Variable (kein Vektor) wirken sollen, ist zu berücksichtigen, daß in der Anwendung "S OF E" E stets als Zeiger aufgefaßt wird und auf ein Bitfeld in dem BCPL-Element E ! ADR weist. Mit einer einfachen Variablen sollten i.a. keine Vektorausdrücke gebildet werden, da ihr R-Wert i.a. nicht die Anfangsadresse einer BCPL-Elementfolge ist.

Teilwortmanipulationen auf einfachen Variablen sind dennoch möglich, und zwar genau dann, wenn die Relativadresse ADR des Selektors Null ist und wenn als Wert des Zeigerausdrucks E der L-Wert einer Variablen VAR angegeben wird, etwa durch LV VAR.

Der Ausdruck LV VAR besitzt als R-Wert den L-Wert von VAR. Es wird auf das 0-te BCPL-Element (ADR = 0) relativ zu diesem Zeiger-Wert zugegriffen, d.h. die Variable VAR selbst wird angesprochen.

1.3. Mehrfachzuweisungen

Allgemeine Form:

$E_{L1}, E_{L2}, \ldots E_{Ln} := E_{R1}, E_{R2}, \ldots E_{Rn}$ Vgl. 1.1

Bezeichnung:

Die E_{Li} sind L-Ausdrücke oder Selektorausdrücke

Die E_{Ri} sind beliebige Ausdrücke

Wirkung:

Die Mehrfachzuweisung dient zur vereinfachten Schreibweise von mehreren aufeinanderfolgenden Einzelzuweisungen. Sie entsprechen der Folge:

$E_{L1} := E_{R1}$
$E_{L2} := E_{R2}$
$\vdots$
$E_{Ln} := E_{Rn}$

Auf der rechten Seite einer Mehrfachzuweisung können aufeinanderfolgende, gleiche Ausdrücke durch die Anwendung eines Replikators zusammengefaßt werden (siehe G 10).

Da die Mehrfachzuweisung mehrere Zuweisungen in einer Anweisung zusammenfaßt, können in bestimmten Fällen Blockklammern gespart werden.

a) X, Y : = Y, X

entspricht: X : = Y

Y : = X

Diese Anweisung vertauscht nicht die Werte von X und Y, da zum Zeitpunkt der 2. Zuweisung X bereits mit dem Wert von Y belegt ist.
Damit ist die 2. Zuweisung gleichbedeutend mit

Y : = Y

b) IF X = Y DO
$(V ! 3 : = 0; B : = TRUE $)

Die in diesem Fall durch die Syntax der IF-Anweisung vorgeschriebenen Blockklammern (beide Zuweisungen sollen als zur Anweisung gehörend angesehen werden) können weggelassen werden, wenn folgende Mehrfachzuweisung verwendet wird:

IF X = Y DO
V ! 3, B : = 0, TRUE

c) (SLCT 4 : 8 : 1) : : W1, A : = (SLCT 4) : : W1, B

2. Routine-Aufrufe

Allgemeine Form:

E0 (E1, E2, ... En)

Bezeichnung:

E0 ist ein einfacher Ausdruck

E1, ... En sind beliebige Ausdrücke

E1, ... En heißen aktuelle Aufrufparameter

Wirkung:

Durch den Routine-Aufruf wird ein Unterprogrammsprung ausgeführt, dessen Zieladresse der R-Wert von E0 ist.
Die ersten n formalen Parameter der Prozedur E0 (Routine oder Funktion) werden mit den R-Werten der aktuellen Parameter E_1 ... E_n initialisiert.
Der Rücksprung aus dem Unterprogramm erfolgt in das aufrufende Programm auf die dem Routine-Aufruf folgende Anweisung.

Regeln:

a) Die Anzahl von formalen und aktuellen Parametern kann unterschiedlich sein. Ist die aktuelle Parameterliste kürzer, werden nur die entsprechenden formalen Parameter mit den angegebenen Werten initialisiert. Ist die aktuelle Parameterliste länger, werden die überflüssigen aktuellen Parameter ignoriert.

b) Die Liste der aktuellen Parameter kann leer sein.
Die beiden Klammern dürfen dann nicht wegfallen.

Beispiele:

```
a)      FN1 (X)
b)      VNEW.F := BEF ( )
```

Zuweisungswert ist das Ergebnis einer parameterlosen Funktion

```
c) LET  CHECKOP (OP) = VALOF                    // Funktionsdefinition
   $(   IF OP = '+'     RESULTIS F.PLUS
        IF OP = '-'     RESULTIS F.MINUS
         RESULTIS FEHLER          $)
    :
        CHECKOP(AKTOP) (A, B)                   // Routineaufruf nach
                                                // Funktionsaufruf
```

In Abhängigkeit eines Operators AKTOP wird entweder

```
        F. PLUS   (A, B),
        F.MINUS   (A, B)
oder    FEHLER(A, B) aufgerufen.
```

3. Markierte Anweisungen

Allgemeine Form:

⟨name⟩ : C

Bezeichnung:

⟨name⟩ ist ein Name

C kann eine beliebige, auch leere, Anweisung sein. Eine leere markierte Anweisung ist gegeben, wenn die Marke "⟨name⟩:" vor einer schließenden Blockklammer oder am Ende des Programmes steht.

Wirkung:

Durch die Marke ⟨name⟩ : wird eine (statische) Markenvariable ⟨name⟩ deklariert, die mit der Programmadresse der markierten Anweisung initialisiert ist.

Marken werden i.a. als Sprungziele verwandt.

Über den Geltungsbereich von Marken siehe D 3.

Beispiele:

a) MARK 1 : F( ) (B, C)

Bei Ansprung der Marke MARK 1 wird eine Routine aufgerufen, deren Einsprungadresse sich aus dem Funktionswert von F( ) ergibt.

b) $(... M : $) REPEATWHILE BOOL

Wird die Marke M angesprungen, so wird die Blockende-Behandlung (REPEATWHILE BOOL) angestoßen, d.h. die Wiederholungsbedingung BOOL abgefragt.

Abhängig von deren Wert wird ein weiteres Mal der Block durchlaufen oder mit der auf die Wiederholungsabfrage folgenden Anweisung fortgefahren.

c)
```
      GOTO  M1          // Anfangsbehandlung
       :                // in einem Programm, das immer
M1  : X := Y            // mit GOTO M1
      M1 := M2          // angesprungen wird
       :
M2  :
       :
```

Die Anweisung " GO TO M1" veranlaßt in diesem Falle einen Sprung auf die Marke M2.

d) ENDE : FINISH

Bei Ansprung der Marke ENDE wird die Programmendebehandlung ausgeführt.

e) LET F(A) BE
$(L: $)

Der Ansprung der Marke L veranlaßt in diesem Prozedurrumpf den Rücksprung in das aufrufende Programm.

4. GOTO-Anweisung

Allgemeine Form:

GOTO E

Bezeichnung:

E ist ein beliebiger Ausdruck

Wirkung:

Der lineare Programmablauf wird durch einen Sprung unterbrochen. Das Sprungziel ist die Anweisung, deren Programmadresse mit dem R-Wert von E übereinstimmt.
Es können nur markierte Anweisungen angesprungen werden; i.a. erfolgt der Ansprung über die zugeordnete Markenvariable

```
z.B.    GOTO  M1
          :
          :
        M1: ...
```

Regel:

a) Sprünge aus oder in Prozedurrümpfe mittels GOTO-Anweisung sind verboten; ebenso Sprünge in VALOF-Blöcke, FOR-Schleifen und Unterblöcke mit nichtleerem Deklarationsteil.

b) Bei unzulässigen Sprüngen, z.B. mit Hilfe eines GLOBALs erfolgt keine Warnung.

Beispiele:

a) GOTO NEXT

Es wird zu der Anweisung gesprungen, deren Marke bei der Initialisierung mit dem R-Wert der statischen Variablen NEXT übereinstimmt.
Das braucht nicht unbedingt die gleichlautende Marke NEXT zu sein.

```
M1 : P( )
     ---
     NEXT := M1
     ---
     GOTO NEXT
```

Der statischen Variablen NEXT wurde vor der Sprunganweisung der R-Wert der Marke M1 zugewiesen. Somit wird mit GOTO NEXT auf die Marke M1 gesprungen.

b) GOTO S ! I

Gemäß Definition kann an der Position von E ein beliebiger Ausdruck stehen. Wird als Sprungziel nicht der Name einer (nicht umgesetzten) Marke angegeben, so muß der Benutzer Sorge tragen, daß eine sinnvolle und definierte Sprungausführung stattfindet.

```
S ! 0 := M1
S ! 1 := M2
---     ---
---     ---
S ! N := MN
GOTO  S ! I
```

Der Vektor S wurde vor dem Sprungbefehl mit den R-Werten der Sprungmarken M1 ... MN belegt. Der Sprungbefehl wird nun auf die Marke ausgeführt, deren R-Wert im aktuellen Vektorelement S ! I steht.

c) GOTO X = 0 → L, F (X)

Im Falle X = 0 wird zur Anweisung mit der Marke L gesprungen, sonst auf die Marke, die den Funktionswert F (x) als R-Wert besitzt.

5. IF - , UNLESS - Anweisung

Allgemeine Form:

IF E DO C
UNLESS E DO C

Bezeichnung:

E ist ein beliebiger Ausdruck

C ist eine beliebige Anweisung

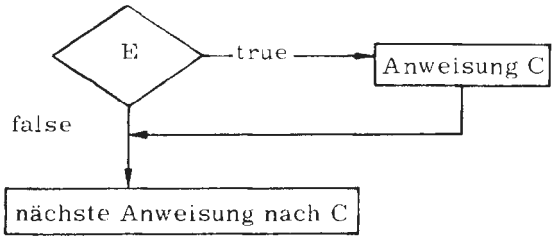
Wirkung:

Für die IF-Anweisung gilt:

Der R-Wert von E wird berechnet. Das Ergebnis darf nur TRUE oder FALSE sein. Ist der R-Wert von E TRUE, wird die Anweisung C ausgeführt, ist das Ergebnis FALSE, wird die Anweisung C übersprungen und mit der statisch darauffolgenden Anweisung fortgefahren.

UNLESS E DO C ist äquivalent mit
IF NOT E DO C

Darstellung der IF-Anweisung als Flußdiagramm:



Regeln:

a) Das Schlüsselwort DO kann weggelassen werden, wenn die syntaktische Eindeutigkeit nicht gefährdet ist.

b) Ist E kein logischer Wert, dann ist die weitere Ausführung undefiniert.

Beispiele:

a) IF X>0 DO X:= X ! 0

Ist der R-Wert von E (X>0) TRUE, wird die auf DO folgende Anweisung (X:= X !0) ausgeführt. Ist das Ergebnis FALSE (X≤0), wird diese Anweisung übersprungen und mit der nächsten fortgefahren.

b) IF X LE Y LE Z B:= TRUE
UNLESS X LE (Y+Z) GOTO W!9

Da die syntaktische Eindeutigkeit gewährleistet ist, kann in beiden Fällen das Schlüsselwort DO weggelassen werden.
Die beiden Anweisungen sind äquivalent mit

IF X LE Y LE Z DO B:= TRUE
UNLESS X LE (Y+Z) DO GOTO W!9

Der Sprungbefehl in der 2. Anweisung wird genau dann ausgeführt, wenn X>(Y+Z) ist.

c) UNLESS MASK::W1 = MASK::W2 $(F( ); RUF (MI,A) $)

Außer wenn die angesprochenen Bitgruppen identisch sind, wird bei dieser Anweisung die Routine RUF aufgerufen.

6. WHILE-, UNTIL-Anweisung

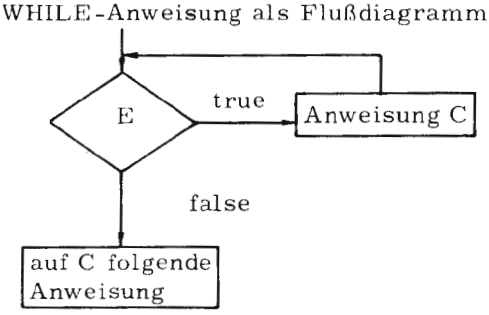
Allgemeine Form:

WHILE E DO C
UNTIL E DO C

Bezeichnung:
E ist ein beliebiger Ausdruck
C ist eine beliebige Anweisung

Wirkung:
Ist bei Durchlaufen der Anweisung WHILE der R-Wert des Ausdruckes E TRUE, wird die Anweisung C ausgeführt. Die Anweisung C wird solange wiederholt ausgeführt, bis der R-Wert von E den Wert FALSE annimmt. Im Falle E = FALSE wird die Anweisung C übersprungen und mit der statisch nächsten Anweisung fortgefahren.

Darstellung der WHILE-Anweisung als Flußdiagramm



UNTIL E DO C ist gleichbedeutend mit
WHILE NOT (E) DO C

Regeln:

a) Das Schlüsselwort DO kann weggelassen werden, wenn die syntaktische Eindeutigkeit gewährleistet ist.

b) Ist E kein logischer Wert, dann ist die weitere Ausführung undefiniert.

Beispiele:

a)
```
I := 0
WHILE I NE 10 DO $( ERG ! I := F(I)
                    I := I+1    $)
```

Die in Blockklammern stehenden Anweisungen werden solange wiederholt ausgeführt, bis I den Wert 10 annimmt.
Bei I = 10 wird nicht mehr C, sondern die darauffolgende Anweisung durchlaufen.

Obiges Beispiel ist äquivalent mit

```
I := 0
UNTIL I=10  $( ERG ! I := F(I)
               I := I+1   $)
```

Und mit

```
FOR J = 0 TO 9 DO
   ERG ! J := F(J)
I := 10
```

b) WHILE B

```
$(  B:= READ(5,"/S26, A3", 2, ST, LV Z)
    :
      $)
```

Die Anweisung C wird solange ausgeführt, bis B den Wert FALSE annimmt.

c) UNTIL BFUN(X) = AFUN( )

```
        $( V ! X :=    BFUN (X)
           X :=X+ 1       $)
```

Erst wenn BFUN(X) = AFUN( ) ist, wird die Anweisung C übersprungen.

7. TEST-Anweisung

Allgemeine Form:

```
TEST  E  THEN  C1  OR  C2
```

Bezeichnung:

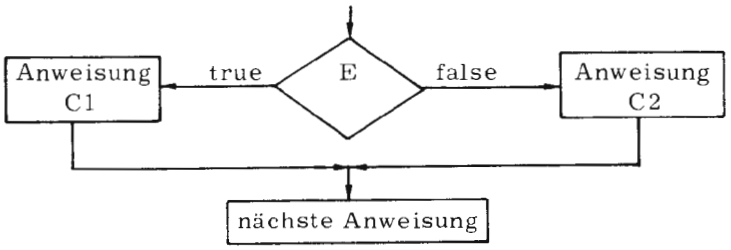
E ist ein beliebiger Ausdruck

C1 und C2 sind beliebige Anweisungen.

Wirkung:

In Abhängigkeit vom R-Wert des Ausdruckes E wird Anweisung C1 oder Anweisung C2 ausgeführt. C1 wird durchlaufen, wenn E den Wert TRUE hat, C2 wird durchlaufen im Falle E = FALSE. Die jeweils nicht angesprochene Anweisung wird übersprungen und mit der auf die TEST-Anweisung statisch folgenden Anweisung fortgefahren.

Darstellung der TEST-Anweisung als Flußdiagramm



Regeln:

a) Ist der R-Wert von E weder TRUE noch FALSE, ist die weitere Ausführung undefiniert.

Beispiele:

```
a) TEST  LV  A     EQ    V ! P
   THEN  LIST ! P := LV  V ! P
   OR          A := V ! P

b) TEST  A  GE  10  THEN  A := 0
      OR   $( A := A+1;   P(A)   $)
```

8. REPEAT-Anweisung

Allgemeine Form:

```
C  REPEAT
```

Bezeichnung:

C ist eine beliebige Anweisung. Sie ist die letzte Anweisung vor REPEAT.

Wirkung:

REPEAT veranlaßt die wiederholte Ausführung der Anweisung C. Im allgemeinen ist C ein Block, dessen wiederholte Ausführung durch eine programmierte Unterbrechung (z.B. GO TO oder BREAK → siehe F 1.) beendet wird.

Beispiele:

a)
```
$( ... IF  A  GO TO M1
       ...    $)   REPEAT
```

Die Anweisung C wird solange ausgeführt, bis auf Grund der IF-Abfrage der Block mit einem Sprung auf die außerhalb des Blockes liegende Marke M1 verlassen wird.

b)
```
$ ( S ! J := PNEU-PALT
      PALT := PNEU; PNEU := KFU ( )
   TEST (PNEU-PALT) = 0  THEN  BREAK
   OR   J := J + 1       $)   REPEAT
```

Die Ausführung der Wiederholungsanweisung wird beendet, wenn die Anweisung BREAK erreicht wird.

c)
```
IF  A = 0  R( )  REPEAT
```

Die letzte Anweisung vor REPEAT ist der Routineaufruf R ( ). REPEAT bezieht sich also nicht auf die gesamte IF-Anweisung. Obiges Beispiel wird ausgeführt wie

```
IF  A = 0  DO  $( R( )  REPEAT  $)
```

Der Abbruch muß in R z.B. durch FINISH erfolgen.

9. REPEATWHILE -, REPEATUNTIL - Anweisung

Allgemeine Form:

C REPEATWHILE E

C REPEATUNTIL E

Bezeichnung:

E ist ein beliebiger Ausdruck

C ist eine beliebige Anweisung; sie ist die letzte Anweisung vor REPEATWHILE E bzw. REPEATUNTIL E.

Wirkung:

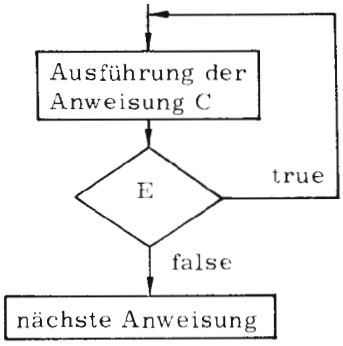
Nach Ausführung der Anweisung C wird der Ausdruck E der REPEATWHILE-Anweisung auf die Wahrheitswerte TRUE und FALSE untersucht. Ist der Ausdruck TRUE, wird die Anweisung C erneut durchlaufen. Dies wird solange wiederholt, bis E den Wert FALSE annimmt. Es wird dann nach der REPEATWHILE-Anweisung fortgefahren. Die Anweisung C wird mindestens einmal durchlaufen.

Ein Abbruch der Schleifenverarbeitung kann auch durch eine in der Anweisung herbeigeführte Unterbrechung erfolgen (z. B. GOTO oder BREAK siehe F 11.).

C REPEATUNTIL E ist äquivalent mit

C REPEATWHILE NOT E

REPEATWHILE-Anweisung als Flußdiagramm



Regel:

a) Ist der Wert von E weder TRUE noch FALSE, ist die weitere Ausführung undefiniert.

Beispiele:

a) $(... M : $) REPEATWHILE LOGI

Wird die Marke M erreicht, so wird die vorgegebene Bedingung abgeprüft. Hat die Variable LOGI den Wert TRUE, wird der Anweisungsblock wiederholt und zwar so oft, bis LOGI den Wert FALSE annimmt. Dann wird nach der Wiederholungsanweisung fortgefahren.

b)
```
TEST BOOL 1 THEN
    TEST BOOL 2 THEN $(1 K ! I : = F (TRUE, I)
                         BOOL 1 : = FALSE $)1
                    OR $(2 K ! I : = P (FALSE, I)
                         I : = I + 1 $)2
                           REPEATUNTIL K ! I = TRUE
                    OR MISS (FALSE)
```

Die Wiederholungsanweisung bezieht sich nur auf den mit 2 benannten Block, da gemäß Syntax nur die letzte vorgehende Anweisung C angesprochen wird.

c)
```
$( WP : = WP + 1
   READ (5, "A1", 1, LV V ! WP)
            $) REPEATWHILE 'A' LE V ! WP LE 'Z'
```

Es wird der Block solange wiederholt durchlaufen, bis eines der durch READ eingelesenen Zeichen kein Buchstabe ist. Der binäre Wert des Zeichens im Zentralcode-Sinne ist dann entweder kleiner als der von 'A' oder größer als der von 'Z'.

10. FOR-Schleifen

Allgemeine Form:

```
FOR N = E1 TO E2 [ BY E3 ] DO C
```

Bezeichnung:

N ist ein beliebiger Name

E1, E2 sind beliebige Ausdrücke

E3 ist ein konstanter Ausdruck

Wirkung:

Die FOR-Anweisung enthält die Deklaration einer dynamischen Variablen mit Namen N (Laufvariable); diese Variable ist in der Anweisung C bekannt. Beim Verlassen der FOR-Schleife wird die Variable aufgegeben. Vor dem ersten Durchlauf erhält die Laufvariable den Anfangswert; übersteigt der Wert der Laufvariablen den Endwert (bei positiver Schrittweite) bzw. unterschreiteet den Endwert (bei negativer Schrittweite) so wird hinter der FOR-Anweisung fortgefahren. Im anderen Fall wird die Anweisung C durchgeführt, die Laufvariable um die Schrittweite erhöht bzw. vermindert und die Prüfung erneut durchgeführt.
Der Endwert wird nur einmalig am Schleifenanfang berechnet. Ist bereits hier der Endwert kleiner als die Laufvariable (bei positiver Schrittweite) bzw. größer als die Laufvariable (bei negativer Schrittweite) wird hinter der FOR-Anweisung fortgefahren.

Regeln:

a) Das Schlüsselwort DO kann weggelassen werden, wenn die syntaktische Eindeutigkeit gewährleistet ist.

b) Die Schrittweitenangabe "BY E3" ist optional. Entfällt sie, wird die Schrittweite 1 angenommen.

c) E1 und E2 werden nur einmal zu Beginn der Ausführung der FOR-Schleife berechnet.

Beispiele:

a)
```
FOR N = 0 TO 10 BY 2 DO
  C (N);
```
ist äquivalent mit
```
$(A  LET N = 0
  WHILE N LE 10 DO $( C(N);
                      N := N+2  $)A
```

ist äquivalent mit

```
$(  LET   N = 0
 LOOPANF:  IF  N>10  GO TO  M1
           C (N);  N:= N+2
           GO TO  LOOPANF
           M1 :   $)
```

b) FOR LAUF = -10 TO 0 DO
FELD !(LAUF + 11) : = F (LAUF) REM (LAUF-1)
entspricht:

```
FELD ! 1 : = F (-10)  REM  -11
FELD ! 2 : = F  (-9)  REM  -10
:
FELD ! 11 : = F (0)   REM  -1
```

c) FOR I = 1 TO F (CAL) BY 2
CAL : = F (I)

Trotz des im Schleifenrumpf sich ändernden CAL ist der Wert des Endparameters eindeutig festgelegt, da er nur einmal zu Beginn der Schleife berechnet wird.

F

11. BREAK-Anweisung

Allgemeine Form:

BREAK

Wirkung:

Die BREAK-Anweisung bewirkt, daß die Abarbeitung eines Schleifenblockes abgebrochen und hinter die innerste, die BREAK-Anweisung umgebende Schleife gesprungen wird.

Regel:

BREAK-Anweisungen sind nur erlaubt in Blöcken, die Teil einer Schleife sind (WHILE-, UNTIL-Anweisungen, Wiederholungsanweisungen und FOR-Schleifen).

Beispiele:

a) TEST P THEN
$(IND := F(IND)
IF IND = 0 BREAK
S ! IND := P / IND $) REPEAT
OR Z (Q)

Der TRUE-Zweig der TEST-Anweisung wird solange wiederholt durchlaufen bis IND = 0 ist.

Der Sprung hinter die umfassende Schleife erfolgt nicht auf den FALSE-Zweig der TEST-Anweisung, sondern auf die der gesamten TEST-Anweisung folgende Anweisung.

b) UNTIL J = 0 DO
$(IF A GR CASEK ! J BREAK
CASEK ! (J + 1) := CASEK ! J
J := J - 1 $)

12. FINISH-Anweisung

Allgemeine Form:

FINISH

Wirkung:

Beim Erreichen einer FINISH-Anweisung wird der Programmlauf beendet.

Am statischen Programmende wird eine FINISH-Anweisung generiert und kann deshalb weggelassen werden.

Beispiele:

a) FOR Z = A TO S ! A DO
$(IF Z < 0 FINISH
FU ! Z := Z + Z
WRITE (6, "I7 /", 1, FU ! Z) $)
FINISH

```
b) TEST  A  THEN
            $(  WRITE (6 ,"/'KEIN ERGEBNIS' /", 0)
              FINISH  $)
            OR  GO  TO  ANF
```

13. RETURN-Anweisung

Allgemeine Form:

```
RETURN
```

Wirkung:

Mit RETURN erfolgt ein Rücksprung aus einer Routine in das aufrufende Programm. Das Sprungziel ist die dem Routine-Aufruf statisch folgende Anweisung.

Innerhalb einer Routine können mehrere RETURN-Anweisungen stehen. Das jeweils dynamisch durchlaufene RETURN veranlaßt den Rücksprung. Da am Ende jeder Routine ein RETURN generiert wird, kann die Anweisung an dieser Stelle entfallen.

Beispiele:

```
a) LET  MAP (F,X) BE                        //Rekursives Verfolgen
       $( IF X = 0  RETURN                  //einer Verweis-
          IF H1 ! X = S.COMMA  DO           //struktur
                $(  MAP ( F, H3 ! X)
                    F  (H2 ! X)
                       RETURN  $)
                       F (X)    $)

b) LET  STORE (LISTE, COUNT)  BE            // Einlesen von
       $(   READ (5, "A1", 1, LV  CH)       // Alphatext und
          IF 'A' GR CH GR 'Z' RETURN        // Eintragen der
            LISTE ! COUNT := CH             // auftretenden Buchstaben
                    RETURN  $)              // in eine Liste
```

14. RESULTIS-Anweisung

Allgemeine Form:

RESULTIS E

Bezeichnung:

E ist ein beliebiger Ausdruck.

Wirkung:

Die RESULTIS-Anweisung muß mindestens einmal in einem Ergebnisblock (VALOF-Block siehe G 4.7) stehen, und sie darf nur dort stehen.
Sie bewirkt, daß der Durchlauf des aktuellen innersten Ergebnisblockes abgebrochen wird und der R-Wert des in der RESULTIS-Anweisung stehenden Ausdrucks E als Resultat des Ergebnisblockes übergeben wird.
Es dürfen mehrere RESULTIS-Anweisungen in einem VALOF-Block stehen, die jeweils dynamisch zuerst durchlaufene kommt zu dieser Wirkung.

Beispiele:

```
a)  X := A / VALOF $( IF B = 0 $( FEHLER ("ZERODIVISION")
                                  RESULTIS 1 $)
                      RESULTIS B $)
```

In Abhängigkeit der Größe B wird entweder die Zuweisung
X := A / B (B ≠ 0)
oder die Zuweisung
X := A / 1 (B = 0) ausgeführt.

```
b)  FUNCT (P1, VALOF $( TEST P1 THEN RESULTIS P ! W
                        OR $( G(C); RESULTIS P ! (2 * W) $) $)
```

Der 2. Parameter des Prozeduraufrufes FUNCT ergibt sich in Abhängigkeit vom 1. Parameter entweder zu P ! W oder P ! (2 * W).

15. SWITCHON-Anweisung

Allgemeine Form:

SWITCHON E INTO ⟨block⟩

Bezeichnung:

E ist ein beliebiger Ausdruck

⟨block⟩ ist ein Block, der mindestens eine Marke folgender Form enthält:

CASE K :

DEFAULT :

K ist eine Konstante

Die Reihenfolge der Marken ist beliebig.

Wirkung:

Der R-Wert des Ausdruckes E wird berechnet. In Abhängigkeit des Wertes wird zu der CASE-Marke gesprungen, deren konstanter Wert mit dem des Ausdruckes E übereinstimmt. Ergibt sich bei keiner der CASE-Marken eine Übereinstimmung mit dem R-Wert von E, wird bei Existenz der DEFAULT-Marke dorthin gesprungen; existiert keine DEFAULT-Marke, wird hinter der SWITCHON-Anweisung fortgefahren.

Die der CASE-Marke folgenden Anweisungen werden durchlaufen bis eine Sprunganweisung auftritt. Eine CASE-Marke wirkt dabei nicht als Sprunganweisung, vielmehr mündet bei fehlender Sprunganweisung die Ausführung in den folgenden CASE-Zweig. Eine spezielle Anweisung ist die ENDCASE-Anweisung, die einen Sprung hinter den SWITCHON-Block bewirkt. Es ist möglich, innerhalb einer SWITCHON-Anweisung von einer CASE-Anweisungsfolge in eine andere genau so wie in andere Programmteile zu springen. Dies bietet sich an, wenn in beiden Fällen die gleiche Anweisungsfolge benötigt wird.

Allgemeine Form einer SWITCHON-Anweisung:

```
SWITCHON E INTO   $( [D]
CASE K1: [C] [ENDCASE]
CASE K2: [C] [ENDCASE]
   :
DEFAULT: [C] [ENDCASE]
CASE Km: [C] [ENDCASE]   $)
```

D sind Deklarationen

C ist eine Anweisung (-sfolge)

Regeln:

a) Die Konstanten der einzelnen CASE-Marken müssen verschieden sein.

b) Die DEFAULT-Marke darf höchstens einmal vorkommen.

c) Dynamische Variable im Deklarationsteil des SWITCHON-Blockes werden nicht initialisiert.

Beispiele:

```
a) MANIFEST   $( S.POS = 1; S.NEG = 3; ...   $)
    :
    LET CONT (P) BE   $(
    SWITCHON VAL ! P INTO    $(
     CASE S.NEG: ...
                 ...
                 FINISH
     CASE  S.POS: ...
                  ...
                 GO TO MPOS
     DEFAULT    :...
                 ...  $)
      .
MPOS: :
      .
      .
      .
      RETURN   $)
```

Stimmt der R-Wert von VAL ! P mit der Manifest-Konstanten S.NEG überein, wird das Programm nach der zugehörigen Anweisungsfolge beendet. Bei Übereinstimmung mit der Konstanten S.POS wird nach der Anweisungsfolge zu der außerhalb der SWITCHON-Anweisung liegenden Marke MPOS gesprungen. Ist der R-Wert von VAL ! P mit keiner der Marken identisch, erfolgt ein Sprung auf die Marke DEFAULT.
Weitere Beispiele sind unter F 16. zu finden.

16. ENDCASE-Anweisung

Allgemeine Form:

ENDCASE

Wirkung:

Die Anweisung ENDCASE ist nur in SWITCHON-Anweisungen zugelassen. Sie schließt eine zu einer der CASE-Marken gehörende Anweisungsfolge ab und veranlaßt, daß die Programmausführung hinter der schließenden Blockklammer des SWITCHON-Blockes fortgesetzt wird.

Regeln:

a) Vor der schließenden SWITCHON-Blockklammer kann die ENDCASE-Anweisung entfallen.

Beispiel:

```
LET  RECHNE (OP, OP1, OP2) = VALOF
$(A  LET  FS = NIL                          // In Abhängigkeit des
     SWITCHON  OP  INTO                     // Operationscodes OP
     $(SW  DEFAULT :  FS := "FALSCHER OPERATOR"
                      ENDCASE               // werden mit den Operanden
           CASE '-' :  OP2 := -OP2          // OP1 und OP2 Operatio-
           CASE '+' :  RESULTIS OP1 + OP2   // nen durchgeführt oder
           CASE '*' :  RESULTIS OP1 * OP2   // ein Fehlerkommentar
           CASE '/' :  UNLESS OP2 = 0 RESULTIS OP1 / OP2
                       FS := "ZERODIVIDE"   $)SW
                       FEHL (FS)        $)A // für einen folgenden
                                            // Fehlerroutineaufruf
                                            // gebildet.
```

AUSDRÜCKE

AUSDRÜCKE

1. Ausdrucksarten

BCPL-Ausdrücke sind Syntaxproduktionen, denen sich zur Objektzeit ein Wert zuordnen läßt (R-Wert); die Berechnung des Wertes kann Nebenwirkungen haben (Funktionsaufrufe usw.)
Gewisse Ausdrücke besitzen außer dem R-Wert auch einen L-Wert (Adresse), der selbst wieder R-Wert eines anderen Ausdrucks sein kann; auch die Berechnung des L-Wertes eines Ausdrucks kann Nebenwirkungen haben.
Der Kontext bestimmt, ob ein (Teil-) Ausdruck im R-Modus oder im L-Modus zu berechnen ist, d.h. ob sein R-Wert oder sein L-Wert (dessen Existenz in diesem Fall zu fordern ist) zu berechnen ist.
Während der Wert (R-Wert) eines Ausdrucks unabhängig vom syntaktischen Kontext ist, sind die Art der Berechnung und damit die auftretenden Nebenwirkungen kontextabhängig (Optimierung von Sprungbedingungen).
Ferner ist eine semantische Restriktion zu beachten:
Einige an sich (syntaktisch) zugelassene Ausdrücke sind semantisch undefiniert (d.h. R-Wert und Nebenwirkungen sind undefiniert → Vertauschung von Teilausdrücken). Kapitel G spezifiziert für die einzelnen syntaktischen Ausdrucksklassen Existenz und Algorithmus zur Berechnung von R-Wert und L-Wert (falls existent), sowie deren Nebenwirkungen, ferner syntaktische Einschränkungen.
Außerdem werden zwei semantische Ausdrucksklassen spezifiziert:
die Klassen der L-Ausdrücke und die der konstanten Ausdrücke.

a) Einfache Ausdrücke (siehe G 4.)

Das sind die am stärksten gebundenen (Teil-) Ausdrücke (d.h. von der höchsten Priorität), die als Komponenten von komplexeren Ausdrücken (zeitlich) zuerst berechnet werden.
Innerhalb von einfachen Ausdrücken wird von links nach rechts gerechnet z.B.

(F → G,H) ( )

b) Monadische Ausdrücke (siehe G 5.)

Ein monadischer Ausdruck besteht entweder aus einem monadischen Operator und einem Operanden, der selbst ein einfacher Ausdruck (Ausdruck höherer Priorität) ist, oder aus einem einfachen Ausdruck.

c) Dyadische Ausdrücke (siehe G 6.)

Ein dyadischer Ausdruck besteht aus einem dyadischen Operator, der zwei Operanden verknüpft, die selbst einfachere Ausdrücke sind (d.h. von gleicher bzw. höherer Priorität).

d) Bedingte Ausdrücke (siehe G 7.)

Diese Ausdrücke nehmen, in Abhängigkeit eines Bedingungsausdruckes, den Wert eines von zwei verschiedenen Ausdrücken an.

a) - d) beschreiben alle möglichen Ausdrücke.
Die zwei folgenden Ausdrucksklassen sind semantische Ausdrucks-Teilklassen.

e) Konstante Ausdrücke (siehe G 8.)

Die R-Werte dieser Ausdrücke werden zur Compilezeit berechnet. Der einmal zu diesem Zeitpunkt eingesetzte R-Wert kann zur Objektzeit nicht mehr verändert werden.

f) L-Ausdrücke (siehe G 9.)

Diese Ausdrücke besitzen einen L-Wert, d.h., ihnen ist die Adresse eines BCPL-Elements zugeordnet.

2. Priorität von Ausdrücken

Um in einem Ausdruck die Operanden eines Operators zu bestimmen, werden monadischen und dyadischen Operatoren und Ausdrücken Prioritäten zugeordnet, die eine Hierarchie der Operatoren und (Teil-) Ausdrücke festlegen. Operationen höherer Priorität werden vor solchen niedrigerer Priorität ausgeführt, solche gleicher Priorität werden von links nach rechts ausgeführt, sofern nicht durch Klammerung eine andere Reihenfolge festgelegt wird.

Monadische Operatoren, die selbst Operanden sind, werden vor der übergeordneten Operation ausgeführt. Dabei erstreckt sich der Geltungsbereich des monadischen Operators auf alle Operanden und Operationen, deren Priorität höher ist als die des monadischen Operators. Dyadische Operationen gleicher Priorität werden von links nach rechts ausgeführt.
Ausnahmen bilden die monadischen Operatoren TABLE und SLCT (siehe G 5.5 und G 5.6).

Für die meisten Ausdrücke werden im Rahmen der Syntaxerklärung mögliche Rechts- und Linksoperanden mit den folgenden Abkürzungen beschrieben:

- ⟨ ⟨P⟩ - Ausdruck⟩ steht für Ausdruck der Priorität ≧ ⟨P⟩
- ⟨ D ⟨P⟩ - Ausdruck⟩ steht für dyadischer Ausdruck der Priorität ≧ ⟨P⟩
- ⟨ M - Ausdruck⟩ steht für "Monadischer Ausdruck".
- ⟨D - L ⟩ steht für dyadischen Ausdruck von L-Typ
- ⟨D(P)-L⟩ steht für dyadischer Ausdruck von L-Typ der Priorität ≧ ⟨P⟩
- ⟨M - L ⟩ steht für monadischer Ausdruck von L-Typ

2.1. Hierarchietabelle für Operatoren

Monadische Operatoren	Dyadische Operatoren	Priorität	Zugehörige Ausdrücke
		10	Einfache Ausdrücke
	!	9	Vektorausdrücke
	:: | OF		Selektorausdrücke (Teilwortausdrücke)
RV | !	/	8	RV-Ausdrücke
LV | @	*		LV-Ausdrücke
	REM		
	+	7	Arithmetische Ausdrücke
	-		
+			Vorzeichen-Ausdrücke
-			
	LS | <	6	Vergleichs-Ausdrücke
	LE | < =		
	EQ | =		
	NE		
	GE | > =		
	GR | >		
	LSHIFT | <<		SHIFT-Ausdrücke
	RSHIFT | >>		
NOT		5	NOT-Ausdrücke
	LOGAND | /|	4	
	LOGOR | |/	3	Logische Ausdrücke
	EQV	2	
	NEQV		
	COND |— ~	1	
TABLE		0	Tables
SLCT			Selektoren

3. Beispiele:

a) V ! +A * B

Der linke Operand des Vektoroperators ! ist der einfache Ausdruck V (Priorität 10). Der rechte Operand von ! ist der monadische Ausdruck +A * B, denn das monadische + (Priorität 7) bindet den dyadischen Operator * (Priorität 8)

Der Ausdruck wird deshalb berechnet zu:

(V) ! (+(A * B))

Die geklammerte und die ungeklammerte Form obigen Ausdrucks führt, z.B. links von einem Multiplikationszeichen stehend, zu 2 verschiedenen Ausdrucksbearbeitungen:

V ! + A * B * C

Der ungeklammerte Ausdruck wird berechnet wie bereits oben angegeben. Der monadische Operator "+" bezieht sich nun auf das Produkt A * B * C.

Der Ausdruck wird berechnet zu:

(V) ! (+(A * B * C))

Die Berechnung mit dem geklammerten Ausdruck verläuft anders:

((V) ! (+(A * B))) * C

b) V ! RV A * B

Auf Rechtsoperandenposition des dyadischen Operators " ! " steht der monadische Operator RV (Priorität 8). Die monadische Operation wird zuerst ausgeführt, erstreckt sich aber nur auf den einfachen Ausdruck A, da wegen gleicher Priorität des folgenden Multiplikationszeichens nur A und nicht A * B Operand des RV-Operators ist. Nach Ausführung der monadischen Operation, können wieder die üblichen Prioritätsregeln für dyadische Operatoren angewandt werden.

Der Ausdruck wird berechnet zu:

(V ! (RV A)) * B

c) V * RV B ! A REM - C + D

Zwei monadische Operatoren stehen auf Rechtsoperandenposition eines dyadischen Operators: RV und das Minuszeichen. Der RV-Operator (Priorität 8) bindet den Vektoroperator (Priorität 9); das monadische Minuszeichen (Priorität 7) wirkt wegen gleicher Priorität des folgenden Operators nur auf den einfachen Ausdruck C.
Für die restlichen dyadischen Operatoren gelten wieder die üblichen Prioritätsregeln.

Der Ausdruck wird berechnet zu:

((V * (RV (B ! A)))REM (- C)) + D

Anmerkung:

Bei arithmetischen Ausdrücken ergeben sich die üblichen Bindungsregeln.

4. Einfache Ausdrücke

Die Klasse der einfachen Ausdrücke wird gebildet durch:

4.1. Namen

Namen sind BCPL Grundsymbole

Syntax:

Ein Name beginnt stets mit einem Buchstaben, dem in beliebiger Folge Buchstaben, Ziffern und Punkte folgen können
Ein Name darf nicht mit einem Schlüsselwort identisch sein.

Beschreibung:

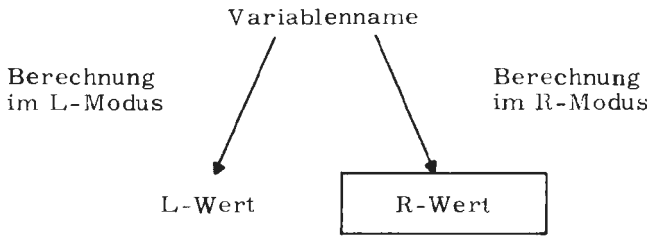
Manche Namen werden verwandt als Benennung für:

a) Konstante

Durch die Deklaration von MANIFEST-Konstanten (siehe E 2.) werden Namen direkt Konstanten (R-Werten) zugeordnet. Ihre Berechnung kann nur im R-Modus erfolgen.

b) Variable

Jede andere Deklaration erklärt Namen zu Variablennamen.
Eine Variable definiert die Zuordnung eines BCPL-Elements zu einem Namen. Sie besitzt deshalb einen R-Wert (Inhalt) und einen L-Wert (Adresse) und kann somit sowohl im R-Modus als auch im L-Modus berechnet werden.
Das Ergebnis der Berechnung im R-Modus ist ihr R-Wert (Inhalt), das Ergebnis der Berechnung im L-Modus ist ihr L-Wert (Adresse).



Das Ergebnis für beide Berechnungsmodi ist ein Bitmuster von der Länge eines BCPL-Elements.

Regel:

Ein Name darf max. 50 Zeichen lang sein.

Beispiele:

```
LL11.A   S.1..A   ALPHA 9
 S.BEGIN
```

4.2. Zahlen

Zahlen sind BCPL-Grundsymbole
Es gibt drei Arten von Zahlendarstellung:

- Dezimalzahlen
- Oktalzahlen
- Sedezimale Zahlen.

Syntax:

⟨Dez.Zahl⟩ ::= ⟨Ziffer⟩ | ⟨Dez.Zahl⟩ ⟨Ziffer⟩

⟨Oktalzahl⟩ ::= $8 ⟨Oktalziffer⟩ | ⟨Oktalzahl⟩ ⟨Oktalziffer⟩

⟨Sedez.Zahl⟩ ::= $H ⟨Sedez.Ziffer⟩ | ⟨Sedez.Zahl⟩ ⟨Sedez.Ziffer⟩

Beschreibung:

Zahlen sind R-Ausdrücke und können nur im R-Modus berechnet werden.

Der R-Wert von Dezimalzahlen ist die Dualzahl gleichen Wertes.

Jede Oktalziffer repräsentiert 3 Bits (Triade).

Der R-Wert von Oktalzahlen, ergibt sich aus der rechtsbündigen Ablage der Triaden.

Jede sedezimale Ziffer stellt 4 Bits (Tetrade) dar.

Der R-Wert von sedezimalen Zahlen ergibt sich aus der rechtsbündigen Ablage der Tetraden.

Implementierung TR 440:

a) Der R-Wert von Dezimalzahlen muß betragsmäßig kleiner als $2^{23}-1 = 8388607$ sein. Operanden und Ergebnisse einer mathematischen Operation müssen betragsmäßig $\leq 2^{22}-1 = 4\,194\,303$ sein.

b) Oktalzahlen dürfen nicht länger als 8 Oktalziffern sein (= 24 Bit).

c) Sedezimale Zahlen dürfen nicht länger als 6 sedezimale Ziffern sein (= 24 Bit).

d) Der R-Wert der dezimalen 0 ist stets die negative 0 (alle Bits gesetzt). Dabei ist es gleichgültig, wie sie erzeugt wurde.

Beispiele:

Dezimalzahlen: 191 999999

Oktalzahlen : interne Darstellung:

$87172	...	LLL	OOL	LLL	OLO
&817		...		OOL	LLL

Sedezimale Zahlen:

$HAABBC7	...	LOLO	LOLO	LOLL	LOLL	LLOO	OLLL
&HDEF1			...	LLOL	LLLO	LLLL	OOOL

4.3. String-Konstante (Strings)

String-Konstante sind BCPL-Grundsymbole

Syntax:

⟨Stringkonstante⟩ ::= " [⟨Alphazeichen⟩]$_{0}^{255}$ "

' [⟨Alphazeichen⟩]$_{2}^{256}$ '

Alphazeichen sind alle Zeichen des Zentralcodes.

Beschreibung:

Eine Stringkonstante besitzt wie jede andere Konstante nur einen R-Wert und kann so nur im R-Modus berechnet werden.

Der R-Wert verweist auf eine Folge von BCPL-Elementen, die den String als Zeichenkette enthalten. Das erste Zeichen gibt die Länge des Strings in Zeichen an; das ist die Anzahl der Alphazeichen zwischen den Stringklammern ' bzw. '' nach Abzug des Ersetzungsoperators (siehe unten).

Implementierung TR 440:

a) Je drei aufeinanderfolgende Zeichen des Strings werden in einem BCPL-Element gespeichert.

b) Ist das letzte BCPL-Element (TR 440 Halbwort) nicht vollständig mit Stringzeichen belegt, wird es zum rechten Rand mit Ignore-Oktaden (Null-Oktaden) aufgefüllt.

c) Die Alphazeichen der Stringkonstanten werden im Zentralcode abgespeichert.

Das Zentralcodezeichen '*' dient als Ersetzungsoperator

*N	wird ersetzt durch NL (Newline) das Steuerzeichen 'neue Zeile' ist sonst nicht direkt darstellbar.
*S	wird ersetzt durch SP (Space) *S ist identisch mit einem in der Stringkonstanten auftretenden Blank
* ⟨Alphazeichen⟩	wird ersetzt durch das angegebene Alphazeichen selbst.

Insbesondere muß demnach ein Stern im String dargestellt werden durch: **.

Mit Hilfe des Ersetzungsoperators können auch die Zeichen ' und " dargestellt werden. Diese Zeichen sind sonst innerhalb der Stringkonstanten verboten, da sie die Stringkonstante vorzeitig beenden würden.

Regel:

Enthält ein String mindestens zwei Zeichen, kann er in einfache Apostrophs eingeschlossen werden.

Beispiele:

a) die Stringkonstante:

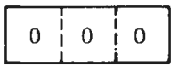
'*NNEUE␣␣ZEILE'

verweist auf die nachstehende Folge von BCPL-Elementen

b) die Stringkonstante:

"" (leerer String, nur bei Doppelapostroph erlaubt)

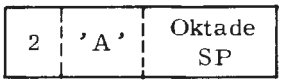
verweist auf das BCPL-Element



c) die Stringkonstante:

"A*S"

verweist auf das BCPL-Element



4.4. Zeichen-Konstante

Zeichen-Konstante sind BCPL-Grundsymbole.

Syntax:

'⟨Alphazeichen⟩' | '*⟨Alphazeichen⟩'

Beschreibung:

Eine Zeichenkonstante besitzt wie jede andere Konstante nur einen R-Wert.

"*" hat dieselbe Bedeutung als Ersetzungszeichen wie in Stringkonstanten (siehe G 4.3).

Implementierung TR 440: Der R-Wert des Alphazeichens ist der Binärwert der Zentralcodeoktade

Beispiele:

'*N' ''' '**' 'A' 'Y'

4.5. Wahrheitswerte

Syntax:

TRUE | FALSE

Beschreibung:

TRUE und FALSE besitzen nur R-Werte.
Der R-Wert von FALSE ist ein Bitmuster von lauter Nullen; der von TRUE ist ein Bitmuster von lauter Einsen.

Implementierung TR 440: Im TR 440 haben TRUE und der R-Wert 0 dieselbe Darstellung.

4.6. Geklammerte Ausdrücke

Syntax:

(⟨Ausdruck⟩)

Beschreibung:

Ausdrücke werden geklammert, um die fest vorgegebene Priorität der Ausdrucksabarbeitung (siehe Hierarchietabelle G 2.1) zu ändern. Geklammerte Ausdrücke (Priorität = 10) werden stets zuerst berechnet. Ohne Berücksichtigung des übrigen Ausdruckes gilt für den geklammerten und denselben ungeklammerten Ausdruck Identität bezüglich der Art seiner Berechnung seines Wertes und gegebenenfalls seines L-Wertes.

Beispiele:

a) A ! J + Z1 * P/P1 REM P - 1

Ohne Klammerung von Teilausdrücken wird obiger Ausdruck berechnet zu:

(A ! J) + (Z1 * P/P1 REM P) - 1

Abarbeitung von links nach rechts wegen gleicher Priorität der Operatoren

Folgende Ausdrucksabarbeitung kann, entgegen der vorgegebenen Priorität, durch Klammerung von Teilausdrücken gefordert werden:

```
A ! (J + Z1) * P/(P1  REM (P - 1) )
```

4.7. Ergebnis-Blöcke (VALOF-Blöcke)

Syntax:

```
VALOF    (block)
```

Beschreibung:

Auf Ausdrucksposition können VALOF-Blöcke stehen. Die in diesem Block stehenden Anweisungen werden bis zum Erreichen einer RESULTIS-Anweisung ausgeführt. Der bei dieser RESULTIS-Anweisung stehende Ausdruck bestimmt den Wert des VALOF-Blockes. Es sind mehrere RESULTIS-Anweisungen in einem Block möglich.

Beispiele:

a)
```
V ! VALOF   $( IF I< 0   $( G(I); RESULTIS  -I   $)
                            RESULTIS I   $)
```

Der Wert des rechten Vektoroperanden ergibt sich in Abhängigkeit seines Vorzeichens entweder zu I oder zu -I.

b)
```
A / VALOF   $(  UNLESS B = 0  RESULTIS   B
                FEHLERMELD ("ZERODIVISION")
                RESULTIS 1     $)
```

c)
```
FEHLTEXT := VALOF   $( SWITCHON  FS  INTO
                       $(  DEFAULT : RESULTIS "UNBEKANNTER"
                                     FEHLERSCHLÜSSEL"
                        CASE 1 :  RESULTIS "NAME NICHT
                                     DEFINIERT"
                        CASE 2 :  RESULTIS "FEHLERHAFTE
                                     ANWEISUNG"
                        CASE 3 :  ...
                        :                    $)
                                        $)
```

d) VALOF $(I := FALT
FALT := FNEU
FNEU := I + F(I)
RESULTIS FNEU < FALT $)

4.8. Funktionsaufrufe

Syntax:

⟨einfacher Ausdruck⟩ (⟨Ausdrucksliste⟩)

Die Liste der Ausdrücke kann leer sein. Listentrenner ist Komma.

Beschreibung:

Sind Ausdrücke in der Liste vorhanden, liefern sie die aktuellen Werte für die Formalparameter (Ausdrucksliste siehe G 10).
Ergebnis des Funktionsaufrufes ist ein R-Wert, der innerhalb des Funktionsrumpfes ermittelt wird.

Es wird die Funktion aufgerufen, deren Programmadresse mit dem R-Wert des einfachen Ausdrucks übereinstimmt.
Im Normalfall besteht der einfache Ausdruck aus dem Namen einer Funktionsvariablen (Funktionsname), die durch eine Funktionsdefinition definiert und mit der Programmadresse der Funktion initialisiert wurde. (siehe E 1.3).

Beim Aufruf von nichtrekursiven Funktionen sind als einfache Ausdrücke nur Namen zugelassen. Zusätzlich muß zwischen Namen und Funktionsklasse dieselbe Zuordnung getroffen sein, wie bei der Funktionsdeklaration.

Bemerkung:

Da der syntaktische Aufbau von Routine- und Funktionsaufrufen der gleiche ist, kann erst aus dem syntaktischen Zusammenhang auf die Art des Aufrufes geschlossen werden.
Wird eine Funktion als Routine aufgerufen, geht das Funktionsergebnis verloren.
Wird eine Routine als Funktion aufgerufen, ist das übergebene Funktionsergebnis undefiniert.

Beispiele:

a) F (A, LV B)

b) FUNCT ( ) // (leere Aufrufliste)

c) CAS (I, V ! I, $ 8 733, 99 + K)

d) (OP = '+' → OP. PLUS, OP = '-' → OP. MINUS, FEHLER) (OP1, OP2)

Abhängig vom R-Wert der Variablen OP wird die Routine

OP. PLUS (OP1, OP2)
OP. MINUS (OP1, OP2)
oder FEHLER (OP1, OP2) aufgerufen.

e) NONREC 11 : F, H
LET F (N) BE $(... $)
:
:
H : = F // zulässig; da auch H die NONREC-
H (7) // Klasse 11 zugeordnet wurde
:
:
(Z → H, F) (1) // verboten; da auf Position der
// Funktionsvariablen kein Name,
sondern ein Ausdruck steht.

5. Monadische Ausdrücke

Die Gruppe der monadischen Ausdrücke wird gebildet durch:

Ausdruck	Operator	Priorität	Beschreibung in
- RV-Ausdrücke	RV	8	5.1
- LV-Ausdrücke	LV	8	5.2
- Ausdrücke mit Vorzeichen	+ | -	7	5.3
- NOT-Ausdrücke	NOT	5	5.4
- Selektoren	SLCT	0	5.5
- tables	TABLE	0	5.6

Monadische Ausdrücke bestehen aus einem einfachen Ausdruck oder einem monadischen Operator mit zugehörigem Operanden. Der Operand seinerseits kann wieder ein monadischer Ausdruck sein oder ein dyadischer Ausdruck (siehe G 6.), dessen Priorität größer als die des monadischen Operators ist.

Für die Syntaxerklärung der monadischen Ausdrücke sei folgende Begriffserklärung vorausgeschickt:

⟨M-Ausdruck⟩ :: = ⟨Einfacher Ausdruck⟩ | ⟨i-monop⟩ ⟨M-Ausdruck⟩ |
⟨i-monop⟩ ⟨D(i+1) Ausdruck⟩

⟨i-monop⟩ steht für "Monadischer Operator der Priorität ≥ i" (siehe G 2).

Beachte:

Die monadischen Operatoren TABLE (siehe G 5.6) und SLCT (siehe G 5.5) werden gesondert behandelt.

5.1. RV-Ausdrücke

Der monadische Operator RV hat die Priorität 8.

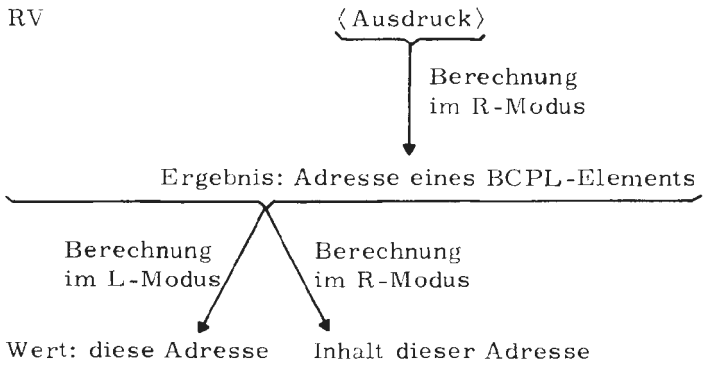
Syntax:

RV-Ausdruck :: = RV ⟨D(9)-Ausdruck⟩ | RV ⟨M-Ausdruck⟩

Beschreibung:

Der Operand des RV-Operators wird im R-Modus berechnet; das Ergebnis ist ein Bitmuster, das als Adresse eines BCPL-Elements aufgefaßt wird.

Wird im L-Modus gerechnet, ist diese Adresse das Ergebnis. Wird im R-Modus gerechnet, ist das Ergebnis der Inhalt (R-Wert) der durch diese Adresse bezeichneten Speicherzelle.



Beispiele:

a) X := RV V ! I

Der Inhalt von V ! I wird als Adresse eines BCPL-Elementes aufgefaßt. Da der RV-Ausdruck auf der rechten Seite der Zuweisung im R-Modus berechnet wird, ist das Ergebnis dieses Ausdrucks der Inhalt der so adressierten Speicherzelle.
Eine Klammerung des Vektorausdrucks V ! I erübrigt sich, da die Priorität von V ! I größer ist.

b) RV X := V ! I

Der R-Wert von X wird als Adresse eines BCPL-Elementes aufgefaßt. Da der RV-Ausdruck auf der linken Seite einer Zuweisung im L-Modus berechnet wird, ist das Ergebnis diese Adresse.

c) X := RV RV V ! I

Der Ausdruck wird berechnet zu:

X := RV (RV (V ! I))

Beispiel a) wurde dahingehend erweitert, daß das dortige Ergebnis (Inhalt der durch V ! I bezeichneten Speicherzelle) wiederum als Adresse einer Speicherzelle aufgefaßt wird und deren Inhalt nun das Ergebnis ist.

5.2. LV-Ausdrücke

Der monadische Operator LV hat die Priorität 8.
Sein Operand muß ein L-Ausdruck sein (siehe G 9).

Syntax:

⟨LV-Ausdruck⟩ ::= LV ⟨D9-L-Ausdruck⟩ |
LV ⟨M-L-Ausdruck⟩

Beschreibung:

Dem Operanden des LV-Operators muß die Adresse eines BCPL-Elements zugeordnet sein, da er im L-Modus berechnet wird. Der R-Wert des LV-Ausdruckes ist die Adresse dieses BCPL-Elements.
Der LV-Ausdruck selbst ist nicht vom L-Typ, d.h., ihm ist keine Adresse zugeordnet; er kann nicht im L-Modus berechnet werden.

Beispiele:

a) ADR := LV V ! I

In die mit ADR benannte Speicherzelle wird die Adresse des Ausdrucks V ! I (Vektorelement) gespeichert.

b) LV ADR := LV V ! I

Dieses Beispiel ist unzulässig; da die linke Seite einer Zuweisung bereits im L-Modus berechnet wird und die so ermittelte Adresse keinen L-Wert besitzt.

c) F(A, B, LV C)

Statt der sonst bei Prozedurparametern üblichen value-Übergabe (R-Wert-Übergabe) wird für den dritten Parameter die Adresse der Variablen C übergeben.

Weitere Beispiele siehe C 9.9.

5.3. Vorzeichen-Ausdrücke

Das monadische + und das monadische - haben die Priorität 7.

Syntax:

⟨Vorzeichen-Ausdruck⟩ ::= + ⟨D8-Ausdruck⟩ |
+ ⟨M-Ausdruck⟩ |
- ⟨D8-Ausdruck⟩ |
- ⟨M-Ausdruck⟩

Beschreibung:

Für die monadischen Operatoren + und - gelten die üblichen arithmetischen Regeln. Der R-Wert eines negativen Ausdruckes ist das arithmetische Inverse des beteiligten Operanden.
Der L-Wert eines '+' Ausdruckes ist der L-Wert des Operanden. Der '-' Ausdruck hat keinen L-Wert.

Implementierung TR 440: Sind die zu invertierenden R-Werte entweder +0 (kein Bit gesetzt) oder -0 (alle Bits gesetzt) findet keine Invertierung statt. Die R-Werte bleiben erhalten.

Beispiele:

a) + X / Y * Z wird berechnet zu
+ (X / Y * Z)

b) - A REM B + C * D wird berechnet zu
(- A REM B) + (C * D)

5.4. NOT-Ausdrücke

Der NOT-Operator hat die Priorität 5.

Syntax:

⟨NOT-Ausdruck⟩ :: = NOT ⟨D6-Ausdruck⟩ |
NOT ⟨M-Ausdruck⟩

Beschreibung:

Der R-Wert des NOT-Ausdruckes ist das invertierte Bitmuster des Operanden.

Beispiele:

a) B : = NOT B

b) IF NOT C DO E : = D

c) $(V ! I = F(I)
I: = I+1
BOOL : = F.LOGOS(I) $) REPEATUNTIL NOT BOOL

5.5. Selektoren

Der Operator SLCT hat die Priorität 0.

Syntax:

⟨Selektor⟩ :: = SLCT K1 [: K2 [: K3]]

Die Teiloperanden K1, K2 und K3 sind konstante Ausdrücke (siehe G 8).

Beschreibung:

Mit Hilfe eines Selektors ist es möglich, auf auswählbare Bitgruppen eines BCPL-Elementes zuzugreifen, um damit Bit (-feld) manipulationen vorzunehmen.

Die dazu nötigen Angaben werden durch drei Teiloperanden wie folgt beschrieben.

- Bitfeldlänge : K1
- Shift : K2
- Relativadresse : K3 (in BCPL-Elementen)

Regeln:

a) Werden K2 und K3 nicht angegeben, so wird K2 =0 und K3 = 0 eingesetzt.

b) K1, K2 und K3 müssen ≧ 0 sein.

Implementierung TR 440: a) Bitfeldlänge + Shift müssen ≦ 24 sein.
(Bitanzahl pro BCPL-Element)
b) Die Relativadresse muß ≦ 1023 sein.

Zur Ausführung von Selektor-Ausdrücken und Teilwortzuweisungen siehe F 1. 2. Dort sind auch ausführliche Beispiele zu finden.

Beispiele:

a) MANIFEST $(MASKE = SLCT 10:3:4 $)
MASKE OF LV VAR := MASKE OF POINT

b) MANIFEST $(TEIL = SLCT 3 $)
TEIL OF V1 := (SLCT 3:1) OF V1

5.6. Tables

Der Operator TABLE hat die Priorität 0.

Syntax:

⟨table⟩ ::= TABLE ⟨Adressen-Konstanten-Liste⟩
⟨Adressen-Konstanten-Liste⟩ vgl. G 10

Die Adressen-Konstanten-Liste (siehe auch G 10.) kann Konstante, Strings und Tables enthalten. Jedes dieser Listenelemente kann mit einem Replikator von der Form REP n (n = Wiederholungsfaktor) versehen werden. Dieser Replikator bezieht sich auf die jeweils letzte vor ihm stehende Konstante bzw. den letzten Klammerausdruck.

Stehen weitere Tables auf Listenposition, ist durch Klammerung eine eindeutige Abgrenzung zu anderen Listenelementen zu schaffen.

Beschreibung:

Mit dem TABLE-Operator lassen sich aufeinanderfolgende statische (siehe D 4.3) BCPL-Elemente vorbesetzen. Der R-Wert des TABLE ist ein Zeiger, der auf den Anfang dieses Speicherbereiches verweist. Verschachtelte Tables (TABLE's auf Listenposition) werden verwandt, um Listenstrukturen vorzubesetzen. Für jedes weitere TABLE auf Listenposition wird eine neue Liste angelegt, die mittels Verweis im übergeordneten TABLE angesprochen wird (siehe Beispiele).

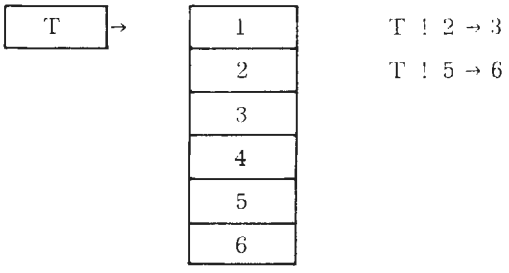
Wird ein Table in einer Adressen-Konstante-Liste durch REP n spezifiziert, so werden n Listen angelegt und die Verweise in diese Listen in n aufeinanderfolgende Listenelementen der übergeordneten Liste eingetragen.
Der Zugriff auf die einzelnen Table-Elemente ist der gleiche wie bei Vektorelementen.

Während bei rekursiven Aufrufen für Vektoren pro Aufruf eine neue Generation Vektorelemente angelegt wird (Vektoren → dynamisch) wird im gleichen Fall für Tables nur eine Generation angelegt (Tables → statisch).

Beispiele:

a) LET T = TABLE 1, 2, 3, 4, 5, 6

Die dynamische Variable T wird mit der Adresse der (statischen) Liste initialisiert.
Die Listenelemente können durch einen Vektorausdruck angesprochen werden.



b) ZI := (TABLE '0', '1', '2', '3') ! I

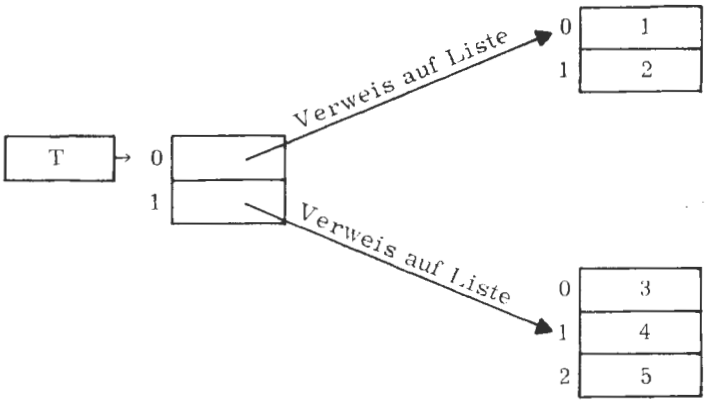
für I = 3 enthält die mit dem Namen ZI bezeichnete Variable nach der Zuweisung rechtsbündig die Oktade '3'.

c) STATIC $(US = TABLE 'A', 'B', ... $) // Umschlüsseltabelle

Die statische Variable US wird mit der Adresse der (statischen) Liste initialisiert.

c) LET T = TABLE (TABLE 1, 2), (TABLE 3, 4, 5)

Das auf Listenelementposition stehende TABLE veranlaßt, daß ein Verweis auf eine weitere Liste angelegt wird. Diese Liste enthält die zu diesem TABLE gehörenden Werte.

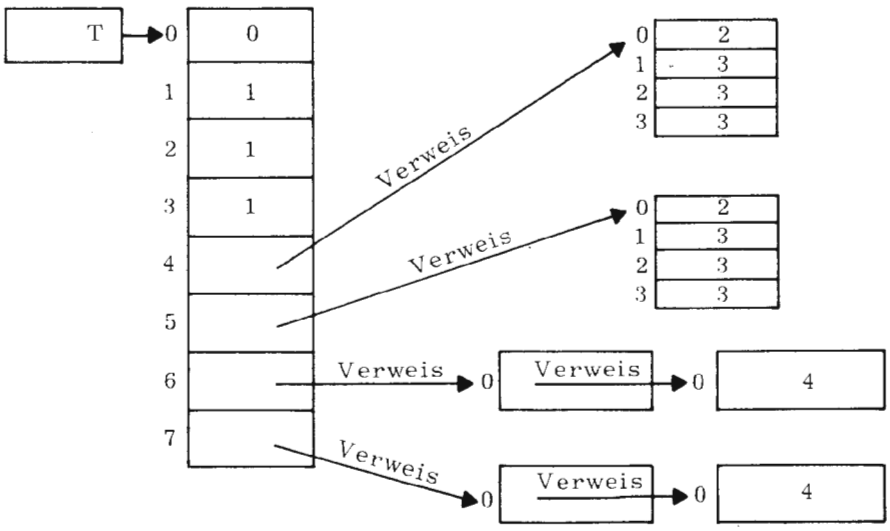


T ! 0 enthält einen Verweis auf die Liste, die die Zahlen 1 und 2 enthält.
T ! 1 enthält einen Verweis auf die Liste, die die Zahlen 3, 4 und 5 enthält.

Für die direkte Adressierung der einzelnen Table-Elemente können folgende Vektorausdrücke verwandt werden.

T ! 0 ! 0 = 1
T ! 0 ! 1 = 2
T ! 1 ! 0 = 3
T ! 1 ! 1 = 4

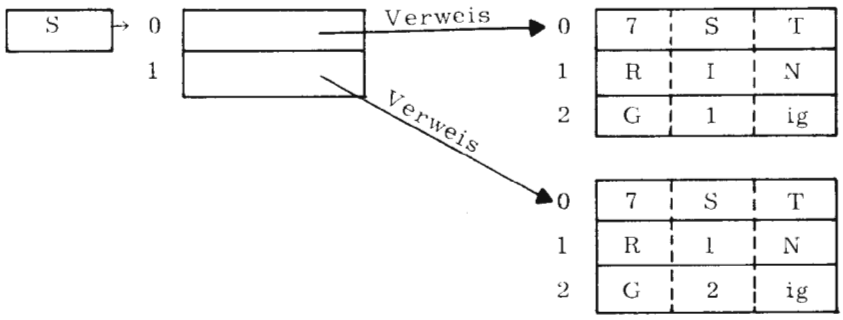
d) T := TABLE 0, 1 REP 3, (TABLE 2, 3 REP 3) REP 2,
(TABLE (TABLE 4)) REP 2



T ! 0 = 0
T ! 1 = 1
T ! 2 = 1
T ! 3 = 1
T ! 4 ! 0 = 2
T ! 4 ! 1 = 3
T ! 4 ! 2 = 3
T ! 4 ! 3 = 3
T ! 5 ! 0 = 2
T ! 5 ! 1 = 3
T ! 5 ! 2 = 3
T ! 5 ! 3 = 3
T ! 6 ! 0 ! 0 = 4
T ! 7 ! 0 ! 0 = 4

e) LET S = TABLE "STRING1", "STRING2"

Da der R-Wert einer Stringkonstanten ein Zeiger auf die einzelnen , die Stringkonstante enthaltende BCPL-Element ist, enthält das Table nicht die Stringkonstanten selbst, sondern nur die Verweise darauf.



6. Dyadische Ausdrücke

Die Gruppe der dyadischen Ausdrücke wird gebildet durch:

Ausdruck	Operator	Priorität	
Vektorausdrücke	!	9	6.1
Selektorausdrücke	OF	9	6.2
(Teilwortausdrücke)			
Arithmet. Ausdrücke	/, *, REM	8	6.3
	+, -	7	
Vergleichsausdrücke	LS, LE, EQ, NE, GE, GR	6	6.4
Shiftausdrücke	LSHIFT, RSHIFT	6	6.5
(Dyadische) logische	LOGAND	4	6.6
	LOGOR	3	
Ausdrücke	EQV, NEQV	2	

Dyadische Ausdrücke bestehen aus einem dyadischen Operator und seinen beiden Operanden.

Bemerkung zur Syntaxbeschreibung:

Will man die Abarbeitung eines Ausdrucks bestimmen, so erkennt man die implizite Klammerung unter Beachtung der Prioritäten, indem man den Ausdruck von links beginnend analysiert.

Die Syntaxbeschreibung dient einerseits dazu, um bei einem Ausdruck einfach nachzuprüfen, ob die implizite Klammerung in der vorgestellten Weise durchgeführt wird, andererseits zur besseren Darstellbarkeit.

Syntax:

⟨Dyadischer Ausdruck⟩ ::=

$$\left\{\begin{array}{l}\langle\ \text{D(i)-Ausdruck}\rangle\\ \langle\ \text{M(i)-Ausdruck}\rangle\end{array}\right\}\langle\text{i-dyop}\rangle\left\{\begin{array}{l}\langle\ \text{D(i+1)-Ausdruck}\rangle\\ \langle\ \text{M-Ausdruck}\rangle\end{array}\right\}$$

⟨ i-dyop⟩ bedeutet:

"Dyadischer Operator der Priorität i"

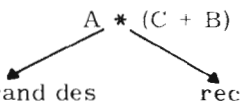
Beschreibung:

Es werden zuerst die beiden Operanden berechnet, dann wird die dyadische Operation ausgeführt.

Beispiele:

a) A * C + B

Annahme: implizite Klammerung sei

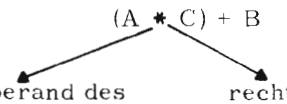


linker Operand des dyadischen Operators ist A (= D(10)-Ausdruck)

rechter Operand ist C + B (= D(7)-Ausdruck)

Annahme ist falsch, da rechter Operand kein ‹D(i+1)-Ausdruck› (≙⟨D-9⟩ Ausdruck) ist.

Annahme: implizite Klammerung sei



linker Operand des dyadischen Operators ist A * C (= D(8)-Ausdruck)

rechter Operand ist B (= D(10)-Ausdruck)

Annahme ist richtig, da die Priorität des dyadischen Operators und der Operandenausdrücke der Syntaxbeschreibung genügen.

b) A - B + C *(D - E)

- und + haben dieselbe Priorität, deshalb erfolgt die Abarbeitung von links nach rechts und (A - B) wird linker Operand von +. Der rechte Operand ergibt sich zu C *(D - E), denn die Operation * hat die höhere Priorität als +.

6.1. Vektorausdrücke

Der dyadische Operator ! (Vektoroperator) hat die Priorität 9.

Syntax:

⟨Vektorausdruck⟩ ::= { ⟨D(9)-Ausdruck⟩ / ⟨M(9)-Ausdruck⟩ } ! { ⟨D(10)- Ausdruck⟩ / ⟨M- Ausdruck⟩ }

Beschreibung:

Im allgemeinen werden Vektorausdrücke dazu benutzt, nur BCPL-Elemente zu adressieren, deren Anfangsadresse zuvor berechnet wurde (z.B. durch VEC ..., LV ..., TABLE).

Der R-Wert des linken Vektoroperanden ist ein Zeiger auf den Anfang einer Reihe aufeinanderfolgender Speicherzellen. Der R-Wert des anderen Vektoroperanden ist dann eine Relativadresse bezogen auf die Vektoranfangsadresse.

Beispiele:

a) B := V ! 3

Der Inhalt des Vektorelementes mit der Relativadresse 3 (4. BCPL-Element) und der zugehörigen Vektoranfangsadresse V (R-Wert) ersetzt den Inhalt des mit B bezeichneten BCPL-Elementes.

b) A ! B ! C wird berechnet zu
(A ! B) ! C da Operationen gleicher Priorität von links nach rechts abgearbeitet werden.

c)
```
LET T = TABLE 'A', 'B', 'C', 'D'
:
:
ZEICH := T ! I
```

Die durch TABLE definierten und vorbesetzten BCPL-Elemente werden mit dem Vektorausdruck T ! I angesprochen.
Für I = 3 wird die Speicherzelle mit der rechtsbündigen Oktade 'D' (4. Zelle wegen Adressierungsbeginn bei 0) angesprochen.

6.2. Selektorausdrücke (Teilwortausdrücke)

Der Selektor- oder Teilwortoperator OF ist ein dyadischer Operator der Priorität 9.
Äquivalentes Zeichen des Operator 'OF' ist '::' .

Syntax:

⟨Selektorausdruck⟩ ::= ⟨9 - Const⟩ OF { ⟨einfacher Ausdruck⟩ | ⟨M - Ausdruck⟩ }

⟨9 - Const⟩ ist ein konstanter Ausdruck (siehe E 8.) dessen Priorität (als Ausdruck) ≧ 9 ist.
Im allgemeinen ist der konstante Ausdruck eine MANIFEST-Konstante, deren Wert ein Selektor ist.

Beschreibung:

Mit Hilfe des Selektor-Operators können Teilwortoperationen ausgeführt, bzw. Bitgruppen verändert werden.

Da Bitgruppen innerhalb eines BCPL-Elementes keine eigene Adresse besitzen, kann auch ein Selektorausdruck keinen L-Wert haben.

Trotzdem kann ein Selektorausdruck auf der linken Seite einer Zuweisung stehen (siehe Teilwortzuweisung F 12).

Der konstante Operand des Selektorausdruckes bestimmt das angesprochene Bitfeld.

Er muß entweder direkt als Selektor angegeben werden oder als MANIFEST-Konstante, die als Selektor definiert ist.

Eine allgemeine Erklärung für den R-Wert-Berechnung des Selektorausdruckes wird für folgendes Beispiel gegeben:

----: = S OF E

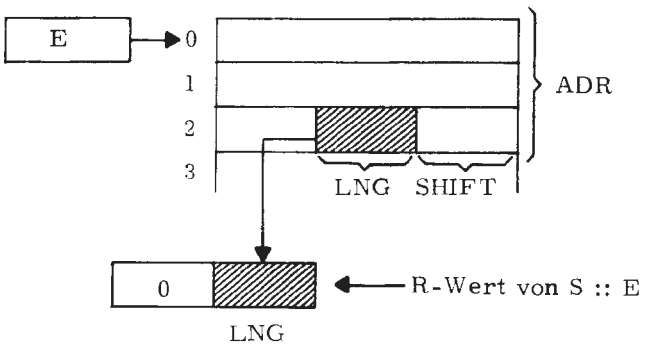
S ist ein Selektor und wurde in einer MANIFEST-Deklaration definiert als S = SLCT LNG : SHIFT : ADR

LNG, SHIFT und ADR sind Konstanten.

E ist ein Ausdruck. Der R-Wert dieses Ausdruckes wird als Anfangsadresse einer Reihe aufeinanderfolgender BCPL-Elemente interpretiert.

Die zu bearbeitende Bitgruppe liegt im BCPL-Element E ! ADR.

Aus diesem BCPL-Element werden jene LNG Bits, die vom rechten Rand des BCPL-Elements den Abstand SHIFT haben, geholt und nach links auf BCPL-Elementgröße mit Nullbits aufgefüllt.



Die verschiedenen Zuweisungsmöglichkeiten mit Selektorausdrücken sind unter D 1.1 nachzulesen.

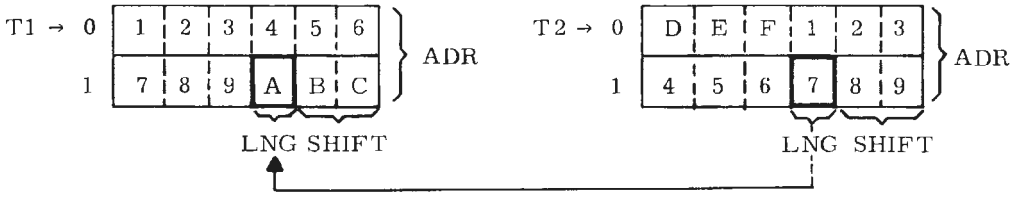
Achtung:

Die arithmetische 0 wird bei Teilwortzuweisungen nicht als 0 erkannt.

Beispiele:

a)
```
MANIFEST  $(  M = SLCT  4:8:1,  $)
LET  T1 = TABLE  $H123456,  $H789ABC
LET  T2 = TABLE  $HDEF123,  $H456789

M :: T1 := M :: T2
```



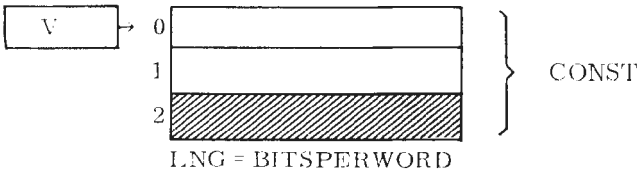

```
T1 ! 1 = $H7897BC
```

b)
```
LET  ROW = TABLE  $877776666, $876543210
(  SLCT  3  ) :: ROW := ( SLCT 3:0:1 ) :: ROW
```

c)
```
MANIFEST  $(  BITSPERWORD = 24  $)
( SLCT  BITSPERWORD : 0 :  CONST)  OF  V
```

'CONST' ist eine Konstante = 1023

Obiger Selektorausdruck wirkt wie V ! CONST



6.3. Arithmetische Ausdrücke

Die arithmetischen Operatoren / , * , REM (Divisions-, Multiplikations-Restoperator) haben die Priorität 8; die arithmetischen Operatoren + , - (Additions-, Subtraktionsoperator) haben die Priorität 7.

Syntax:

⟨ arithmetischer Ausdruck ⟩ ::=

$$\left\{\begin{array}{l}\langle\ \text{D(8)-Ausdruck}\ \rangle\\ \langle\ \text{M(8)-Ausdruck}\ \rangle\end{array}\right\}\ \left\{\begin{array}{l}/\\ *\\ \text{REM}\end{array}\right\}\ \left\{\begin{array}{l}\langle\ \text{D(9)-Ausdruck}\ \rangle\\ \langle\ \text{M - Ausdruck}\ \rangle\end{array}\right\}$$

$$\left\{\begin{array}{l}\langle\ \text{D(7)-Ausdruck}\ \rangle\\ \langle\ \text{M(7)-Ausdruck}\rangle\end{array}\right\}\ \left\{\begin{array}{l}+\\ -\end{array}\right\}\ \left\{\begin{array}{l}\langle\ \text{D(8)-Ausdruck}\ \rangle\\ \langle\ \text{M - Ausdruck}\ \rangle\end{array}\right\}$$

Beschreibung:

Die beiden Operanden werden im R-Modus berechnet, die R-Werte werden als ganze (!) Zahlen aufgefaßt. Auf sie wird die arithmetische Operation angewandt; das Ergebnis ist eine ganze (!) Zahl (ohne Rundung). Diese Zahl ist der R-Wert des Ausdrucks.

Die Operatoren * , / , + und - haben die übliche mathematische Bedeutung.

Mit dem REM-Operator läßt sich der Rest einer Division feststellen.

Implementierung TR 440: Das Vorzeichen des Restes wird vom linken Operand übernommen.

a) A REM B für A = -9 und B = 7
R-Wert des Ausdrucks: -2 (Rest von $\frac{-9}{7}$)

b) A REM B für A = 7 und B = -2
R-Wert des Ausdrucks: 1 (Rest von $\frac{7}{-2}$)

6.4. Vergleichsausdrücke

⟨Vergleichsoperatoren⟩ ::= EQ|NE|LS|GR|LE|GE

Vergleichsoperatoren besitzen die Priorität 6.

Syntax:

⟨Vergleichsausdruck⟩ ::=

$$\left\{\begin{array}{l}\langle\ \text{D(7)-Ausdruck}\ \rangle\\ \langle\ \text{M(7)-Ausdruck}\\ \langle\ \text{D(6)*-Ausdruck}\rangle\end{array}\right\}\ \left\{\begin{array}{l}\text{EQ}\\ \text{NE}\\ \text{LS}\\ \text{GR}\\ \text{LE}\end{array}\right\}\ \left\{\begin{array}{l}\langle\ \text{D(7)-Ausdruck}\ \rangle\\ \langle\ \text{M-Ausdruck}\ \rangle\end{array}\right\}$$

* Ist der D (6)-Ausdruck ein Vergleichsausdruck, so ist das eine verkürzte Schreibweise für die logische Verknüpfung zweier Vergleichsausdrücke. Es gilt folgende Äquivalenz:

$$E_0 \langle \text{verglop} \rangle E_1 \left[\langle \text{verglop}_i \rangle E_{i+1} \right]_{i=1}^{\infty} \quad \triangleq$$

$$E_0 \langle \text{verglop} \rangle E_1 \left[\text{LOGAND } E_i \langle \text{verglop}_i \rangle E_{i+1} \right]_{i=1}^{\infty}$$

Beschreibung:

Die R-Werte von Vergleichsausdrücken können nur die Werte 'TRUE' und 'FALSE' sein. Zur Berechnung von Vergleichsausdrücken wird mit den jeweils rechten und linken Operanden eines Operators nach deren Berechnung die Vergleichsoperation durchgeführt.

Bedeutung der Vergleichsoperatoren

Operator	Bedeutung
EQ	gleich
NE	ungleich
LS	kleiner als
GR	größer als
LE	kleiner oder gleich
GE	größer oder gleich

Die Operanden von EQ und NE werden auf Gleich- bzw. Ungleichheit ihrer Bitmuster verglichen. Mit den Operanden des Operators GR wird ein arithmetischer Vergleich durchgeführt. Im Normalfall werden innerhalb eines Vergleichsausdrucks alle Einzelvergleiche durchgeführt. Ist der Vergleichsausdruck jedoch Teil eines Bedingungsausdrucks (IF-, TEST-, WHILE-, UNTIL-, REPEATWHILE-, REPEATUNTIL Anweisung, bedingter Ausdruck), werden nur solange Einzelvergleiche von links nach rechts ausgeführt, bis ein Vergleich den Wert FALSE ergibt.

Implementierung TR 440: Ist ein Operand eine Zahl, wird beim Vergleich kein Unterschied zwischen einer negativen Null (alle Bits gesetzt) und einer positiven Null (kein Bit gesetzt) gemacht.

Beispiele:

a) IF 0 EQ X LS A ! Y GOTO L

Hat der erste Einzelvergleich 0 EQ X den Wert FALSE, wird der Rest des Vergleichsausdruckes übersprungen und mit der auf 'GOTO L' folgenden Anweisung fortgefahren.
Hat 0 EQ X den Wert TRUE, wird der zweite Einzelvergleich X LS A ! Y geprüft und, falls TRUE, auf die Marke L gesprungen.

b) A ! I := F(X) = G(X) = H(X)

Selbst wenn der erste Einzelvergleich F(X) = G(X) bereits den Wert FALSE hat, wird noch der zweite Einzelvergleich G(X) = H(X) durchgeführt (Vergleichsausdruck ist nicht die Bedingung einer Verzweigung).

c) B := U * V LE W-X/Y GR Z

In jedem Fall wird auch der zweite Vergleichsausdruck W-X/Y GR Z geprüft, auch dann, wenn der Wert des ersten Vergleichsausdruckes U * V LE W-X/Y bereits als FALSE erkannt wurde.

d) IF 'A' LE CHAR LE 'Z'

Ist bereits 'A' LE CHAR nicht erfüllt, wird hinter der IF-Anweisung fortgefahren.

6.5. Shift-Ausdrücke

Die Shiftoperatoren LSHIFT und RSHIFT haben die Priorität 6.

Syntax:

⟨ Shiftausdruck ⟩ ::=

{ D(6)-Ausdruck ⟩ / M(6)-Ausdruck ⟩ } LSHIFT { ⟨ D(7)-Ausdruck ⟩ / ⟨ M-Ausdruck⟩ }

{ ⟨ D(6)-Ausdruck ⟩ / ⟨ M(6)-Ausdruck⟩ } RSHIFT { ⟨ D(7)-Ausdruck ⟩ / ⟨ M-Ausdruck ⟩ }

Beschreibung:

Shiftausdrücke besitzen nur einen R-Wert. Der R-Wert eines Shiftausdruckes von der Form E 1 ⟨op⟩ E 2 (⟨op⟩ = Shiftoperator) ergibt sich wie folgt: Die Operanden E 1 und E 2 werden im R-Modus berechnet.

Der R-Wert des linken Operanden E 1 wird als zu shiftendes Bitmuster aufgefaßt; der R-Wert des rechten Operanden E 2 wird als ganze Zahl aufgefaßt, die angibt, um wieviel Bits das Bitmuster nach links (LSHIFT) bzw. nach rechts (RSHIFT) geshiftet werden soll. In beiden Fällen werden 0-Bits nachgezogen. Das nach dem Shift entstandene Bitmuster ist der R-Wert des Shiftausdruckes.

Implementierung TR 440: Der zweite Operand (Anzahl der Shiftstellen) muß ≦ 24 sein

Regel:

Der zweite Operand muß ≧ 0 sein.

Beispiel:

```
a) MANIFEST   $(  BPW = 24;                    // Bits per word
                  BPC = 8  $)                  // Bits per character

   W := (( CHAR 1  LSHIFT  BPC)  VEL  CHAR 2)
         LSHIFT  BPW - 2 * BPC
```

Zwei Oktaden werden linksbündig in ein BCPL-Element abgelegt.

6.6. Logische Ausdrücke

Der monadische Operator NOT besitzt die Priorität 5 (Beschreibung siehe G 5.4).
Die dyadischen logischen Operatoren LOGAND, LOGOR, EQV, NEQV besitzen folgende Prioritäten.

LOGAND	Priorität 4
LOGOR	Priorität 3
EQV, NEQV	Priorität 2

Äquivalente Darstellungen siehe O 2.

Syntax:

⟨ logischer Ausdruck ⟩ ::=

{⟨ D(4)-Ausdruck⟩ / ⟨ M(4)-Ausdruck ⟩} LOGAND {⟨ D(5)-Ausdruck⟩ / ⟨ M-Ausdruck}

{⟨ D(3)-Ausdruck ⟩ / ⟨ M(3)-Ausdruck ⟩} LOGOR {⟨ D(4)-Ausdruck⟩ / ⟨ M-Ausdruck ⟩}

{⟨ D(2)-Ausdruck ⟩ / ⟨ M(2)-Ausdruck ⟩} {EQV / NEQV} {⟨ D(3)-Ausdruck ⟩ / ⟨ M-Ausdruck ⟩}

Beschreibung:

Logische Ausdrücke besitzen nur einen R-Wert. Bei einer logischen Operation werden die L-Werte der Operanden berechnet und die Resultate bitweise gemäß nach folgender Tabelle verknüpft.

	n-tes Bit im Resultat bei			
n-te Bits der Operanden	LOGAND	LOGOR	EQV	NEQV
L, L	L	L	L	0
0, 0	0	0	L	0
L, 0	0	L	0	L
0, L	0	L	0	L

Der logische Ausdruck wird von links beginnend abgearbeitet. Ist in Bedingungsausdrücken das Gesamtresultat durch ein Teilresultat bestimmt, wird die Abarbeitung abgebrochen.

Steht der logische Ausdruck nicht in einem Bedingungsausdruck, werden alle Operanden berechnet und ausgewertet.

Die Resultate logischer Operationen sind Bitmuster. Sind alle beteiligten Operanden TRUE oder FALSE, so ist entsprechend der logischen Verknüpfung das Resultat TRUE oder FALSE.

Für beliebige Bitmuster ist der Wahrheitswert undefiniert; das Ergebnis sollte vom Benutzer nur als Bitmuster verarbeitet werden.

Beispiele:

a) IF F(X) = 0 LOGOR G(X) = 0
RESULTIS T

Zur R-Wert-Bestimmung des logischen Operators LOGOR genügt bereits das Vorhandensein eines Operanden mit dem Wert TRUE. Ist also die Bedingung F(X) = 0 erfüllt, wird die zweite Bedingung G(X) = 0 nicht mehr überprüft (Ergebnis ist in jedem Fall TRUE).
Nur wenn die Bedingung F(X) = 0 den Wert FALSE annimmt, wird noch die zweite Bedingung geprüft.

b) X := $H ABCDEF EQV $HAACCEE

Nach der gemäß EQV vorgegebenen Bitverknüpfungsart hat X folgenden Wert:

$HFEFEFE

c) Z := X LOGAND $8770077 LOGOR Y ET $8 770

Der R-Wert des logischen Ausdruckes wird wie folgt verknüpft:

Z := (X LOGAND $8770077) LOGOR (Y LOGAND $87700)

7. Bedingte Ausdrücke

Syntax:

⟨ Bedingter Ausdruck ⟩ :: =

⟨ 2-Ausdruck ⟩ —> ⟨ Ausdruck ⟩, ⟨ Ausdruck ⟩

⟨ Ausdruck ⟩ ist ein beliebiger Ausdruck von der Form:

⟨ Ausdruck ⟩ :: = ⟨ Einfacher Ausdruck ⟩ | ⟨ monad Ausdruck ⟩ |
⟨ dyadischer Ausdruck ⟩ | ⟨ bedingter Ausdruck ⟩

Beschreibung:

Der bedingte Ausdruck hat die allgemeine Form:

E 1 → E 2, E 3

Ergibt sich der R-Wert von E 1 zu TRUE, ist das Ergebnis der R-Wert von E 2.

Ist der R-Wert von E 1 FALSE, ist das Ergebnis der R-Wert von E 3. Es wird also in Abhängigkeit von E1 nur jeweils einer der Werte E 2, E 3 berechnet - vorausgesetzt, daß der R-Wert von E 1 ein logischer Wert ist. Ist diese Voraussetzung nicht erfüllt, ist der Wert des bedingten Ausdruckes undefiniert.
Der bedingte Ausdruck besitzt genau dann einen L-Wert, wenn auch die beiden Ausdrücke E2, und E3 L-Werte besitzen. Auch für die L-Wert-Bestimmung muß die Forderung nach einem logischen Wert von E1 erfüllt sein. Andernfalls ist auch die L-Wert-Bestimmung undefiniert.

Beispiele:

a) T 1 LOGOR T 2 → F(T 1), F(T 2)

In Abhängigkeit des logischen Ausdruckes

'T 1 LOGOR T 2' wird entweder F (T 1) (= TRUE-Zweig) oder F (T 2) (= FALSE-Zweig) aufgerufen

b) B 1 → X → X 1, X 2, Y → Y 1, Y 2

Der Ausdruck wird berechnet zu:

B 1 → (X → X 1, X 2), (Y → Y 1, Y 2)

c) FOR I = ANF < 0 → -ANF, ANF TO ENDE DO
$(... $)

d) B → V ! I, V ! (I+1) := G(I)

Der bedingte Ausdruck besitzt einen L-Wert, da auch die Ausdrücke V ! I und V ! (I+1) L-Werte besitzen.

e) (B → F, G) (PARAM)

In Abhängigkeit von B wird F(PARAM) oder G(PARAM) aufgerufen.

f) TRUE → V!I, 327:= 11 /* verboten, obwohl V!I einen L-Wert hat (siehe G 9.).*/

8. Konstante Ausdrücke

Konstante Ausdrücke sind Ausdrücke, deren R-Werte bereits zur Compile-Zeit berechnet werden. Sie können während des Objektlaufes nicht mehr verändert werden.

Syntax:

⟨Konstante⟩ ::= ⟨Zahl⟩ | ⟨Zeichenkonstante⟩ | TRUE | FALSE |
⟨Selektor⟩ | (⟨Konstante⟩) |

⟨MANIFEST-Konstante⟩ | { + | - | NOT } ⟨Konstante⟩ |

⟨Konstante⟩ { * | / | REM | + | - | LSHIFT | RSHIFT | LS | LE | GE | GR | EQ | NE | LOGAND | LOGOR | EQV | NEQV } ⟨Konstante⟩ |

⟨Konstante⟩ → ⟨Konstante⟩ , ⟨Konstante⟩

Beschreibung:

Der R-Wert von Konstanten ergibt sich aus:

Einfachen Ausdrücken	(siehe G 4.)
Selektoren	(siehe G 5.5)
MANIFEST-Konstanten	(siehe E 2.)
Vorzeichen-Ausdrücke	(siehe G 5.3)
Arithm.-Ausdrücke	(siehe G 6.3)
Vergleichs-Ausdrücke	(siehe G 6.4)
Shift-Ausdrücke	(siehe G 6.5)
Logische Ausdrücke	(siehe G 6.6)
Bedingte Ausdrücke	(siehe G 7.)

Konstante Ausdrücke müssen stehen:

- als CASE-Marken (SWITCHON-Statement siehe F 15.)
- in Vektordefinitionen (siehe E 1.2)
- in MANIFEST-, STATIC*-, GLOBAL-Deklarationen (siehe E 2., E 3., E 4.)
- in Selektoren (siehe G 5.5)
- in Tables* (siehe G 5.6)
- als Schrittweite in FOR-Statements, falls eine Schrittweite angegeben ist (siehe F 10.).

* In STATIC-Deklarationen sind genau wie bei Tables auch Adreßkonstante erlaubt.

Beispiele:

a) MANIFEST $(AWERT = 1000; DIFF = 200 $)

Während der Übersetzung werden für die Konstanten Ausdrücke AWERT und DIFF bereits die Konstanten 1000 bzw. 200 eingesetzt.

b) MANIFEST $(MAX = 100; MIN = 10 $)
:
:
DIFF := MAX - MIN + EPS

Der arithmetische Ausdruck MAX - MIN wird bereits zur Compilezeit durch 90 ersetzt.

c) SWITCHON ZIFF INTO $(
CASE '0' : ...
CASE '1' : ...
CASE '2' : ...
:
:
$)

d) MANIFEST $(SEL = SLCT 12 : 3 : 10 $)

9. Ausdrücke vom L-Typ (L-Ausdrücke)

L-Ausdrücke sind dadurch gekennzeichnet, daß ihnen neben einem R-Wert auch ein L-Wert (Adresse) zugeordnet werden kann. Nur diese Ausdrücke können Operand des LV-Operators sein.

Beschreibung:

Ist E ein Ausdruck vom Typ L, so bezeichnet der R-Wert von LV E den L-Wert von E.

Ausdrücke mit L-Wert sind:

- Variable
- RV-Ausdrücke
- Vektorausdrücke
- bedingte Ausdrücke, deren Alternativ-Ausdrücke beide vom Typ L sind
- in Klammern eingeschlossene Ausdrücke vom Typ L
- mit pos. Vorzeichen versehene Ausdrücke vom Typ L

L-Ausdrücke werden benutzt als Operand des LV-Operators, sowie auf der linken Seite von Zuweisungen.

Beispiele:

a) `X → Z1, Y1 := V ! I`

Die Alternativausdrücke Z 1 und Y 1 des Bedingungsausdruckes müssen L-Ausdrücke sein, da sie auf der linken Seite einer Zuweisung benutzt werden.

b) `V ! (I+9) := LV ZMAX`

Der Vektorausdruck V ! (I+9) ist ein L-Ausdruck, ZMAX ist ein L-Ausdruck. Beiden kann eine Adresse zugeordnet werden.

10. Wiederholung von Ausdrücken und NIL, Listen von Ausdrücken und NIL

Einige Listen in BCPL können Ausdrücke und NIL (siehe E 1.1) enthalten. Diese Listen dürfen für die Ausdrücke und NIL Replikatoren enthalten.

Syntax:

⟨Ausdrucksliste⟩ ::= E [REP K] $\left[\text{, E [REP K]}\right]_{0}^{\infty}$

⟨Initialisierungsliste⟩ ::= $\left\{\begin{matrix}\text{NIL}\\ \text{E}\end{matrix}\right\}$ [REP K] $\left[\text{, } \left\{\begin{matrix}\text{NIL}\\ \text{E}\end{matrix}\right\} \text{ [REP K]}\right]_{0}^{\infty}$

⟨Adressen-Konstanten-Liste⟩ ::= $\left\{\begin{matrix}\text{K}\\ \text{String}\\ \text{Table}\end{matrix}\right\}$ [REP K] $\left[\text{, } \left\{\begin{matrix}\text{K}\\ \text{String}\\ \text{Table}\end{matrix}\right\} \text{ [REP K]}\right]_{0}^{\infty}$

Bezeichnung:

E ist ein Ausdruck

K ist eine Konstante

Table oder String dürfen auch in Klammernpaare eingeschlossen sein.

Wirkung:

E REP K ist gleichwertig zu

$\underbrace{\text{E, E, E, ..., E}}_{\text{K-mal}}$

NIL REP K ist gleichwertig zu

$\underbrace{\text{NIL, NIL, NIL, ..., NIL}}_{\text{K-mal}}$

Bei Tables, in der Definition dynamischer Variablen in (Mehrfach-) Zuweisungen und als aktuelle Parameterlisten treten Listen von Ausdrücken und NIL auf, in denen die Wiederholung gleicher Terme abgekürzt geschrieben werden kann mit Hilfe des Replikators.

Regel:

Der Replikator K muß ≧ 1 sein

Beispiele:

a) LET T = TABLE 1 REP 5, (TABLE 2 REP 3, 3) REP 2
ist äquivalent mit
LET T = TABLE 1,1,1,1,1, (TABLE 2,2,2,3), TABLE 2,2,2,3

b) LET A,B,C,D,E,F = 1 REP 3, NIL REP 3
ist äquivalent mit
LET A,B,C,D,E,F = 1,1,1,NIL,NIL,NIL

c) B,C,D := (3+A( )) * 5 - 7 REP 3

d) WRVAL (K.B6, 3, FALSE REP 4, STRINGN REP 2)

BCPL-EA

1. Einführung

1.1. Dateiorganisation

Die Ein- und Ausgabe von BCPL-Information geschieht grundsätzlich über Dateien, die über symbolische Gerätenummern in den einzelnen EA- Anweisungen angesprochen werden.
Diese Gerätenummern sind ganze Zahlen und können folgende Werte annehmen:

$$1 \leqq \text{symb. Gerätenummer} \leqq 99$$

Folgenden Gerätenummern ist eine feste Bedeutung zugeordnet:

logische Gerätenummer	Zuordnung
5	Normaleingabemedium (NEM)
6	Normalausgabemedium (NAM)
8	Konsoleingabemedium (KEM)
9	Konsolausgabemedium (KAM)
4	NAM/KAM

Die diesen vier Gerätenummern zugeordneten Dateien werden dem Benutzer ohne vorherige Deklaration implizit zur Verfügung gestellt.
Die durch andere Gerätenummern angesprochenen Dateien müssen bereits vor der Programmausführung existieren, indem sie vom Benutzer vor dem Objektstart explizit deklariert werden. Für Systemprogramme gibt es im Rahmen der BCPL-EA eine dateidefinierende Prozedur (siehe H 4.14).

1.1.1. Dateityp

Es können Dateien vom Typ RAN (Random mit Satznummer), RAM (Random mit Satzmarke) und SEQ (sequentiell) bearbeitet werden.
Die normale Bearbeitungsart ist sequentiell; bei RAN- und RAM-Dateien kann die sequentielle Bearbeitung durch einen Positionsauftrag POSIT (siehe H 4.9), unterbrochen werden.
Bei Betriebsartwechsel (Lesen ↔ Schreiben) wird der aktuelle Satz abgeschlossen und mit dem nächsten Satz fortgefahren.

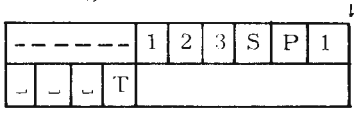
1.1.2. Satzbau

Es können Dateien mit dem Satzbau O (Oktaden), A (Ausgabezeichen) und W (Ganzworte) verarbeitet werden. Bei formatgesteuerter EA sind nur A- und O-Dateien zugelassen, wobei eine A-Datei nur schreibend bearbeitet werden kann.
Es können nur solche Dateien sinnvoll formatfrei (W-Dateien) gelesen werden, die formatfrei mit BCPL beschrieben wurden.

1.1.3. Satzlänge

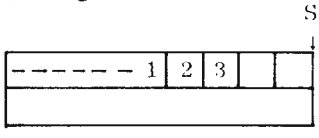
Die Länge der einzelnen Datensätze kann explizit oder implizit vorgegeben werden.
Eine explizite Längenbegrenzung erfolgt durch das im Formatstring stehende Vorschubsteuerzeichen "/". Es veranlaßt den Abschluß des aktuellen Satzes und eine Positionierung auf den nächsten Satz. Eine implizite Satzlängenbegrenzung erfolgt nur dann, wenn die Satzlänge bei der Dateideklaration mit G (genau) oder M (maximal) festgesetzt wurde. Bei formatgesteuerter EA wird die Satzlängenangabe U (ungefähr) behandelt wie M 1535.
Überschreitet die zu lesende oder abzulegende Informationsmenge für ein Formatcodeelement die Restlänge, so wird der aktuelle Satz abgeschlossen und die Information in den nächsten Satz geschrieben, bzw. aus dem nächsten Satz gelesen.
Ist der Satzbau mit G (genau) beschrieben und wird die Satzlänge unterschritten, so erfolgt Fehlerabbruch.

Beispiel:
a) Ausgabe:



Ausgabegrößen	Format
123	
"SP1"	"I3, A3, L4"
TRUE	

b) Eingabe:



Der aktuelle Satz kann das anstehende Datum (Format L4) nicht mehr vollständig aufnehmen. Bei Satzbau M wird der aktuelle Satz abgeschlossen und in den nächsten Satz geschrieben; bei Satzbau G erfolgt Fehlerabbruch.

Sonderregelung für Strings (siehe H 3.7).

2. Formatsteuerung

2.1. EA-Parameter

Ein Read- oder Writeaufruf hat die Form

$\left\{\begin{matrix}\text{WRITE}\\\text{READ}\end{matrix}\right\}$ (SGNR, FMSTR, PARZA, EAPAR_1, ..., $\text{EAPAR}_{\text{PARZA}}$)

FMSTR ist der Formatparameter; auf dieser Position kann die Adresse eines Formatstrings, eine Null oder die Referenz auf einen vorübersetzten Format-String liegen.

SGNR ist die symbolische Gerätenummer

PARZA ist die Anzahl der nachfolgenden EA-Parameter

EAPAR sind die EA-Parameter.
Sie bezeichnen bei der Eingabe die Adresse, auf oder ab der die eingelesene Information abgelegt werden soll. Bei der Ausgabe bezeichnen sie abhängig vom zugeordneten Formatcode den anzugebenden Wert oder die Adresse, auf der der auszugebende Wert liegt. Die Deutung, ob als Adresse oder als Wert, wird folgendermaßen durchgeführt:

Format (-code)	unformatiert	C A L S N I J Z G	
Ausgabe	W	W W W A W W W A A	W = Wert
Eingabe	A	A A A A A A A A A	A = Adresse

2.2. Formatstring

Die Ein- und Ausgabe in BCPL-Programmen kann formatfrei oder formatgesteuert erfolgen.
Die Steuerung erfolgt in den einzelnen EA-Prozeduren über einen Formatstring, der bei formatgesteuerter EA die Formatcodes der einzelnen EA-Elemente enthält. Die Adresse des Formatstrings wird durch den Formatparameter übergeben.
Bei formatfreier EA muß der Wert des Formatparameters 0 sein. Jedes EA-Element wird dann als BCPL-Element (= 1 Halbwort) in binärer Form ein- und ausgegeben. Eine Datei darf nicht gleichzeitig formatgesteuert und formatfrei bearbeitet werden.

Syntax:

⟨formatstring⟩ :: = [r] (⟨formatstring⟩) | ⟨formcode⟩ |
⟨formcode⟩ [,] ⟨formatstring⟩

⟨formcode⟩ :: = [r] ⟨repform⟩ | ⟨H - FC⟩ | ⟨T - FC⟩ | ⟨F-FC⟩

⟨repform⟩ :: = / | ⟨X - FC⟩ | ⟨N - FC⟩ | ⟨A - FC⟩ | ⟨L - FC⟩ |
⟨S - FC⟩ | ⟨C - FC⟩ | ⟨Z - FC⟩ | ⟨I - FC⟩ |
⟨J - FC⟩ | ⟨G - FC⟩ |

Der Replikator r ist eine ganze positive Zahl und kann sich sowohl auf einen Formatcode als auch auf ganze Formatcodegruppen beziehen.

$$r\ \langle repform\rangle \triangleq \underbrace{\langle repform\rangle, \langle repform\rangle, \ldots \langle repform\rangle}_{r\text{-mal}}$$

$$r\ (\langle formatstring\rangle) \triangleq \underbrace{\langle formatstring\rangle, \langle formatstring\rangle, \ldots \langle formatstring\rangle}_{r\text{-mal}}$$

Es ist möglich, in Abhängigkeit der Parameteranzahl eine variabel lange Formatcodefolge zu erhalten.
Ist die Anzahl der EA-Elemente größer als die Anzahl entsprechender Formatcodes, kann den restlichen EA-Elementen entweder der gesamte Formatstring oder ein Teil davon erneut zugeordnet werden.

a) Enthält der Formatstring Klammernpaare, so wird bei Bedarf wie folgt verfahren:
es wird der Formatstring beliebig oft von links nach rechts wiederholt, der in dem am weitesten rechts stehenden Klammernpaar nullter Ordnung eingeschlossen ist.

Die Ordnung eines Klammernpaares ergibt sich aus der Menge ineinander verschachtelter Klammernpaare:

Ordnung der Klammern:

```
... ( ... ) ... ( ... ( ...( ... ) ... ) ... ( ...) ...)      (Ordnung)
    0.    0.    0.    1.   2.   2.   1.   1.  1.   0.
```

b) Soll der ganze Formatstring in Abhängigkeit der EA-Parameteranzahl beliebig oft wiederholt werden, so ist er in Klammern einzuschließen.

Regel:
Kommata oder Blanks müssen dort stehen, wo Ziffern aus Formatcodes mit Replikatoren oder Ziffern eines anderen Formatcodes zusammentreffen und dabei Unklarheiten entstehen.

Beispiele

"I12, 3HABC, 312" oder "I12␣3HABC312"

a) "I12, 3HABC␣312", "I12␣3HABC; 312"
Es müssen Kommata (oder Blanks) gesetzt werden, um die eindeutige Zugehörigkeit der Ziffern zu Formatcodes festzulegen.

b) "(2I4, 3A3, 2I7 /)"
Enthält die EA-Parameterliste mehr als sieben Elemente, so wird für die noch anstehenden Elemente der gesamte Formatstring von links nach rechts evtl. mehrmals - wiederholt. Jede Wiederholung beginnt auf einem neuen Satz (" / ").

c) "I4 (X5, X5) I3 (I4, 2 (A3, I2))"
```
.      :  :  :    : :
0.     0. 0. 1.   1.0.
```

Das am weitesten rechts stehende Klammernpaar wird bei Bedarf wiederholt. Es ergibt sich dann die Formatcodefolge:

I4, A5, X5, I3, I4, A3, I2, A3, I2, I4, A3, I2, A3, I2, I4, A3, ...

Formatcodefolge des nullten Klammerpaares wird wiederholt.

3. Die Formatcodes

- Bei der Eingabe in einem Zahlenformat erfolgt grundsätzlich Fehlerabbruch, wenn die Zahl nicht im darstellbaren Wertebereich liegt:

$$-2^{23} < \text{Zahl} < +2^{23} \qquad ; 2^{23} = 8\,388\,608$$

- Die Werte der zugehörigen EA-Parameter sind in 2.1. erläutert.

3.1. Vorschubsteuerung

3.1.1. Vorschubsteuerung durch "/"

Syntax: [r] /

Der Replikator gibt an, wie oft eine Vorschubsteuerung vorgenommen werden soll.

Wirkung:

Das Vorschubsteuerzeichen " / " bewirkt, daß der aktuelle Satz abgeschlossen und auf den neuen Satz positioniert wird.

3.1.2. Vorschubsteuerung durch "P"

Syntax: [r] P

Der Replikator r gibt an, wie oft die durch P gewünschte Vorschubsteuerung vorgenommen werden soll.

Wirkung:

Der aktuelle Satz wird abgeschlossen und es wird bei r = 1 auf den ersten Satz einer neuen Seite positioniert.

Für r = 2 wird auf den ersten Satz der übernächsten Seite positioniert usw.

Regeln:

a) Der Formatcode "P" ist nur beim Schreiben in A-Dateien und bei Ausgabe mit den symbolischen Gerätenummern 6 und 9 erlaubt.

b) Bei Ausgabe mit der symbolischen Gerätenummer 9 (= Konsolausgabe) wird die Vorschubsteuerung "P" auf "Vorschub auf nächste Zeile" (entspricht "/" oder "1F") abgebildet.

Beispiele:

Siehe H 3.1.4.

3.1.3. Vorschubsteuerung durch "F"

Syntax: [n] F

n kann die Werte $1 \leq n \leq 7$ annehmen.

Wirkung:

Der aktuelle Satz wird abgeschlossen. Mit n Zeilen Vorschub wird auf den nächsten Satz positioniert. "F" hat die gleiche Wirkung wie "1F" oder "/".

Regel:

Der Formatcode "F" ist nur beim Schreiben in A-Dateien und bei Ausgabe mit den symbolischen Gerätenummern 6 und 9 erlaubt.

Beispiele:

Siehe H 3.1.4.

3.1.4. Vorschubsteuerung durch "R"

Syntax: R

Wirkung:

Der aktuelle Satz wird abgeschlossen und ohne Zeilenvorschub (aber mit Wagenrücklauf) wird der nächste Satz eröffnet.
Beim Drucken werden die Sätze zeichenweise übereinander gedruckt (Mehrfachdruck).

Regel:

Der Formatcode "R" ist nur beim Schreiben in A-Dateien und bei Ausgabe mit den symbolischen Gerätenummern 6 und 9 erlaubt.

Beispiele:

Die Ausgabe der folgenden Beispiele erfolgte auf Konsole (Umdefinition der symbolischen Gerätenummer 10 in 9). Die Vorschubsteuerung "P" wird also auf "1F" abgebildet.

```
WRITE(10,['VORSCHUBSTEUERUNG  P    EINE SEITE'[,0)
WRITE(10,[P,'EINE SEITE'/[,0)
WRITE(10,['VORSCHUBSTEUERUNG  F    EINE ZEILE' [,0)
WRITE(10,[F,'EINE ZEILE'/[,0)
WRITE(10,['VORSCHUBSTEUERUNG  F    DREI ZEILEN' [,0)
WRITE(10,[3F,'DREI ZEILEN'/[,0)
WRITE(10,['VORSCHUBSTEUERUNG  R    KEIN ZEILENWECHSEL'
/[,0)
WRITE(10,['0000::::' [,0)
WRITE(10,[R,'////----'/[,0)
WRITE(10,['VORSCHUBSTEUERUNG  R  KEIN ZEILENWECHSEL
 AUFSATZPUNKT AN 5.-TER STELLE DES SATZES'/[,0)
WRITE(10,['(((()))))'[,0)
WRITE(10,[R,4X,'((((',/[,0)
FINISH   &)
```

Konsolausgabe:

```
VORSCHUBSTEUERUNG  P    EINE SEITE
EINE SEITE
VORSCHUBSTEUERUNG  F    EINE ZEILE
EINE ZEILE
VORSCHUBSTEUERUNG  F    DREI ZEILEN


DREI ZEILEN
VORSCHUBSTEUERUNG  R    KEIN ZEILENWECHSEL
ØØØØ::::
VORSCHUBSTEUERUNG  R  KEIN ZEILENWECHSEL    AUFSATZPUNKT AN 5.-TER ST
ELLE DES SATZES
((((ØØØØ
```

3.2. I-Formatcode (I-FC)

Syntax: [r] I w

Der Codereplikator r gibt an, wieviel Daten mittels I-Format übertragen werden sollen.

w ist die Zahl der Oktaden, die ein EA-Element im Ein- oder Ausgabesatz belegt.

Wirkung:

Eingabe

Es werden w Zeichen eingelesen

w > 7 : Vom angegebenen w Zeichen langen Eingabefeld werden nur die rechtsbündigen sieben Stellen in eine Binärzahl konvertiert. Das links vor den sieben Stellen stehende Zeichen wird als Vorzeichen interpretiert. Die ersten w-8 Zeichen müssen Blanks sein und werden ignoriert.

w ≦ 7 : Die erste Stelle wird evtl. als Vorzeichen interpretiert, die restlichen w-1 Stellen werden in eine Binärzahl konvertiert.

Ausgabe

w > 7 : In das w Zeichen lange Ausgabefeld werden rechtsbündig sieben Ziffern mit führenden Vorzeichen ("-" oder "␣") ausgegeben. w-8 Blanks gehen den Ziffern und dem Vorzeichen voran.

w ≦ 7 : Bei negativen Zahlen werden eine Vorzeichenstelle ("-") und daran anschließend w-1 Ziffern ausgegeben.
Bei positiven Zahlen werden w Ziffern ausgegeben.

Regeln:

a) Führende Nullen werden bei der Ausgabe nicht unterdrückt.

b) Bei der Eingabe sind "+" und "␣" auf Vorzeichenposition gleichwertig, es dürfen keine Blanks folgen.

c) Umfaßt der Wert der auszugebenden Größe mehr Stellen, als die Stellenzahl w vorsieht, werden w Sternchen ausgegeben.

Beispiele:

Eingabe

Eingabedaten	Formatcode	Wert der Variablen
+ 1 2 3 4 5 6	I 7	+ 1 2 3 4 5 6
- 4 3 5	I 4	- 4 3 5
␣1 2 3	I 4	+ 1 2 3

Ausgabe

Wert der Variablen	Formatcode	Ausgabedaten
+ 1 2 3 4	I 5	0 1 2 3 4
- 1 2 3 4	I 7	- 0 0 1 2 3 4
+ 9	I 6	0 0 0 0 0 9
+ 4 1 3 6 7 5	I 10	␣␣␣0 4 1 3 6 7 5
- 1	I 9	␣ - 0 0 0 0 0 0 1
- 1 2 3 4 5	I 5	* * * * *

3.3. J - Formatcode (J - FC)

Syntax: [r] J w

Der Codereplikator r gibt an, wieviel Daten mittels J - Format übertragen werden sollen.

w ist die Zahl der Oktaden, die ein EA-Element im Ein- oder Ausgabesatz belegt.

Wirkung:

Eingabe

Es werden w Stellen eingelesen.

Das Eingabefeld darf Ziffern, ein vorangehendes "-" oder "+" -Zeichen und führende Blanks enthalten. Das erste von Blank verschiedene Zeichen ("-", "+" oder Ziffer) bestimmt also das Vorzeichen. Die Ziffernfolge wird in eine Binärzahl konvertiert.

Ausgabe

Es werden genau w Stellen ausgegeben, die rechtsbündig das EA-Element enthalten. Führende Nullen werden unterdrückt. Ein negatives Vorzeichen steht direkt vor der Ziffernfolge. Die evtl. freien Stellen links der Ziffernfolge werden mit Blanks besetzt.

Regeln:

a) Bei der Eingabe sind "+" und "␣" gleichwertig.

b) Umfaßt der Wert der auszugebenden Größe mehr Stellen, als die Stellenzahl w vorsieht, werden w Sternchen ausgegeben.

c) Bei der Eingabe darf dem Vorzeichen kein Blank folgen.

Beispiele:

Eingabe

Eingabedaten	Formatcode	Wert der Variablen
1 2 3 4 5 6	J 6	+1 2 3 4 5 6
␣ ␣ ␣ 1 2 3	J 6	+1 2 3
-1 2 3	J 4	-1 2 3
␣ ␣ ␣ ␣ -1	J 6	-1
-1	J 1	Fehler

Ausgabe

Wert der Variablen	Formatcode	Ausgabedaten
+1 2 3 4	J 5	␣1 2 3 4
1 1 1 1	J 7	␣ ␣ ␣ 1 1 1 1
-2	J 5	␣ ␣ ␣ -2
-1 2 3 4 5	J 5	* * * * *

3.4. N-Formatcode ((N-FC))

Syntax: [r] N

Der Codereplikator r gibt an, wieviel Daten mittels N-Format übertragen werden sollen.

Wirkung:

Eingabe

Es werden solange Blanks überlesen, bis ein Vorzeichen oder eine Ziffer auftritt. Ausgehend von dieser Position werden alle Zeichen bis zum nächsten Nichtziffernzeichen (einschließlich Blank) oder Satzende eingelesen und in Binärform konvertiert. Das Trennzeichen (Nichtziffernzeichen) kann z.B. bei der nächsten Eingabe mit A-Format gelesen oder mit X-Format überlesen werden.

Ausgabe

Der Inhalt des Ausgabeelementes wird als Zahl gedeutet und mit entsprechendem Vorzeichen ("-" , falls negativ) auf der kleinstmöglichen Stellenzahl ausgegeben.

Beispiele:

Eingabe

Eingabedaten	Formatcode	Wert der Variablen
␣␣1324␣␣	N	+1324
␣-99E3	NXN	-99 +3
312+737-1*	3(NA1)	+312 +737 +1
␣␣13 --1	NXN	+13 -1
139734	2N	139734 (Eine Zahl!) trotz Formatcodereplikator

Ausgabe

Wert der Variablen	Formatcode	Ausgabedaten
+123	N	+123
-99999	N	-99999
-0	N	-0
129 78	NXN	129␣78
129 78	NN	12978

3.5. A - Formatcode ⟨A - FC⟩

Syntax: [r] A w

Der Codereplikator r gibt an, wieviel Daten mittels A - Format übertragen werden sollen.

w ist die Anzahl Oktaden, die ein EA-Element im Ein- oder Ausgabesatz belegt.

Wirkung:

Eingabe

Jedes BCPL-Element (TR 440 Halbwort = 24 Bits) kann bis zu drei Zeichen (drei Oktaden) aufnehmen.

w ≦ 2 : Die Zeichen werden rechtsbündig im BCPL-Element abgelegt und nach links mit Ignores (binäre Nulloktaden) aufgefüllt.

w > 2 : Es werden w - 3 Zeichen überlesen, die letzten drei Zeichen werden in der angegebenen Reihenfolge im BCPL-Element abgelegt.

Ausgabe

w ≦ 2 : Die im BCPL-Element rechtsbündig stehenden w Zeichen werden ausgegeben.

w > 2 : Es werden w - 3 Blanks und dann die drei Zeichen des BCPL-Elementes ausgegeben

Beispiele:

Eingabe

Eingabedaten	Formatcode	Inhalt des BCPL-Elements
A 3	A 2	"IG A 3"
+12	A 3	" +12"
AB12C	A 5	" 12C"

Ausgabe

Inhalt des BCPL-Elements	Formatcode	Ausgabedaten
"1AB"	A 1	B
"*-+"	A 3	*-+
" YYY"	A 6	␣␣␣YYY

3.6. L-Formatcode ⟨L-FC⟩

Syntax: [r] L w

Der Codereplikator gibt an, wieviel Daten mittels L-Format übertragen werden sollen.

w ist die Anzahl Oktaden, die ein EA-Element im Ein- oder Ausgabesatz belegt.

Wirkung:

Eingabe

Es werden w Zeichen eingelesen. Der logische Wert TRUE wird erkannt, wenn (abgesehen von führenden Blanks) die eingelesene Zeichenfolge mit dem Alphazeichen T beginnt; der Wert FALSE wird erkannt, wenn die Zeichenfolge mit dem Alphazeichen F beginnt. Ist das erste von Blank verschiedene Zeichen weder T noch F, erfolgt Fehlerabbruch.

Ausgabe

Es werden w-1 Leerzeichen und daran anschließend in Abhängigkeit des Variablenwertes das Alphazeichen T (für TRUE) oder F (für FALSE) ausgegeben.

Beispiele:

Eingabe

Eingabedaten	Formatcode	Wert der Variablen
␣␣T	L 3	TRUE
F	L 1	FALSE
␣TEE␣	L 6	TRUE
FALSCH 9␣	L 8	FALSE

Ausgabe

Wert der Variablen	Formatcode	Ausgabedaten
TRUE	L 1	T
FALSE	L 4	␣␣␣F
FALSE	L 3	␣␣F

3.7. S - Formatcode (Stringformat) ⟨S - FC⟩

Syntax: [r] S w

Der Codereplikator r gibt an, wieviel Strings mittels S-Format übertragen werden sollen.

w ist die Anzahl Oktaden.

Eingabe

Die w gelesenen Zeichen werden als BCPL-String abgelegt. Der Inhalt des zugehörigen Eingabeelementes bezeichnet die Adresse, ab der der BCPL-String abgelegt werden soll.

Ausgabe

Das zum aktuellen S-Format gehörende Ausgabeelement adressiert einen BCPL-String der linksbündig in die w nächsten Stellen des aktuellen Ausgabesatzes ausgegeben wird. Ist w größer als die Anzahl auszugebender Zeichen, wird das Ausgabefeld rechts mit Blanks aufgefüllt.
Ist die Stellenzahl w größer als die Anzahl noch zu belegender Stellen im Ausgabesatz, wird auf den nächsten Satz positioniert und der BCPL-String ab dort auch über Satzgrenzen hinweg ausgegeben. Ist w kleiner als die Stringlänge, werden nur w Zeichen ausgegeben.

Achtung:
Es ist darauf zu achten, daß die Stringadresse zulässig und bei der Eingabe w ≦ 255 ist.

Beispiele:

Der R-Wert von S sei eine zulässige Stringadresse und S sei in den folgenden Beispielen das EA-Element.

Eingabe

Eingabedaten	Formatcode	Wirkung
"A B C D E F G"	S 7	String "A B C D E F G" wird ab der durch S bezeichneten Adresse als BCPL-String abgelegt.

Ausgabe

String	Formatcode	Ausgabedaten
"A B C D E F"	S 10	A B C D E F ␣ ␣ ␣ ␣
"A B C D E F"	S 3	A B C
"A B C D E F"	S6	A B C D E F

3.8. X - Formatcode (X - FC)

Syntax: [n] X

n ist eine ganze positive Zahl

Wirkung:

Eingabe

Die Konstante n gibt an, wieviel Zeichen zu überlesen sind.

Ausgabe

Die Konstante n gibt an, wieviel Blanks auszugeben sind.

Regel:

Ist n nicht angegeben, wird n = 1 angenommen.

Beispiele:

Eingabe

Eingabedaten	Formatcode	Wirkung
"A B C 1 2 3"	3 X	Die Zeichen A B C werden überlesen

Ausgabe

Formatcode	Ausgabedaten
3 X	␣ ␣ ␣
X	␣

3.9. H - Formatcode (H - FC)

Syntax:

n H a_1 a_2...a_n ist äquivalent mit

' a_1 a_2...a_n'

Wirkung:

Eingabe

Es werden n Zeichen überlesen

Ausgabe

Die angegebenen n Zeichen werden ausgegeben

Beispiele:

Eingabe

Eingabedaten	Formatcode	Wirkung
"A B C D E F G"	3 H W 1 1	3 Zeichen (A B C) werden überlesen
"1 2 3 4 5 6"	'ALPHA'	5 Zeichen (1 2 3 4 5) werden überlesen

Ausgabe

Formatcode	Ausgabedaten
8 H ERGEBNIS	ERGEBNIS
' SUMME : ␣␣ '	SUMME : ␣␣

3.10. Z - Formatcode ⟨Z - FC⟩

Syntax: [r] Z

Der Codereplikator r gibt an, wieviel Daten mittels Z-Format übertragen werden sollen.

Wirkung:

Eingabe

Es werden 9 Stellen eingelesen. Die drei ersten Stellen müssen die Zeichenfolge Blank, Typenkennung, Blank enthalten. Ab der 4. Stelle werden die restlichen 6 Stellen als Tetradenmuster gedeutet und dem aktuellen Eingabeelement als R-Wert zugeordnet.

Ausgabe

Es werden 9 Stellen ausgegeben: Blank, Typenkennung, Blank und sechs Tetraden. Sie stellen den R-Wert des aktuellen Ausgabeelementes dar.

Bemerkung:

Zur Anwendung und für das Verständnis des Z-Formats ist die Kenntnis der internen Datenablage erforderlich (maschinenabhängig).

Regel:

Der EA-Parameter muß bei der Ein- und Ausgabe eine Adresse sein.

Beispiele:

```
READ (5,  " Z " , 1, ADR)
```

Der Eingabeparameter ADR muß eine Adresse sein

```
READ (10, " Z " , 1, LV  VAR1)
```

Von der Variablen VAR1 wird auf Parameterposition die Adresse übergeben

```
WRITE (6,  " Z " , 1, ADR)
WRITE (10," Z " , 1, LV  VAR1)
```

3.11. C - Formatcode ⟨C - FC⟩

Syntax: [r] C p

p ist eine ganze Zahl, die größer oder gleich Null ist

Wirkung:

Eingabe

Es werden p Zeichen überlesen und das folgende Zeichen rechtsbündig in das zugehörige BCPL-Element abgelegt. Das Halbwort wird mit Ignores aufgefüllt. Wird beim Überlesen der Zeichen oder beim Einlesen des letzten Zeichens Satzende überschritten, so wird statt des Zeichens im BCPL-Element die Rückmeldung für Satzende abgelegt. Das nichtabgelegte Zeichen kann mit dem nächsten EA-Aufruf gelesen werden.

Die Rückmeldungen lauten: (Dezimalwerte)

21 ...	Satzende erreicht oder überschritten
33 ...	Ende der Eingabe (nur bei KEM)
255 ...	Dateiende erreicht.

Ist $p > 0$ so können trotz Satzendemeldung bereits die ersten Zeichen des nächsten Satzes gelesen sein.

Ausgabe

Es werden p Blanks und das rechtsbündig im EA-Element stehende Zeichen ausgegeben. Hat das EA-Element den Wert 21, so wird statt der Ausgabe des letzten Zeichens der aktuelle Satz abgeschlossen.

Beispiele:

Eingabe

Eingabedaten	Formatcode	Wert des aktuellen BCPL-Elements
ABCD!1234	C 4	IGIG!

↑

aktuelle Position

Ausgabe

Wert des aktuellen BCPL-Elements	Formatcode	Ausgabebild
"ABC"	C 6	␣␣␣␣␣␣C

3.12. T - Formatcode ⟨T - FC⟩

Syntax: T p

p ist eine ganze positive Zahl

Wirkung:

Eingabe

Im aktuellen Satz wird auf das p-te Zeichen positioniert und für den nächsten Formatcode ab dort gelesen.

Ausgabe

Im aktuellen Satz wird auf das p-te Zeichen positioniert, das nächste geschriebene Zeichen wird auf diese Position abgelegt.

Achtung:

p darf die aktuelle Satzlänge nicht übersteigen.

Beispiel:

Ausgabedaten	Format	Wirkung
"NAME", 27	/S7, T5, NX	NAME 27␣
12, 513,	/N T 7N	12␣␣␣␣513

3.13. G - Formatcode ⟨G - FC⟩

Syntax: G n

n ist eine ganze positive Zahl

Wirkung:

Eingabe

Es werden n Zeichen eingelesen und ab der, durch den R-Wert des Eingabeparameters bezeichneten Adresse abgelegt.

Der Unterschied zum S-Format beruht darauf, daß ein mit dem G-Format eingelesener Zeichensatz in Stringhandling-Format abgelegt wird (siehe auch String Handling FORTRAN) während das S-Format BCPL-Strings verarbeitet.

Ausgabe

Es werden n Zeichen ab der durch den Ausgabeparameter bezeichneten Adresse ausgegeben. Der String muß im Stringhandling-Format abgelegt sein.

Ist n größer als die Stringlänge, so werden entsprechend viel Blanks an den String angefügt.

Ist n kleiner als die Stringlänge, so werden nur n Zeichen ausgegeben.

Regel:

Der EA-Parameter muß bei der Ein- und Ausgabe eine Adresse sein.

Beispiele:

```
READ (10, "G4 ", STRL1)
```

Eingabe	Formatcode	Wirkung
A 1 B 2	G4	String A 1 B 1 wird ab der durch STRL 1 bezeichneten Adresse im Stringhandling-Format abgelegt.

Ausgabe

String	Formatcode	Ausgabedaten
RESULTA	G 7	RESULTA
RESULTA	G 3	RES
RESULTA	G 9	RESULTA␣␣

4. EA-Prozeduren

Alle EA-Prozeduren sind nichtrekursiv. Wie bei allen anderen BCPL-Prozeduren ist die Parameterübergabe by value; d.h. es werden nur die Werte der Variablen übergeben.

Bei den Eingabeprozeduren, die die eingelesenen Daten einem Variablennamen zuordnen sollen, ist deshalb auf Parameterposition die Adresse dieser Variablen zu übergeben (z.B. mittels LV-Operator). Ein Dateiende wird bei den Eingabeprozeduren READ [P] und READRANGE über einen Funktionswert zurückgemeldet (siehe H 3.11).

Die EA-Prozeduren müssen mit:
NONREC 1 : READ, WRITE,...
EXTERNAL BL.EA: READ, WRITE,...
deklariert werden.

4.1. READ

Syntax:

READ(SGNR, FMADR, PARZA, ADRELM, ADRELM, ... ADRELM)

Bezeichnung:

SGNR : symbolische Gerätenummer
FMADR : Adresse eines Formatstrings (Formatparameter)
PARZA : Anzahl der Eingabeelemente
ADRELM: Adresse eines Eingabeelementes (EA-Parameter)

Wirkung:

Es werden gemäß der im Formatstring vorgegebenen Formatvorschriften die Eingabedaten aus der durch SGNR bezeichneten Datei gelesen.
PARZA gibt die Zahl der Eingabeelemente an. Ein Datenende (= Dateiende) kann über einen Funktionswert, den die Routine nach jedem Aufruf liefert, abgefragt werden. Verlief die Ausführung der READ-Anweisung fehlerfrei (= kein Dateiende) ist der Funktionswert TRUE; wurde das Dateiende erreicht und kann der Leseauftrag nicht (vollständig) durchgeführt werden, so wird der Wert FALSE zurückgemeldet.

Regel:

Ein Datensatzende während einer READ-Ausführung bewirkt nicht die Rückmeldung FALSE; es wird im nächsten Satz weitergelesen.

Beispiele:

a) READ(5, "L4,A3", 2, LV BOOL, LV VAR1)

Daten: ␣␣FAALP

Nach Ausführung der READ-Anweisung sind die Eingabeparameter wie folgt belegt:

Variable	Wert
BOOL	FALSE
VAR1	"ALP"

b) READ(10, "I6, A5/NXN", 4, LV A, LV B, LV C, LV D)

Daten: ␣␣1234 ABWP1 ABWP2 ABWP3

␣␣␣␣513, ␣␣-12,

Nach Ausführung der READ-Anweisung sind die Variablen wie folgt belegt:

Variable	Wert
A	1234
B	"WP1"
C	513
D	-12

c) Das Vorschubsteuerzeichen am Ende des Formatstrings bewirkt, daß bei der nächsten READ-Anweisung sofort aus einem neuen Satz gelesen wird.

```
FOR I = 0 TO 10 DO
$(
      :
      A:= READ(10, "2I4, 2S11/", 4, LV VEK!I, LV FELD! I,
                                    STRING1, STRING2)
          TEST A THEN GO TO M1
                  OR $(          $)   M1: ...   $)
```

Datensatzende

Daten: ␣␣12␣-13 STRINGLEIN 1
STRINGLEIN 2
␣␣␣1 3012 AUSGABEDAT 1
AUSGABEDAT 2 ← Dateiende

Für zwei READ-Aufträge (I = 0, I = 1), (eine erneute Ausführung wird durch Dateiende verhindert) gilt nach erfolgter Eingabe folgende Variablenbelegung:

Variable	Wert
VEK ! 0	+12
FELD ! 0	-13
STRING 1	STRINGLEIN 1
STRING 2	STRINGLEIN 2
VEK ! 1	+1
FELD ! 1	+3012
STRING 1	AUSGABEDAT 1
STRING 2	AUSGABEDAT 2

d) Einlesen einer Lochkarte im A1-Format

```
FOR I = 1 TO 80 DO
READ(5, "A1", 1, LV V! (I-1))
```


4.2. WRITE

Syntax:

WRITE(SGNR, FMADR, PARZA, EAELM, ... EAELM)

Bezeichnung:

SGNR : symbolische Gerätenummer
FMADR : Adresse eines Formatstrings
PARZA : Anzahl der Ausgabeelemente
EAELM : Ausgabeelemente

Wirkung:

Es werden gemäß der im Formatstring vorgegebenen Formatvorschriften, die Ausgabedaten in die durch SGNR bezeichnete Datei ausgegeben.
PARZA gibt die Zahl der Ausgabeelemente an.

Beispiele:

a) WRITE(6, "\, L6, 2A5", 4, A, B, V!1, V!2)

Variable	Wert
A	-777
B	FALSE
V!1	"ALC"
V!2	"123"

Ausgabebild:

↓ Position des letzten Zeichens von vorangehender EA-Prozedur

Leerzeile

- 777 ␣␣␣␣␣ F ␣␣ ABC ␣␣ 123

b) FOR I = 0 TO 2 DO
WRITE(10, "J 6/", 1, V!I)

Variable	Wert
V!0	1111
V!1	222
V!2	33

Ausgabebild:

Datensatzanfang ↓ Datensatzende ↓

␣␣1111
␣␣␣222
␣␣␣␣33

c) WRITE(6, "I9, G5, S25", 3, A, STHAND, STRING)

Variable	Wert
A	- 317
STHAND	"ALPHA TEXT"
STRING	"ERGEBNIS ␣DER ␣RUNDUNG"

Ausgabebild: Datensatzende ↓

␣- 0000317 ALPHA
ERGEBNIS ␣ DER ␣RUNDUN
G ␣␣␣␣␣

Der String wird in einen neuen Satz ausgegeben, da er nicht mehr vollständig in den aktuellen Satz geschrieben werden kann. Ab hier wird der String auch über Satzgrenzen hinweg ausgegeben.

4.3. READP

Syntax:

READP(SGNR, FMADR, PRZADR)

Bezeichnung:

SGNR : symbolische Gerätenummer

FMADR : Adresse eines Formatstrings

PRZADR: Adresse einer Eingaberoutine

Wirkung:

Die Eingabeprozedur READP ruft eine Routine PRZADR auf, deren einer Formalparameter durch die Adresse einer Systemeingabeprozedur ersetzt wird. In der Prozedur PRZADR werden über die als Formalparameter bekannte Prozedur die Eingabedaten gemäß den Formatvorschriften im Formatstring aus der Datei mit der symbolischen Gerätenummer SGNR eingelesen.

Die Prozedur PRZADR muß wie die darin verwendete Parameterprozedur nicht rekursiv erklärt sein. Ihre Nonrec-Klassen dürfen beliebig, aber nicht gleich sein.

Beispiele:

a)
```
EXTERNAL  BL.EA : READP
NONREC 1 : READP
GLOBAL  $( VECTOR : 100; ... $)
:
:
NONREC 12 : EINGA
:
:
LET  EINGA (SP)  BE
$( NONREC 11 :  SP
     FOR  I = 0  TO  10  DO
          SP (LV  VECTOR ! I)    $)
:
:
READP (5, "11I6", EINGA)
```

Obiges Beispiel ist äquivalent mit:

```
FOR  I = 0  TO  10  DO
READ (5, "I6", 1,  LV  VECTOR ! I)
```

b)
```
GLOBAL  $( STRING 1: 103; STRING 2 : 104; ...   $)
:
:
NONREC  10 : EP
:
:
LET  EP (GIBEIN)  BE
$(NONREC  11 : GIBEIN
    GIBEIN (STRING 1);  GIBEIN (STRING 2)     $)
:
:
READP (12, " S10, S8 ", EP)
:
:
READP (13, " 2S20 ",  EP)
```

Obiges Beispiel ist äquivalent mit:

```
READ (12, " S10, S8 " ,2,STRING1, STRING2)
READ (13, " 2S20 "     , 2, STRING1, STRING2)
```

4.4. WRITEP

Syntax:

WRITEP (SGNR, FMADR, PRZADR)

Bezeichnung:

SGNR : symbolische Gerätenummer

FMADR : Adresse eines Formatstrings

PRZADR: Adresse einer Ausgaberoutine

Wirkung:

Die Ausgabeprozedur WRITEP ruft eine Routine PRZADR auf, deren einer Formalparameter durch die Adresse einer Systemausgabeprozedur ersetzt wird. In der Prozedur PRZADR werden über die als Formalparameter bekannte Prozedur die Ausgabedaten gemäß den Formatvorschriften im Formatstring in die Datei mit der symbolischen Gerätenummer SGNR ausgegeben.

Die Prozedur PRZADR muß wie die darin verwendete Parameterprozedur nicht rekursiv erklärt sein. Ihre Nonrec-Klassen dürfen beliebig, aber nicht gleich sein.

Beispiele:

a)
```
NONREC  98 :  AUSG
:
:
LET AUSG (PAR)  BE
$(  NONREC  99 :  PAR
   PAR (1); PAR (2); PAR (3); PAR (4); PAR (5)  $)
:
:
WRITEP (6,  "14, I5/12, 2I4", AUSG)
```

Dieses Beispiel führt zu folgendem Ausdruck:

```
␣␣␣1␣␣␣␣2
␣3␣␣␣4␣␣␣5
```

b)
```
GLOBAL  $( VECT : 99; ...    $)
:
:
NONREC  80 : EAF
:
:
LET EAF (NUM)  BE
$(  NONREC  81 : NUM
        FOR I = 0 TO  10  DO
          NUM (LV  VECT ! 1)   $)
START: $(  LET V = VEC 10
             VECT : = V
:
:
READP (5,  " 11Z ",  EAF)
:
:
WRITEP (6, " Z/",  EAF)
```

4.5. INFORM

Syntax:

INFORM ("formatstring")

Wirkung:

Wird eine Eingaberoutine häufig mit dem gleichen Formatstring aufgerufen, ist es sinnvoll, den Formatstring nur einmal zu übersetzen und abzulegen. Dies kann durch Verwendung der Prozedur INFORM erreicht werden. Sie übergibt bei Aufruf einen Parameter, der bei den folgenden Eingabeprozeduren stets als Formatparameter angegeben werden kann. Diese Angabe hat dieselbe Wirkung wie ein ausgeschriebener Formatstring; die Abarbeitung kann schneller erfolgen.

Regel:

Es können max. 10 Formatstrings übersetzt werden.

Beispiele:

a) A : = INFORM ("(L 3, G10 /)")
READ (10, A, 4, LV B1, BSTR1, LV B 2, BSTR 2)
ist identisch mit:
READ (10, "(L 3, G10 /)", 4, LV B1, BSTR 1, LV B2, BSTR 2)

b) FORMSTR : = INFORM (" S20, Z, G 8 / ")
FOR J = 0 TO 5 DO
READ (5, FORMSTR, 3, STRINGN, LV BM, LV SHANDL)

4.6. OUTFORM

Syntax:

OUTFORM ("formatstring")

Wirkung:

Wird eine Ausgaberoutine häufig mit dem gleichen Formatstring aufgerufen, ist es sinnvoll, den Formatstring nur einmal zu übersetzen und abzulegen. Dies kann durch Verwendung der Prozedur OUTFORM erreicht werden. Sie übergibt bei Aufruf einen Parameter, der bei den folgenden Ausgabeprozeduren stets als Formatparameter angegeben werden kann. Diese Angabe hat dieselbe Wirkung wie ein ausgeschriebener Formatstring.

Regel:

Es können max. 10 Formatstrings übersetzt werden.

Beispiele:

a)
```
AFORM := OUTFORM("'RESULT : ␣␣', J 6, 'CASE ␣␣ :' N")
:
:
FOR K = 10 TO 20 DO
WRITE(6, AFORM, 2, V!(K-10), K)
```

ist äquivalent mit:

```
FOR K = 10 TO 20 DO
WRITE(6, "'RESULT : ␣␣', J 6,'CASE ␣␣ : ␣␣', N", 2, V!(K-10), K)
```

b)
```
OUT := OUTFORM ("(A5, I3, A5)")
:
:
WRITE (10, OUT, 3, A, B, C)
:
:
WRITE ( 6, OUT, 6, B1, B2, B3, B4, B5, B6)
```

4.7. READRANGE

Syntax:

READRANGE (SGNR, ADR, LNG)

Bezeichnung:

SGNR: symbolische Gerätenummer

ADR : Adresse eines Vektors

LNG : Länge des Eingabebereichs

Wirkung:

Der für die Eingabe anstehende Satz wird in den durch ADR und LNG beschriebenen Vektor eingelesen. Ein Transport erfolgt also satzweise.

Regeln:

a) Die Vektoradressen müssen gerade Adressen sein.

Ist ein Vektor mit LET V = VEC n (n = Länge des Ausgabebereichs) deklariert worden, sind für die READRANGE-Prozedur die Vektoradressen LV V!0, LV V!2, LV V!4, LV V!6 ... zugelassen.

b) Ist der anstehende Satz länger als der Vektor, so erfolgt Fehlerabbruch.

c) READRANGE meldet nach erfolgtem Transport die Anzahl belegter Halbworte zurück.

d) Bei Dateiende erfolgt Rückmeldung mit - 1.

e) Bei einer Datei, die mit READRANGE bearbeitet wird, sind die EA-Routinen READ, READP, WRITE und WRITEP nicht zugelassen.

f) LNG muß eine gerade Zahl sein.

Beispiele:

a)
```
LET  VECT = VEC  100
:
:
FOR  K - 0  TO  9  DO
READRANGE (15,  LV VECT! (K *10),  10)
```

Es werden 10 Sätze mit READRANGE gelesen. Diese Sätze füllen den gesamten Vektor VECT. Die Vektoradressen der einzelnen READRANGE-Aufrufe ergeben sich zu LV V! 0, LV V!10, LV V!20, ... LV V!90

b)
```
LET V = VEC 99 AND ELANZ = VEC 9
:
:
ZUS : = 0
FOR  J = 0  TO  9  DO
$(  ELANZ ! J : -  READRANGE (5, LV  V ! ZUS, 20)
IF  ELANZ ! J = -1  DO  BREAK
ZUS : = ZUS+ELANZ ! J  $)
```

Es werden Sätze variabler Länge gelesen. Es werden dabei maximal 20 Vektorelemente gefüllt, in jedem Fall ist es eine gerade Elementzahl. Die Vektoradressen werden in Abhängigkeit der bereits gefüllten Vektorelemente gebildet. Nach Eingabeende ist der Vektor gepackt beschrieben.

4.8. WRITERANGE

Syntax:

WRITERANGE (SGNR, ADR, LNG)

Bezeichnung:

SGNR: symbolische Gerätenummer

ADR : Adresse eines Vektors

LNG : Länge des Vektors

Wirkung:

Der durch ADR und LNG beschriebene Vektor wird als ein Satz in die durch SGNR bezeichnete Datei ausgegeben.

Regeln:

a) Die Vektoradressen müssen gerade Adressen sein.
Ist ein Vektor mit LET V = VEC n (n = Länge des Vektors) deklariert worden, sind für die WRITERANGE-Prozedur nur die Vektoradressen LV V!0, LV V!2, LV V!4, ... zugelassen.

b) Ist der Vektor länger als der zur Verfügung stehende Ausgabesatz, so erfolgt Fehlerabbruch.

c) Bei einer Datei, die mit WRITERANGE bearbeitet wird, sind die EA-Routinen READ, READP, WRITE und WRITEP nicht zugelassen.

d) Die Vektorlänge muß eine gerade Zahl sein.

Beispiele:

a)
```
LET VECT = VEC 100
  :
  :
FOR K = 0 TO 9 DO
WRITERANGE(10, LV VECT!(K*10), 10)
```

Es werden 10 Sätze geschrieben. Jeder Satz enthält in aufsteigender Reihenfolge je 10 Elemente des angegebenen Vektors. Die Vektoradressen der einzelnen WRITERANGE-Aufrufe ergeben sich zu
LV VECT ! 0, LV VECT ! 10, LV VECT ! 20,... LV VECT ! 90

4.9. POSIT

Syntax:

POSIT (SGNR, POS)

Bezeichnung:

SGNR : symbolische Gerätenummer

POS : Satznummer

Wirkung:

Es wird der aktuelle Satz abgeschlossen und auf den angegebenen Satz positioniert.
POS ist dabei ein beliebiger Ausdruck, der links mit Ignores aufgefüllt, als Satznummer gedeutet wird. Die Satznummer wird auf die Größe einer vollen Satznummer (48 Bits) erweitert.

Regel:

a) Die Positionierung ist nur bei Random-Dateien zulässig.

b) Ist zum Zeitpunkt einer POSIT-Anweisung der zuletzt bearbeitete Satz noch nicht abgeschlossen, wird er abgeschlossen und es wird auf den angegebenen Satz positioniert.

Beispiele:

a)
```
POSIT (5, 10)
```

b)
```
MANIFEST $( S = SLCT 8 : 16 $)
:
POSIT (S :: LV SATZSCHL, SATZSCHL LOGAND $H00FFFF)
```

In der Variablen SATZSCHL steht in den ersten 8 Bits die symbolische Gerätenummer; die rechten 16 Bits werden als Satzmarke interpretiert. Durch die logische Verknüpfung wird für die Bildung der Satznummer die symbolische Gerätenummer ausgeblendet.

c)
```
MANIFEST $( DATEND = 1;... $)
:
LET POINT = TRUE
FOR I = 2 TO 50 DO
$( POSIT (15, POINT -> 1, V ! DATEND)
TEST POINT THEN
POINT := READ (15, " 3A3 ", 3, LV A, LV B, LV C)
OR BREAK
:
: $)
```

Bei Dateiende (POINT = FALSE) wird auf den ersten Satz der Datei positioniert und die FOR-Schleife verlassen.

4.10. CLOSE

Syntax:

```
CLOSE (SGNR)
```

Bezeichnung:

SGNR: symbolische Gerätenummer

Wirkung:

Die durch die symbolische Gerätenummer angesprochene Datei wird abgeschlossen.

Beispiele:

a) CLOSE (SANR)

b) CLOSE (10)

4.11. REWIND

Syntax:

REWIND (SNGR)

Bezeichnung:

SGNR: symbolische Gerätenummer

Wirkung:

Es wird auf den Anfang der Datei positioniert.
Wird nach der REWIND-Anweisung auf die durch SGNR bezeichnete Datei schreibend zugegriffen, wird auf den ersten Satz positioniert; erfolgt ein lesender Zugriff, wird auf den ersten definierten Satz positioniert. Bei sequentiellen Dateien ist der erste Satz auch der erste definierte Satz.

Beispiele:

a)
```
WRITE (18, "/2A3,' LETZTER SATZ' ", 2, V ! 9, V ! 10)
REWIND (18)
READ (18,...)
```

Es wird auf den ersten definierten Satz positioniert.

b)
```
LET I = 2
:
$( POSIT (3, I)
   LES := READ (3, "'SATZNR', I4, 2A3", 3, LV T, LV AT, LV BT)
:
I := I+1   $) REPEATWHILE LES
REWIND (3)
WRITE (3, "' ANZAHL SAETZE = ␣', N/", 1, (I - 2))
```

Es wird auf den ersten Satz positioniert, in den die Anzahl der zuvor angesprochenen Dateisätze geschrieben wird.

4.12 FORWIND

Syntax:

FORWIND (SGNR)

Bezeichnung:

SGNR: symbolische Gerätenummer

Wirkung:

Es wird hinter den letzten gültigen (definierten) Satz positioniert.

Beispiele:

```
POSIT (NR, 10)
FOR K = 10 TO 33 DO
WRITE (NR, "2A3,'RESULT :=', I6/", 3, B1, V ! K, RES)
:
:
REWIND (NR)
:
:
WRITE (NR, "'BELEGT BIS SATZ 33'/", 0)
:
:
FORWIND (NR)
```

Nach Beschreiben der Datei (Satz 10 - Satz 33) wird auf den ersten Satz positioniert, dieser beschrieben und danach wird auf den 34. Satz (noch undefiniert) positioniert.

4.13. BACKSPACE

Syntax:

BACKSPACE (SGNR)

Beschreibung:

SGNR: symbolische Gerätenummer

Wirkung:

Die Anweisung BACKSPACE bewirkt das Zurücksetzen der durch die symbolische Gerätenummer beschriebenen Datei um einen Satz. Es wird auf den zuletzt bearbeiteten Satz positioniert.

Regel:

Ist zum Zeitpunkt des BACKSPACE-Aufrufes der aktuelle Satz nicht beendet, wird er abgeschlossen. Danach wird auf den Anfang dieses Satzes positioniert.

Beispiele:

a)
```
POSIT (SGNR, 19)
WRITE (SGNR, "S20",1, STRING)
BACKSPACE (SGNR)
BACKSPACE (SGNR)
```

Die erste BACKSPACE-Anweisung bewirkt, daß der Satz mit der Nummer 19 abgeschlossen wird und auf seinen Anfang positioniert wird. Die zweite BACKSPACE-Anweisung bewirkt ein Positionieren auf den Satz mit der Nummer 18.

b)
```
FOR J = 0 TO 20 DO
READRANGE (4, LV V!(J * 10), 10)
:
FOR J = 1 TO 3 DO
BACKSPACE (4)
WRITERANGE (4, LV V!50, 20)
```

In einer sequentiellen Datei werden 21 Sätze geschrieben.
Durch BACKSPACE wird auf den 18. Satz positioniert. Ab hier werden die Sätze erneut beschrieben.

c)
```
FORWIND (10)
BACKSPACE (10)
READ (10, "'LETZTE BEARBEITETE AUFTRAGSNR: ␣', I5", 1, LV NR)
```

In einer nichtleeren Datei wird der letzte Satz gelesen.

4.14. DECLDAT

Syntax:

```
DECLDAT ("Name", SGNR, SZSCHL, SZZAHL, SBSCHL, SZBAU, TRAEG)
```

Bezeichnung:

"Name"	:	Dateiname
SGNR	:	symbolische Gerätenummer
SZSCHL	:	Satzzahlschlüssel
SZZAHL	:	Satzzahl
SBSCHL	:	Satzbauschlüssel
SZBAU	:	Satzbau
TRAEG	:	Dateiträger und -typ

Wirkung:

Durch die DECLDAT-Anweisung wird eine Datei in der Standarddatenbasis deklariert, deren Name, symbolische Gerätenummer und Aufbau durch die Parameter beschrieben werden.

Die Leistungen dieser Prozedur entsprechen denen des Dateikommandos.

Die möglichen Eingaben zu SZSCHL, SBSCHL und TRAEG, sowie deren Wirkung sind wie folgt definiert:

SZSCHL

Wert	Bedeutung	
1	M	(= maximal)
2	G	(= genau)
3	U	(= ungefähr)

SBSCHL

Der Parameter SBSCHL ist eine zweistellige Zahl, deren erste und zweite Ziffer die folgenden Werte annehmen können.

1. Ziffer

Wert	Bedeutung	
1	M	(= maximal)
2	G	(= genau)
3	U	(= ungefähr)

2. Ziffer

Wert	Bedeutung	
1	O	(= Oktaden)
2	W	(= Ganzworte)
4	A	(= Ausgabezeichen)

TRAEG

Der Parameter TRAEG ist eine zweistellige Zahl, deren erste und zweite Ziffer die folgenden Werte annehmen können.

1. Ziffer

Wert	Bedeutung	
1	P	(= Platte)
2	T	(= Trommel)
3	MB	(=Magnetband)

2. Ziffer

Wert	Bedeutung	
1	SEQ	(= sequentiell)
2	RAN	(= Random mit Satznummer)
3	RAM	(= Random mit Satzmarke)

Regeln:

a) Der Parameter "Name" darf nur die Zeichenfolge für den Dateinamen enthalten. Angaben zu Generations- und Versionsnummer sowie eine zusätzliche Datenbasisangabe sind nicht erlaubt.

b) Der Dateiname darf maximal 12 Zeichen lang sein, das erste Zeichen muß ein Buchstabe sein, ihm können Buchstaben oder Ziffern folgen.

Beispiele:

a) Es soll eine RAM-Datei mit Satzbau G 80 O (genau 80 Oktaden) und Satzzahl M 500 (maximal 500) auf der Trommel deklariert werden.

```
DECLDAT ("DAT1",  10,  1,  500,  21,  80,  23)
                       |        |        |
                       |        |        RAM
                       |        Oktaden
                       maximal
                                genau    Trommel
```

b) Es soll eine SEQ-Datei mit Satzbau U 100W (ungefähr 100 Ganzworte) und Satzzahl G 30 (genau 30) auf der Platte deklariert werden.

```
DECLDAT ("ABL",  99,  2,  30,  32,  100,  11)
                      |        |        |
                      genau    Ganzworte  SEQ
                               |
                               ungefähr   Platte
```

4.15. Datei

Syntax:

DATEI (SGNR, "Name")

Bezeichnung:

SGNR : symbolische Gerätenummer

"Name" : Dateiname

Wirkung:

Die Prozedur DATEI veranlaßt die Zuordnung einer bereits deklarierten Datei zu einer symbolischen Gerätenummer. Die Datei muß in der Standard-Datenbasis liegen.

Beispiele:

a) DATEI (15, "AUSGABE")

Es wird der Datei "AUSGABE" die symbolische Gerätenummer 15 zugeordnet.

b)
```
DECLDAT ("BDA1", 13, 1, 40, 21, 90, 13)
:
:
DATEI (6, "BDA1")
```

Die durch DECLDAT getroffene Zuordnung Dateiname-symbolische Gerätenummer wird durch die Prozedur DATEI erweitert, d.h. verschiedene symbolische Gerätenummern sprechen dieselbe Datei an.

4.16. LOESCH

Syntax:

LOESCH (SGNR)

Bezeichnung:

SGNR : symbolische Gerätenummer

Wirkung:

Die durch SGNR bestimmte Datei wird gelöscht.

Beispiel:

a) LOESCH (13)

b) LOESCH (V ! 20)

4.17. KEYTO

Syntax:

KEYTO (SGNR, ADR)

Bezeichnung:

SGNR:	symbolische Gerätenummer
ADR:	Adresse eines BCPL-Elements

Wirkung:

Bei jedem Satztransport für die Datei mit der symbolischen Gerätenummer SGNR (z.B. beim Bearbeiten des Formatcodes "/" oder nach Ausführung der Routine POSIT) wird die Nummer (bei SEQ- und RAN-Dateien) bzw. die Marke (bei RAM-Dateien) des transportierten Satzes in die Variable mit der Adresse ADR abgelegt. Der KEYTO-Aufruf muß dynamisch vor dem durch eine EA-Anweisung bewirkten Satztransport stehen.

Achtung:

Beim Schreiben in eine Datei wird die Nummer bzw. Marke des zuletzt vollständig geschriebenen Satzes zurückgeliefert.
Beim Lesen aus einer Datei wird die Nummer bzw. Marke des aktuellen Satzes, der noch nicht vollständig abgeschlossen zu sein braucht, zurückgeliefert (siehe Beispiele b und c).

Beispiele:

a) In einer RAM-Datei sind die Korrektursätze durch die vorangehende Zeichenfolge KOR gekennzeichnet. Im folgenden Beispiel werden mit KEYTO die zugehörigen Satznummern gefunden.

RAM-Datei:

```
          Satzmarke
          ^^^^^^
          000010  1o1o1
          000020  1o1o2
          000021  KOR1o1o3
          000030  1o1o4
          000031  KOR1o1o5
          000032  KOR1o1o6
          000040  1o1o7
```

```
  MANIFEST  &(  INPUT=10  &)
  START:  &(
  LET ZEILNR,WEITE,FIRSTTHREE,KORR,KORRZAEHL=NIL REP 4
,0
  READ(8,[A3[,1,LV KORR)
  DATEI(INPUT,[D1[)
  KEYTO(INPUT,LV ZEILNR)
  LIES:
  WEITE:=READ(INPUT,[A3 [,1,LV FIRSTTHREE)
  IF WEITE
    &(WTRUE
        IF FIRSTTHREE=KORR
            &(KORR
                WRITE(9,['SATZNUMMER DES KORREKTURSA
TZES=  ',N/[,1,ZEILNR)
                KORRZAEHL:=KORRZAEHL+1    &)KORR
IF   (READ(INPUT,[/[,0))       GOTO LIES  &)WTRUE
  WRITE(9,['GESAMTZAHL DER KORREKTURZEILEN=   ',N/[,1,
KORRZAEHL)
  FINISH   &)
```

Ausdruck:

```
START STDHP  E:KORR.
SATZNUMMER DES KORREKTURSATZES=  21
SATZNUMMER DES KORREKTURSATZES=  31
SATZNUMMER DES KORREKTURSATZES=  32
GESAMTZAHL DER KORREKTURZEILEN=   3

ENDE STDHP
```

b) In einer sequentiellen Datei wird geschrieben und anschließend aus ihr gelesen.

```
LET WERT,I,SATZNR=10,NIL,NIL
DECLDAT([D1[,10,3,10,31,10,11)
KEYTO(10,LV SATZNR)
FOR I=1 TO 5
&(  WRITE(10,[10/15[,3,WERT,WERT)
WRITE(9,[14/[,1,SATZNR)
&)
WRITE(10,[/[,0)
WRITE(9,[14/[,1,SATZNR)
REWIND(10)
FOR I=1 TO 5
&(  READ(10,[10/15[,3,LV WERT,LV WERT)
WRITE(9,[14/[,1,SATZNR)  &)
```

Ausdruck:

```
0001
0002
0003
0004
0005
0006
0007
0008
0009
0010
0011
0012
```

c) Schreiben und Lesen in bzw. aus einer RAM-Datei

```
LET WERT,I,SATZMA=10,NIL,NIL
DECLDAT([D1[,10,3,10,31,10,13)
KEYTO(10,LV SATZMA)
POSIT(10,3)
FOR I=1 TO 6
&( WRITE(10,[19/15[,9,WERT,WERT)
POSIT(10,I+2)
WRITE(9,[14/[,1,SATZMA)
&)
WRITE(10,[/[,0)
WRITE(9,[14/[,1,SATZMA)
POSIT(10,2)
FOR I=1 TO 6
&( READ(10,[19/15[,9,LV WERT,LV
WRITE(9,[14/[,1,SATZMA)
POSIT(10,I+2)  &)
```

Ausdruck:

```
0004
0003
0005
0007
0009
0011
0012
0003
0005
0005
0007
0007
0009
```

4.18. DATERR

Syntax:

DATERR (ERROUT, VBL)

Bezeichnung:

ERROUT:
ist der Name einer Prozedur oder ein Ausdruck, der eine Prozedur-adresse enthält. Die Prozedur muß im Benutzerprogramm als nicht-rekursiv definiert sein; die Nonrecklasse muß von 1 verschieden, kann aber sonst beliebig sein.

VBL:
ist die Adresse eines Speicherbereichs von mindestens 4 BCPL-Elementen
oder- 1

Wirkung:

Die Prozedur DATERR dient zum Anmelden einer eigenen Fehlerbehandlung (falls VBL = 0 ist) oder zum Abmelden einer (zuvor angemeldeten) Fehlerroutine.

Ist eine eigene Fehlerbehandlung angemeldet worden, so wird beim Auftreten von Fehlern in der EA (also immer dort, wo normal Fehlertexte ausgegeben werden) der Vektor VBL mit Fehlerinformation gefüllt und anschließend während des EA-Aufrufs die Prozedur ERROUT aufgerufen.

Diese Prozedur, die vom Benutzer zu schreiben ist, kann nun über die Fehler-information den Fehler analysieren und über den weiteren Programmablauf entscheiden. Dies geschieht über den Funktionswert der Prozedur ERROUT der an die EA zurückgemeldet wird.

Damit kann man nun entweder der EA die weitere Fehlerbehandlung überlassen, man kann den Fehler ignorieren oder über globale Weichen an der Aufrufstelle selbst den Fehler behandeln.

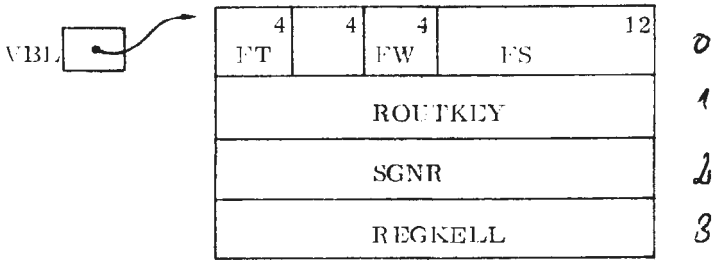
Achtung:

Bei der Fehlerbehandlung darf keine weitere EA-Prozedur aufgerufen werden. Nicht alle Fehler, die aus SSR-Fehlern hervorgehen, werden als SSR-Fehler weitergereicht (z.B. Dateiname falsch,...). Ein Aufruf von DATERR gilt für alle nachfolgenden EA-Aufrufe bis zum nächsten DATERR-Aufruf.

Folgende Werte sind in DATERR als Funktionswerte zugelassen:

Resultat	8:	mit normaler Fehlerbehandlung fortfahren
	9:	Fehlermeldung ignorieren und in der EA fortfahren (das ist i.a. nur bei Warnungen sinnvoll)
	10:	Operatorlauf mit Fehler beenden, ohne Ausgabe einer Fehlermeldung der EA
sonst:		Es wird keine Fehlermeldung abgesetzt, die fehlerauslösende EA-Prozedur (READ, POSIT,...) bricht ab und kehrt an die Aufrufstelle im rufenden Programm zurück. Als Rückmeldung wird dort die Rückmeldung der Fehlerprozedur abgeliefert.

Die Fehlerinformation im Vektor hat folgenden Aufbau:



Es bedeutet:

FT... Fehlertyp = $H0 SSR-Fehlermeldung-Abwickler
= $HC SSR-Fehlermeldung-Datenbasis
= $H4 Fehlermeldung aus der BCPL-EA

FW... Fehlerart (nur falls FT = $H4)
= $H 1 Meldung Warnung:...
= $H 2 Meldung Fehler:...

FS... Fehlerschlüssel (nur falls FT = $H4)

Die Bedeutung der Fehlerschlüssel sind im Anhang (H 4.18.1) erläutert.

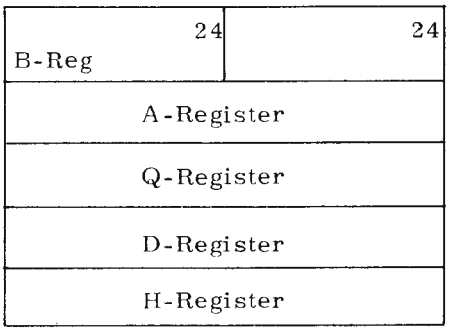
ROUTKEY... Schlüssel der die fehlerauslösende EA-Prozedur ausweist

=	1:	FORWIND
=	2:	REWIND
=	3:	POSIT
=	4:	BACKSPACE
=	5:	CLOSE
=	6:	READ
=	7:	READP
=	8:	READRANGE
=	9:	WRITE
=	10:	WRITEP
=	11:	WRITERANGE
=	12:	INFORM/OUTFORM
=	13:	DATEI
=	14:	DECLDAT
=	15:	LOESCH
=	16:	KEYTO

SGNR... symbolische Gerätenummer der Datei, bei deren Bearbeitung der Fehler auftrat

REGKELL... Zeiger auf einen Speicherbereich, in dem die Register bei SSR-Fehlermeldungen abgelegt werden.

Der Speicherbereich hat folgenden Aufbau:



4.18.1. Anhang

Bedeutung des Fehlerschlüssels FS

FS =	4:	REKURSIVER AUFRUF EINER EA-ROUTINE
	5:	OPERATORENDE WAEHREND EA-AUFRUF
	6:	STATISCHER SPEICHER ERSCHOEPFT
	7:	ZUWENIG KSB VEREINBART
	8:	AUFRUF NACH FEHLERABSCHLUSS
	9:	DATEIZUORDNUNG FEHLT
	10:	DATEIZUORDNUNG MEHRDEUTIG
	11:	DATEI-PZW. DATENBASISNAME FALSCH
	12:	KENNDATEN UNVOLLSTAENDIG
	13:	PASSWORT FALSCH
	14:	O-DATEI SATZWEISE BEARBEITET
	15:	NUR O-ODER A-DATEIEN ZULAESSIG
	16:	MAXIMALE SATZLAENGE 1500 OKTADEN
	17:	MAXIMALE SATZLAENGE UEBERSCHRITTEN
	18:	UNFORMATIERTES BEARBEITEN EINER O-DATEI
	19:	ABBRUCH NACH ERROR-PROZEDUR
	20:	NUR A, O, ODER W-DATEIEN ZULAESSIG
	21:	UNZULAESSIGE POSITIONIERUNG
	22:	BANDWECHSEL NICHT IMPLEMENTIERT
	23:	REWIND AUF NORMALMEDIEN NICHT ERLAUBT
	24:	BETRIEBSARTWECHSEL NICHT ERLAUBT
	25:	GEFORDERTE BETRIEBSART NICHT ERLAUBT
	26:	SGNR NICHT ZWISCHEN 1 UND 99
	27:	FORWIND AUF NORMALMEDIEN VERBOTEN
	28:	POSIT AUF SEQ-DATEI NICHT ERLAUBT
	29:	SATZLAENGE UNTERSCHRITTEN
	30:	UEBERSETZUNG: FEHLERHAFTER FORMATSTRING
	31:	FEHLERHAFTER FORMATSTRING, AUSFUEHRUNG ABGEBROCHEN
	32:	FORMATCODELAENGE GROESSER ALS SATZLAENGE
	33:	EINGABESTRING LAENGER ALS 255 OKT.
	34:	UNZULAESSIGE ZAHLENDARSTELLUNG
	35:	EINGABE NICHT GEMAESS L-FORMAT
	36:	EINGABE NICHT GEMAESS Z-FORMAT
	37:	FEHLER AUS S&GZF-PROGRAMMEN
	38:	T- ODER C-FORMATPOSITION NICHT IM SATZ

39:	ROUTINE NOCH NICHT IMPLEMENTIERT
40:	FORMATPARAMETER FALSCH
41:	BACKSPACE AUF NORMALMEDIEN NICHT ERLAUBT
42:	VORUEBERSETZUNG NUR ZEHNMAL MOEGLICH
43:	BACKSPACE AM DATEIANFANG NICHT ERLAUBT
44:	DATEIENDE UEBERSCHRITTEN
45:	EINGELESENE ZAHL ZU GROSS
46:	FORTSETZUNG AUS FEHLERROUTINE VERBOTEN

4.18.2. Programmbeispiel mit Fehlerbehandlungsprozedur DATERR

Protokoll:

```
//  **********  MONTAGEOBJEKT  STOHP  **********
    10      //          BEISPIELE FUER EIGENE EA-FEHLER-BEHANDLUNG
    20
    30      MONREC 1: READ, WRITE, DATERR, READRANGE
    40      MONREC   2: SSR
    50      MONREC  11: IGNORE, RETTOOBJ, SSRCHECK, STOPFEHL, IGNWARN
    60
    70      EXTERNAL  BL.EA: READ, WRITE, DATERR, READRANGE
    80      EXTERNAL  BL.INTR: SSR
    90
   100      GLOBAL    ((  START: 1
   110                   ENDE : 2
   120                   FEHLZ: 10
   130                                 ))
   140
   150      MANIFEST  (( R.IGN =  9         // RUECKMELDUNG
   160                   R.STD =  8
   170                   R.ABB = 10
   180                   FT = SLCT 4:20    // ZUGRIFF FEHLERINFORMATION
   190                   FW = SLCT 4:12
   200                   FTEA = OH 4             // EA SCHLUESSEL
   210                   FW.W = OH 1       // WARNUNG
   220                                                          ))
   230
   240      STATIC    ((  VBL = TABLE  0 REP 4
   250                   VBSSR60 = TABLE 0 REP 4
   260                   CH = 0                        ))
   270      LET  TEXTAUS (T1,T2)  BE
   280      //          HILFSROUTINE ZUR STRINGAUSGABE
   290                  WRITE (6, [// 'EA-1520/ S30 / 'VBL-DUMP:'/ [, 2, T1, T2)
   300
   310      AND RETTOOBJ () =
   320      //          RUECKKEHR INS OBJEKT NACH SETZEN EINER FEHLERWEICHE
   330           VALOF ((1 DUMPEVBL ()
   340                     FEHLZ := TRUE  // FUER AUSSEN
   350                      RESULTIS  FALSE  // DATEI ENDE SIMULIEREN
   360                                    ))1
   370
   380      AND  IGNWARN () =
   390      //          BEI WARNUNG TEXT UNTERDRUECKEN UND FORTFAHREN
   400           VALOF  ((1  DUMPEVBL ()
   410                       RESULTIS ( FT OF VBL = FTEA LOGAND  FW OF VBL = FW.W)->
   420                                                              R.IGN, R.STD
   430                  ))1  /* DAMIT WIRD ENTWEDER DER FEHLER IGNORIERT ODER NORMAL
   440                          DURCH DIE EA PER FEHLERMELDUNG WEITERBEHANDELT */
   450
```

```
460       AND  SSRCHECK () =
470       //        BEI SSR-FEHLERN STANDARD BEHANDLUNG SONST IGNORIEREN
480            VALOF  U(1
490                       DUMPEVBL ()
500                       IF  FT OF VBL  = FTEA  RESULTIS R,IGN  // KEIN SSR-FEHLER
510                       DUMPE (VBL!3, 12)
520                       RESULTIS  R,STD        // NORMALE FEHLERBEHANDLUNG
530                                              U)1
540
550       AND DUMPE (AA, LNG) BE
560                 U(U   VBSSR60!2, VBSSR60!3 := LV AA!0, _  _     -
570                       SSR (6,0, VBSSR60)
580       AND DUMPEVBL()  BE  DUMPE (VBL, 4)     // DUMP FEHLERINFORMATION
590       ALARM:    GOTO -1              // ALARMADRESSE FUER SSR-FEHLER
600       START:    VBSSR60!0, VBSSR60!1 := ALARM, UHO    // MODUS TEIL
610
620                 FEHLZ := FALSE
630                 DATERR (RETTOOBJ, VBL)          // ANMELDEN FEHLER
640                 TEXTAUS  ('FEHLER', 'DATEIZUORDNUNG FEHLT')
650                 U( LET A, B = NIL, NIL
660                    A := READ (20, [ AO[, 1, LV B) REPEATWHILE A
670                    IF  FEHLZ  DO WRITE (6, L/'KEIN DATEIENDE, FEHLER'/[,0) U)
680
690
700                 // NUN UMMELDEN DER FEHLERADRESSE ZUM IGNORIEREN VON WARNUNGEN
710                 DATERR (IGNWARN, VBL)
720                 TEXTAUS ('WARNUNG', 'DDATEI ... ')
730                 U( LET V = VEC 20
740                   READRANGE (1, V, 20)    &)
750                  TEXTAUS ('FEHLER', 'DATEI UNBEKANNT')
760                  &( LET V = VEC  20
770                     READRANGE (11, V, 20)       &)
790                  // UMMELDEN ZUM PRUEFEN AUF SSR-FEHLER
800                  DATERR (SSRCHECK,VBL)
810                  TEXTAUS ('SSRFEHLER', 'SCHREIBEND AUF LD-DATEI')
820                  WRITE (9,[ 'KEINE AUSGABE MOEGLICH' [, 0)
```

Fehlerprozedur RETTOOBJ

Fehlerprozedur IGNWARN

SSRCHECK

UMFANG DER ZWISCHENSPRACHE: 1778 HALBWORTE

ES WURDEN KEINE SYNTAXFEHLER FESTGESTELLT

EA-Fehler im Objekt:

```
$STARTE,DATEI=1-DAT01'11-DAT11

START STOMP


EA-FEHLER
DATEIZUORDNUNG FEHLT
VBI-DUMP:

002738

KEIN DATEIENDE, FEHLER


EA-WARNUNG
DATEI ...
VBI-DUMP:

002738


EA-FEHLER
DATEI UNBEKANNT
VBI-DUMP:

002738


FEHLER: BEI TYPE READRANGE DATEI 011
DATEI- BZW. DATEIBASISNAME FALSCH

ENDE STOMP
```

FT-FTEA

READ

Datei 20

1 4020 0090 0006

1 0001 4003 0016

FW-Fehler

READRANGE

Datei 1

1 4013 E00? 0004

1 000 01003016

Warnung

1 4020 3500 0004

1 000 0F003016

SSR-Fehler:

```
¤STARTE,DATEI=1-DAIC1'11-UTTD'9-BLC1MSTR

START STDHP


EA-FEHLER
DATEIZUORDNUNG FEHLT
VBL-DUMP:
          .
          .
          .
          .
          .
          .

EA-SSRFEHLER
SCHREIBEND AUF LD-DATEI
VBL-DUMP:                                          SSR-Fehler DB
                                                   ↗
002734                                             1 C00046000009          1 000009003016
                                                                           ↓
003016                                                                     QCR-Block
003018          R.RA ──►1 000E9FC00046 Datenbasis  2 000000000000          2 000000000000          3 002C36000009
003020                    2 400000000000 Fehlerschlüssel                                           2 000000000000

SSR-FEHLER DATENBASIS: NR 070  ROUTINE WRITE DATEI 009

++++++0046 AUF DATEI &STDDB.BLC1MSTR (1,00)  IST SCHREIBSPERRE GESETZT.
SSR 253 12 AUF ADRESSE E9E
OLN: STDHP

ENDE STDHP

+++++OPERATORLAUF MIT FEHLER BEENDET: STDHP
```

CODEPROZEDUREN

CODEPROZEDUREN

1. Einführung

An BCPL-Montageobjekte können Codeprozeduren (Routinen und Funktionen) anmontiert werden. Dem Benutzer wird dazu ein Satz von Makros zur Verfügung gestellt, über den bestimmte Codeteile in Codeprozeduren vereinfacht angesprochen werden können.
Die über Makros ansprechbaren Befehlscodefolgen sind die Codeteile für Eingang und Rücksprung bei Prozeduren sowie Abspeicherung von und Zugriff auf aktuellen Parameter. Bei Änderung der Anschlußbedingungen erspart die Verwendung der Makros die nötigen Codeänderungen.

1.1. Allgemeine Konventionen

a) Dem Benutzer stehen die Indexzellen vom aktuellen Stand des Unterprogrammordnungszählers bis 255 zur Verfügung.

b) Der Unterprogrammordnungszähler wird mit 30 initialisiert und bei jedem Aufruf von nichtrekursiven Prozeduren um eins erhöht.

c) Sämtliche BCPL-Elemente sind Halbworte mit Typenkennung 1. Dies ist bei der Parameterübergabe zu beachten.

2. Informationsaustausch zwischen BCPL-Programmen und Codeprozeduren

Ein Informationsaustausch zwischen BCPL-Programmen und Codeprozeduren kann über Parameter, statische Variable und Funktionswerte erfolgen.

2.1. Statische Variable

2.1.1. Deklaration als unechte Montagenamen oder Kontaktnamen

Erfolgt die Definition der statischen Variablen im TAS-Programm, kann das BCPL-Programm über eine EXTERNAL-Deklaration auf diese Größe zugreifen.

Beispiele:

	BCPL-Programm	TAS-Programm
a)	EXTERNAL F.NUMB	F&NUMB = 0/HV, EINGG F&NUMB,
b)	EXTERNAL CODE1 : **FKT1**	FKT1 = 1/HV, EINGG (FKT1),

Die Codeprozedur in Beispiel b) muß den Montageobjektnamen CODE1 besitzen (z.B. durch MO = CODE1 im UEBERSETZE-Kommando).

Erfolgt die Definition der statischen Variablen im BCPL-Programm, kann das TAS-Programm über eine EXTERN-Erklärung auf diese Größe zugreifen.

Beispiele:

	BCPL-Programm	TAS-Programm
a)	EXTERNAL C.CHAR STATIC $(C.CHAR = 0 $)	EXTERN C&CHAR,
b)	EXTERNAL : ADD STATIC $(ADD = '+' $)	EXTERN BCPL1 (ADD)

Das BCPL-Programm in Beispiel b) muß den Momtageobjektnamen BCPL1 besitzen (Spezifikation MO = BCPL1 im UEBERSETZE-Kommando).

2.1.2. Deklaration als globale Größe

Auf der BCPL-Seite werden globale Variable in der GLOBAL-Deklaration definiert, wobei ihnen eine Relativadresse in einer COMMON-Zone (bezogen auf den Zonenanfang) zugeordnet wird. Im Codeteil kann auf diese Adressen zugegriffen werden.
Der Zugriff erfolgt auf die entsprechenden Adressen der CZONE ZGLOBAL im Variablenteil.
Die Referenz bei globalen Größen erfolgt also nicht über den Namen, sondern über die zugehörige Relativadresse im Globalvektor.

Regeln:

a) Globale Größen dürfen nicht EXTERNAL deklariert sein.

b) Im MONTIERE-Kommando müssen die beteiligten Montageobjekte namentlich aufgeführt werden, wenn nicht der MO-Name über eine EXTERNAL-Deklaration bekannt gemacht wurde.

Beispiele:

```
        BCPL-Quelle                          TAS-Programm
                                             .
                                             .
        GLOBAL $( ANF:1 $)
        GLOBAL $( P:100; Q:200 $)            ZGLOBAL = CZONE V,
                .                            ABLAGE ZGLOBAL (V0),
ANF:    .                                    ZGLOBAL = ASP 100,
        .                                    P = 10/HV  -- Adresse 100 --
        P := P+1                             ASP 99,
        Q := Q+P                             Q = 20/HV  -- Adresse 200 --
                                             AEND (V0),
                                             .
                                             .
                                             B2V  P,
                                             .
                                             .
                                             C2   Q,
```

P und Q sind über die Globaladressen 100 bzw. 200 im BCPL- und Codeteil bekannt. Sie können in beiden Teilen gelesen und verändert werden.
Die BCPL-Quelle habe den Namen BCPLMO und enthält die Startadresse. Das Tasprogramm habe den Namen TASMO. Dann muß im MONTIERE-Kommando, wenn im BCPL-Programm keine EXTERNAL-Deklaration für den Tasmontageobjektnamen gegeben wird, das Anmontieren des TAS-Programms explizit verlangt werden

z.B. ⌑MONTIERE, MO = TASMO'BCPLMO

Enthält die BCPL-Quelle die Deklaration

EXTERNAL TASMO

beschränkt sich das MONTIERE-Kommando auf

⌑MONTIERE, MO = BCPLMO (siehe auch Kommandohandbuch)

2.2. Übergabe als Parameter

Parameter werden von der BCPL-Seite grundsätzlich "by value" übergeben. Codeprozeduren arbeiten in diesem Fall mit den R-Werten der Parameter Ein "call by reference" (Übergabe von Adresse) ist möglich, indem der LV-Operator auf Parameterposition benutzt wird.

2.3. Übergabe des Funktionswertes

Vor Rücksprung aus einer Funktion muß der Funktionswert in das rechte Halbwort des Akkumulators gebracht werden. Die linke Seite des Akkumulators ist vorzeichengleich mit TK 1 aufzufüllen.

3. Vereinbarungen und Makroverwendung in Codeprozeduren

3.1. Vereinbarungen der Prozedurvariablen

Die Prozedurvariable muß als Adresskonstante und durch EXTERNAL- oder GLOBAL-Deklaration der rufenden BCPL-Seite bekannt gemacht worden sein.
Die Initialisierung für die Codeprozedur ⟨name⟩ erfolgt durch:

⟨name⟩ = * ⟨name⟩ / AV1, bei Externdeklaration
bzw.
* ⟨name⟩ / AV1, im Globalvektor

Die Befehlsadresse * ⟨name⟩ wird durch das Makros NRECENTRY bzw. RECENTRY (siehe I 3.2.5) bei Prozedurbeginn definiert.

3.2. Makros

3.2.1. Bereitstellung von Makros

Alle Makros liegen in der Datei R&RAHMEN. Zu Beginn des TAS-Programms werden sie über

```
HDEF &OEFDB( R&RAHMEN,  BCPL&MAKROS),
BCPL&MAKROS,
```

verfügbar gemacht.

Die über die Makros generierten Tasbefehlsfolgen werden durch

DRUCK 3,

protokolliert.

3.2.2. Makro NRCZONE

In nichtrekursiven Prozeduren muß für die Abspeicherung der von der BCPL-Aufrufseite übergebenen aktuellen Parameter eine COMMON-Zone (Parameterbereich) zur Verfügung gestellt werden.

Syntax:

NRCZONE (⟨nonreclasse⟩ [, ⟨parameterzahl⟩]),

⟨nonreclasse⟩ ist eine zweistellige Zahl, die mit der auf der BCPL-Seite für diese Prozedur vergebenen ⟨nonreclasse⟩ übereinstimmen muß.
Fehlt die Parameteranzahl, wird als Voreinstellung der Wert 0 eingesetzt.
In einem Tasmontageobjekt darf zu einer NONREC-Klasse nur ein Makroaufruf erfolgen.

Beispiel:

BCPL-Programm	TAS-Programm
EXTERNAL B.BRING	
NONREC 12 : B.BRING	NRCZONE (12, 2),
:	
:	
B.BRING (A, LV Z)	

3.2.3. Makro NRECENTRY

Die Befehle für die Anfangsbehandlung einer nichtrekursiven Prozedur werden durch Aufruf des Makros NRECENTRY erzeugt.

Syntax:

NRECENTRY (⟨name⟩ [, ⟨Parameteranzahl⟩, ⟨Zonen-Nr.⟩]),

Beispiel:

BCPL-Programm	TAS-Programm
	:
EXTERNAL XAP	EINGG XAP,
NONREC 13 : XAP	XAP = *XAP/AV1,
:	NRCZONE (13, 3),
XAP (VECADR, LV VAR, $8773)	NRECENTRY (XAP, 3, 13),

3.2.4. Makro NRECRET

Dieses Makro veranlaßt den Rücksprung aus einer nichtrekursiven Prozedur ins rufende Programm.

Syntax:

NRECRET,

3.2.5. Makro RECENTRY

RECENTRY veranlaßt in rekursiven Codeprozeduren das Sichern und Umstellen von Verwaltungsinformation bezüglich des Arbeitsspeichers.

Syntax:

RECENTRY (⟨name⟩)

⟨name⟩ ist der Name der Prozedur.

Beispiel:

BCPL-Programm	TAS-Programm
	:
EXTERNAL FUN.A	EINGG FUN&A,
:	FUN&A = * FUN&A / AV1,
FUN.A (X, Y, Z)	RECENTRY (FUN&A),

3.2.6. Makros RECRET

Vor dem Rücksprung aus einer rekursiven Codeprozedur muß der ehemalige Zustand (vor Sprung in Codeprozedur) des Arbeitsspeichers wieder hergestellt werden.

Syntax:

RECRET,

3.2.7. Makro USEPARAM

Über das Makro USEPARAM kann der Benutzer auf die einzelnen Aufrufparameter zugreifen.

Syntax:

USEPARAM (⟨code⟩, ⟨parameternummer⟩ [, ⟨zonennummer⟩]),

Das Makro liefert den Code für den Zugriff auf einen Parameter. Über ⟨code⟩ wird festgelegt, in welcher Form (mit welchem TAS-Befehl) auf den Parameter zugegriffen werden soll.
Wird das Makro in rekursiven Codeprozeduren verwandt, kann die Angabe der Zonennummer entfallen (ist voreingestellt mit 0). In nichtrekursiven Codeprozeduren ist die Angabe der Zonennummer obligat.

Beispiele:

	BCPL-Programm	TAS-Programm
a)	EXTERNAL CASEP	.
	NONREC 20 : CASEP	USEPARAM (MCFU, 2, 20),
	.	C2 0,
	CASEP (A, LV B)	USEPARAM (B2V, 1, 20)
	.	.
b)	EXTERNAL MO : ADD	EINGG (ADD),
	.	HDEF &OEFDB (R&RAHMEN,
		BCPL&MAKROS),
	SUM := ADD (OP1, OP2)	BCPL&MAKROS,
	.	ADD = * ADD/AV1,
		RECENTRY (ADD),
		USEPARAM (B2V, 1),
		USEPARAM (A2, 2),
		RECRET,
		.

In Beispiel b) ist eine Funktion (MO-Name = MO) unter dem Namen ADD als Codeprozedur definiert.
Die Befehle im Makro USEPARAM gewährleisten, daß der Funktionswert in vorgeschriebener Weise übergeben wird.

UEBERSETZEN, MONTIEREN UND STARTEN VON BCPL-PROGRAMMEN

UEBERSETZEN, MONTIEREN UND STARTEN VON BCPL-PROGRAMMEN

1. Übersetzen von BCPL-Programmen

BCPL - (Teil-) Programme werden durch das UEBERSETZE-Kommando mit der Spezifikation SPRACHE = BCPL übersetzt.
Da in BCPL-Quellen einige Zeichen (z.B. ' $') nicht zum Zeichenvorrat des KC 1 (Lochkartencode Nr. 1) gehören, muß bei Verwendung dieser Zeichen vor der Quelle eine Codeumsteuerkarte für KC 2 liegen.

Also z.B.

```
␣UEBERSETZE, ..., QUELLE = /
/ 1XUM, COD = KC 2 ␣.
```

Diese Karte kann entfallen, wenn die im KC 1 nicht vorhandenen Zeichen z.B. durch ihre Ersatzdarstellung beschrieben werden (für ' $(' oder ' $)' → BEGIN oder END).

Die Spezifikation PROTOKOLL wird wie folgt ausgewertet:

- PROTOKOLL = - : kein Quellprotokoll
- PROTOKOLL = -STD- : Quellprotokoll und Adreßbuch des Codegenerators.
- PROTOKOLL = O : Zusätzlich zu -STD- Ausgabe des Objektcodes (TAS-Code), zusammen mit dem Adreßbuch des Codegenerators lassen sich logische Fehler in der Quelle im Objektprogramm lokalisieren.
- PROTOKOLL = A : Referenzliste mit Angaben zum Typ der verwendeten Größen, der zugehörigen Adressen und der Zeilennummern von Deklaration und Referenz.
- PROTOKOLL = R : Referenzliste.

Die Spezifikation MO wird ausgewertet wie im Kommandohandbuch beschrieben.

Mit den Angaben in der Spezifikation VERSION läßt sich die Freispeicherverwaltung im erzeugten Objekt beeinflussen. Es bedeuten:

- VERSION = - Beim Programmstart wird ein Minimum an Freispeicher zur Verfügung gestellt. Im weiteren Programmablauf wird bei Bedarf Speicher nachgefordert. Freigewordener Speicher wird nicht mehr zurückgegeben.

- VERSION = X Wirkung wie bei VERSION = -, darüberhinaus wird freigewordener Speicher zurückgegeben.

- VERSION = B Es wird keine Speicherverwaltung durchgeführt. Der Speicherbedarf des Hauptprogramms (an dynamischen Variablen) steht als Freispeicher zur Verfügung.

- VERSION = BH(n) n ist eine natürliche Zahl. Es wird keine Speicherverwaltung durchgeführt, als Freispeicher werden n K Ganzworte zur Verfügung gestellt.

2. Montieren von BCPL-Programmen

In der Spezifikation MO müssen alle Montageobjekte aufgeführt werden, die anmontiert werden sollen. Es genügt die Angabe des Montagenamens des Hauptprogramms, wenn alle anderen Montageobjekte dort durch eine EXTERNAL-Deklaration vereinbart wurden.
Ist der Name des Hauptprogramms STDHP, so kann er entfallen (siehe auch Kommandohandbuch).

3. Starten von BCPL-Programmen

Mit dem STARTE-Kommando werden BCPL-Programme gestartet. Die Spezifikation PROGRAMM wird ausgewertet wie im Kommandohandbuch beschrieben. In der Spezifikation DATEI wird die Zuordnung zwischen symbolischen Gerätenummern und Dateinamen getroffen.

TESTHILFEN, TESTEN VON BCPL-PROGRAMMEN

TESTHILFEN, TESTEN VON BCPL-PROGRAMMEN

1 Referenzliste und Adreßbuch

Mit der Spezifikation PROTOKOLL im UEBERSETZE-Kommando kann u. a. vorgesehen werden, welche zusätzlichen Angaben ins Ablaufprotokoll aufgenommen werden sollen.

Folgende Angaben dienen dem Programmtest:

PROTOKOLL = R	Zusätzlicher Druck von Referenzlisten; alle im Quellprogramm vereinbarten Größen werden alphabetisch angelistet und zwar nach Symbolischer Adresse (Quellstatement-Nr. der Vereinbarung eines Namens) und Referenzen auf diesen Namen (Nr. der Quellzeilen), d.h. alle Zeilen, in denen der Name auftritt.
PROTOKOLL = A	Auflisten des Adreßbuches; alle im Quellprogramm vereinbarten Größen werden entsprechend ihrer NONREC-Klasse und ihres Typs und Scope's ausgewiesen.
PROTOKOLL = KO	Protokoll wird auf Terminal (Konsole) angelistet.

Die Angaben R A KO sind kombinierbar (durch Apostroph getrennt, z. B. PROTOKOLL = KO'R'A).

Das folgende BCPL-Programm, bestehend aus dem Hauptprogramm STDHP und dem externen Unterprogramm UP1, wurde im UEBERSETZE-Kommando mit der Spezifikation = R'A übersetzt.

```
*UEBERSETZE
 1 QUELLE        = ENG&BCPL4
 2 SPRACHE       = BCPL
 3 NUMERIERUNG   = -STD-
 4 MD            = -STD-
 5 VARIANTE      = 0
 6 PROTOKOLL     = R'A
 9 MV            = -STD-

START PS&BCPLCUMP (19.00)
```

Hauptprogramm STDHP

Referenzliste und Adreßbuch

```
MONTAGEOBJEKT  STDHP
     10        NONREC 1: WRITE,READ
     20        EXTERNAL BL,EA:WRITE,READ
     30        GLOBAL &( START:1 &)
     40        START: &( EXTERNAL M
     50        NONREC 10:M
     60        STATIC &( ERG=0 &)
     70        MANIFEST &( A2=5; A3=10 &)
     80        &( LET B1,B2,B3,A1=2,4,6,0
     90        AND V=VEC 10
     95        AND IA,IE = NIL,NIL
    100        AND SUB1(X,Y)=
    110        VALOF &( LET A,B=50,100
    120        IF (A+B)<(X+Y) RESULTIS X+Y
    130        RESULTIS A+B &)
    140        WRITE(6,C/'AUSGABE A=',I7/C,1,A1)
    145        WRITE(6,C/6(I7/)L,6,A1,A2,A3,B1,B2,B3)
    150        FOR I=1 TO 10
    160        &( A1:=A1 + M(B1+I*1)
    170        B2:=B2 + A1
    180        B3:=B2 * B2
    190        V11:=SUB1(B2,B3)
    200        WRITE(6,C/'VEKTOR : ',I7/C,1,V11)
    210        &) &) &)

UMFANG DER ZWISCHENSPRACHE:  1055 HALBWORTE
```

```
*UEBERSETZE
 1 QUELLE        = ENG&BCPL2
 2 SPRACHE       = BCPL
 3 NUMERIERUNG   = -STD-
 4 MD            = UP1
 5 VARIANTE      = 0
 6 PROTOKOLL     = R'A
 9 MV            = -STD-

START PS&BCPLCUMP (19.00)
```

Unterprogramm UP1
Referenzliste und Adreßbuch

```
MONTAGEOBJEKT (UP1)
     10        EXTERNAL M
     20        NONREC 10:M
     30        LET M(X) BE
     40        X:=X+1

UMFANG DER ZWISCHENSPRACHE:  405 HALBWORTE
```

```
LISTE DER VERABREDETEN NAMEN:

AUFBAU:
<NAME S.LZUG> <REFERENZ AUF DEFINITION> (/ <NONREC-KLASSE> /) <TYP> <BEDEZ,ADR,> <SYMBOL,ADR,> <REFERENZEN>
     (1)              (2)                                        (3)     (4)     (5)       (6)          (7)

TYP-ABKUERZUNGEN:

   EXTERNE      GLOBALE      INDEX-       STATISCHE    DYNAMISCHE   MANIFEST-    NONREC-
   VARIABLE:    VARIABLE:    VARIABLE:    VARIABLE:    VARIABLE:    KONSTANTEN   VEREINBARUNGEN

       E            G            X            F                         M            NR
       EF           GF           XF           R                                                 FUNKTION
       ER           GR           XR           L                                                 ROUTINE
       EL           GL           XL           S                                                 MARKE(LABEL)
       ES                                                                                        STATIC-VARIABLE

                                                 D                                             EINFACHE VARIABLE
                                                 P                                             PARAMETER
                                                 I                                             ZAEHLVARIABLE
                                                 V                                             VEKTORVARIABLE

                    (3)
     (1)      (2)    (4)    (4)   (5)   (6)      (7)

A . . . . . .  110          D     4     4      120     130
A1 . . . . . .  80          D     5     5      140     145     160     160     170
A2 . . . . . .  70    M     S           5      145
A3 . . . . . .  70    M     A          10      145
B . . . . . .  110          D     5     5      120     130
B1 . . . . . .  80          D     2     2      145     160
B2 . . . . . .  80          D     3     3      145     170     170     180     180     190
B3 . . . . . .  80          D     4     4      145     180     190
BL.EA . . . . .  20          E           1
ERG . . . . . .  60          S           L2
I . . . . . .  150          I    15    21      160     160     190     200
IA . . . . . .  95          D     7     7
IE . . . . . .  95          D     8     8
M . . . . . .  40E   NR 10  E           4      50NR    160
READ . . . . .  10          NR          1
 . . . . . .  20E   NR  1  E           3
START . . . . .  30          GL    1     1      40L
SUB1 . . . . .  190          E           L4     190
V . . . . . .  90          V     6     6      190     200
WRITE . . . . .  10NR  NR  1  E           2      20E     140     145     200
X . . . . . .  100          P     2     2      120     120
Y . . . . . .  100          P     3     3      120     120
ES WURDEN KEINE SYNTAXFEHLER FESTGESTELLT


START PSEBCPL2 (19.01)
MO STDHP WURDE ERZEUGT
```

Adreßbuch und Referenzliste des Hauptprogramms STDHP

```
LISTE DER VERWENDETEN NAMEN:

AUFBAU:
<NAMENSBEZUG> <REFERENZ AUF DEFINITION> (/ <NONREC-KLASSE> /) <TYP> <SEDEZ,ADR,> <SYMBOL,ADR,><REFERENZEN>

TYP-ABKUERZUNGEN:

    EXTERNE      GLOBALE      INDEX-       STATISCHE    DYNAMISCHE   MANIFEST-    NONREC-
    VARIABLE:    VARIABLE:    VARIABLE:    VARIABLE:    VARIABLE:    KONSTANTEN   VEREINBARUNGEN

       E            G            X                                      M            NR
       EF           GF           XF           F                                                 FUNKTION
       ER           GR           XR           R                                                 ROUTINE
       EL           GL           XL           L                                                 MARKE(LABEL)
       ES                                     S                                                 STATIC-VARIABLE

                                                           D                                     EINFACHE VARIABLE
                                                           P                                     PARAMETER
                                                           I                                     ZAEHLVARIABLE
                                                           V                                     VEKTORVARIABLE


M . . . . . .      10E   NR 10  ER          9        20NR      30R

X . . . . . .      30             P    2     2        40        40
ES WURDEN KEINE SYNTAXFEHLER FESTGESTELLT

START PS&BCPL2 (19,01)
MO UP1 WURDE ERZEUGT

ENDE PS&BCPL2 (19,01)  0,65
```

Adreßbuch und Referenzliste des Unterprogramms UP 1

2 Ablaufprotokollierung (Tracing)

Ist die Spezifikation TRACE im UEBERSETZE-Kommando besetzt, wird vom BCPL-Compiler eine Ablaufüberwachung für den Objektlauf eincompiliert. Dadurch ist eine zeilenbezogene Ablaufverfolgung möglich. Das Tracing kann auf Zeilenbereiche und/oder Statementarten beschränkt werden.

Nachstehende Spezifikationswerte bzw. -angaben werden bei der Ablaufprotokollierung ausgewertet:

TRACE = -STD- Standardangabe;
protokolliert werden alle RETURN- und RESULTIS-Anweisungen sowie Prozeduraufrufe bzw. -eingänge im gesamten Quellbereich.

TRACE = GOTO In die Ablaufprotokollierung werden alle RETURN und RESULTIS-Anweisungen aus dem gesamten Quellbereich aufgenommen.

TRACE = CALL Ablaufprotokollierung der Prozedur-Aufrufe bzw. -eingänge im gesamten Quellbereich angelistet.

TRACE = $(a_1 - e_1)$ ' $[(a_2 - e_2)]$ ...
gleichbedeutend mit TRACE = -STD- im Quellzeilenbereich a_1 bis e_1 und a_2 bis e_2.
Eine Quellzeilenbereichsangabe kann auch bei TRACE = GOTO und TRACE = CALL angegeben werden.

Das folgende Beispiel zeigt die Ablaufprotokollierung mit den Angaben

TRACE = CALL (160-190)' (200)' GOTO (120)

Angelistet werden die jeweiligen Aufrufe für die Funktionsnamen name, sowie die NONREC-Klasse, Anzahl der Parameter und der Wert des jeweils ersten Parameters und bei der Angabe GOTO der über die RETURN- bzw. RESULTIS-Anweisung zurückgelieferte Wert (sedezimal, bei Werten kleiner 2^{10} zusätzlich als Dezimalwert).

```
***** BCPL TRACING *****

ZEILE STM

   160  1     AUFRUF FN M : NR.KLASSE 10.   1 PARAM.
              ERSTER PARAMETER : ' 000003 '
   190  1     AUFRUF FN SUB1 : NR.KLASSE  0.   2 PARAM.
              ERSTER PARAMETER : ' 000008 '
   200  1     AUFRUF RT WRITE : NR.KLASSE  1.   4 PARAM.
              ERSTER PARAMETER : ' 000006 '

VEKTOR : 0000150
   160  1     AUFRUF FN M : NR.KLASSE 10.   1 PARAM.          TRACE = CALL
              ERSTER PARAMETER : ' 000006 '
   190  1     AUFRUF FN SUB1 : NR.KLASSE  0.   2 PARAM.       TRACE = CALL   DO-Schleife
              ERSTER PARAMETER : ' 000013 '
   120  3     RUECKKEHR FN SUB1 RESULTIS 00017C DEZ. WERT = 380TRACE = CALL   2. Durchlauf
   200  1     AUFRUF RT WRITE : NR.KLASSE  1.   4 PARAM.      TRACE = GOTO
              ERSTER PARAMETER : ' 000006 '

VEKTOR : 0000380
   160  1     AUFRUF FN M : NR.KLASSE 10.   1 PARAM.
              ERSTER PARAMETER : ' 00000B '
   190  1     AUFRUF FN SUB1 : NR.KLASSE  0.   2 PARAM.
              ERSTER PARAMETER : ' 00002A '
   120  3     RUECKKEHR FN SUB1 RESULTIS 0007E
   200  1     AUFRUF RT WRITE : NR.KLASSE  1.   4 PARAM.
              ERSTER PARAMETER : ' 000006 '

VEKTOR : 0001806
   160  1     AUFRUF FN M : NR.KLASSE 10.   1 PARAM.
              ERSTER PARAMETER : ' 000012 '
   190  1     AUFRUF FN SUB1 : NR.KLASSE  0.   2 PARAM.
              ERSTER PARAMETER : ' 000054 '
   120  3     RUECKKEHR FN SUB1 RESULTIS 001BE4
   200  1     AUFRUF RT WRITE : NR.KLASSE  1.   4 PARAM.
              ERSTER PARAMETER : ' 000006 '

VEKTOR : 0007140
   160  1     AUFRUF FN M : NR.KLASSE 10.   1 PARAM.
              ERSTER PARAMETER : ' 00001B '
   190  1     AUFRUF FN SUB1 : NR.KLASSE  0.   2 PARAM.
              ERSTER PARAMETER : ' 00009A '
   120  3     RUECKKEHR FN SUB1 RESULTIS 005D3E
   200  1     AUFRUF RT WRITE : NR.KLASSE  1.   4 PARAM.
              ERSTER PARAMETER : ' 000006 '

VEKTOR : 0023870
   160  1     AUFRUF FN M : NR.KLASSE 10.   1 PARAM.
              ERSTER PARAMETER : ' 000026 '
   190  1     AUFRUF FN SUB1 : NR.KLASSE  0.   2 PARAM.
              ERSTER PARAMETER : ' 000107 '
   120  3     RUECKKEHR FN SUB1 RESULTIS 010F38
   200  1     AUFRUF RT WRITE : NR.KLASSE  1.   4 PARAM.
              ERSTER PARAMETER : ' 000006 '

VEKTOR : 0069432

   160  1     AUFRUF FN M : NR.KLASSE 10.   1 PARAM.
              ERSTER PARAMETER : ' 000033 '
   190  1     AUFRUF FN SUB1 : NR.KLASSE  0.   2 PARAM.
              ERSTER PARAMETER : ' 0001A8 '
   120  3     RUECKKEHR FN SUB1 RESULTIS 02BFE8
   200  1     AUFRUF RT WRITE : NR.KLASSE  1.   4 PARAM.
              ERSTER PARAMETER : ' 000006 '
```

```
VEKTOR : o18o2oo
   16o  1     AUFRUF FN M : NR.KLASSE 1o.   1 PARAM.
              ERSTER PARAMETER : ' oooo42 '
   19o  1     AUFRUF FN SUB1 : NR.KLASSE  o.   2 PARAM.
              ERSTER PARAMETER : ' ooo28C '
   12o  3     RUECKKEHR FN SUB1 RESULTIS o67F1C
   2oo  1     AUFRUF RT WRITE : NR.KLASSE  1.   4 PARAM.
              ERSTER PARAMETER : ' oooooo '

VEKTOR : o425756
   16o  1     AUFRUF FN M : NR.KLASSE 1o.   1 PARAM.
              ERSTER PARAMETER : ' oooo53 '
   19o  1     AUFRUF FN SUB1 : NR.KLASSE  o.   2 PARAM.
              ERSTER PARAMETER : ' ooo3C4 '
   12o  3     RUECKKEHR FN SUB1 RESULTIS oE31D4
   2oo  1     AUFRUF RT WRITE : NR.KLASSE  1.   4 PARAM.
              ERSTER PARAMETER : ' oooooo '

VEKTOR : o93o26o
   16o  1     AUFRUF FN M : NR.KLASSE 1o.   1 PARAM.
              ERSTER PARAMETER : ' oooo66 '
   19o  1     AUFRUF FN SUB1 : NR.KLASSE  o.   2 PARAM.
              ERSTER PARAMETER : ' ooo563 '
   12o  3     RUECKKEHR FN SUB1 RESULTIS 1Do9AC
   2oo  1     AUFRUF RT WRITE : NR.KLASSE  1.   4 PARAM.
              ERSTER PARAMETER : ' oooooo '

VEKTOR : 19o3o2o  STDHP: TRACE = CALL (160-190) '(200)'GOTO (120)

ENDE STDHP  o.21
```

STDHP: TRACE = CALL (160-190) '(200)' GOTO (120)
UP1: ohne TRACE

Die Verwendung der Spezifikation TRACE = DEBUG [(a-b)] im UEBERSETZE-Kommando erlaubt das wahlweise Überlesen von Quellzeilen. Im Quelltext können gewünschte Statements mit der Zeichenfolge $T eingeleitet werden. Ohne Spezifikation TRACE = DEBUG wirkt diese Zeichenfolge $T wie das Kommentarzeichen // (Überlesen bis Zeilenende). Bei angegebener Spezifikation wird die Zeichenfolge $T ignoriert und damit der Rest der Zeile ausgewertet. Dadurch ist es möglich, Quelleinschübe für Testhilfen vorzusehen.

Beispiel:

```
$T  WRITE (6, [/'VEKTOR :' I7/[, 1, V ! I)
```

3 Rückverfolger

Der Rückverfolger ist ein Dienst, der ausgehend von einer Programmadresse, die Unterprogrammverschachtelung feststellt, die zum Erreichen dieser Programmadresse geführt hat.

Der Rückverfolger kann an beliebiger Stelle während des Programmlaufes (über Kontrollereignisse) oder nach dem Auftreten von Fehlern oder Alarmen aufgerufen werden; im Fehler- bzw. Alarmfall weist der Rückverfolger ausgehend von der fehlerhaften Adresse die aktuelle Unterprogrammverschachtelung auf.

```
MONTAGEUBJEKT  STUHP
      10        NUNREC 1: WRITE,READ
      20        EXTERNAL BL,EA:WRITE,READ
      30        GLUBAL &( START:1 &)
      40        STARI: &( EXTERNAL M
      50        NUNREC 10!M
      60        STATIC &( ERG=0 &)
      70        MANIFEST &( A2=5; A3=10 &)
      80        &( LET B1,B2,B3,A1=2,4,0,0
      90        AND V=VEC 10
     100        AND SUB1(X,Y)=
     110        VALUF &( LET A,B=50,100
     120        IF (A+B)<(X+Y) RESULTIS X+Y
     130        RESULTIS A+B &)
     140        WRITE(0,L/'AUSGABE A=',I7/L,1,A1)
     145        WRITE(0,L/6(I7/)L,6,A1,A2,A3,B1,B2,B3)
     150        FUR I=1 TO 10
     160        &( A1:=A1 + M(B1+I*I)
     170        B2:=B2 + A1
     180        B3:=B2 * B2
     190        V!1:=SUB1(B2,B3)
     200        WRITE(0,L/'VEKTUR : ',I7/L,1,V!I)
     210        &) &) &)


MONTAGEUBJEKT  UP1
      10        EXTERNAL M
      20        NUNREC 10:M
      30        LET M(X) BE
      40        X:=X/0
```

```
(1)  FEHLER: ARITHMETISCHER ALARM. MOEGLICHE FEHLERSITUATION:

                                        FESTKOMMADIVISION DURCH 0

     RUECKVERFOLGER:
       2. BCPL-PROZ    *M              ADRESSE  1-    E   ZEILE   40-1   <--
(2)    1. BCPL-HP      STDHP           ADRESSE  1-   50   ZEILE   160-1

     ENDE PSLRUECKVERF (15.00)  0.13
```

Das Beispiel zeigt die aktuelle Unterprogrammverschachtelung im Fehlerfall: der Fehler tritt in der externen Prozedur M in Zeile 40 im ersten Statement auf. Der Aufruf der Prozedur M erfolgt in Zeile 160 des Hauptprogramms STDHP.
Um den Rückverfolger zu starten, ist es notwendig, daß im UEBERSETZE-Kommando die vorbesetzte Spezifikation VARIANTE = D nicht überschrieben wird (sonst muß die Spezifikation explizit angegeben werden).

4 Quellbezogener Dump

Der quellbezogene Dump kann beliebig während des Programmlaufes und nach Auftreten von Fehlern und Alarmen aufgerufen werden.
Die Dumps erlauben auch während des Programmlaufs die Werte einzelner Variablen zu erfragen und zu ändern.

Für den BCPL-Quelldump können nachstehende Spezifikationswerte im STARTE-Kommando angegeben werden:

¤ STARTE,DUMP = BL-ALLES [(a)]	Dump der Variablen aller Montageobjekte mit Ausnahme von a
BL-NEST [(a)]	Dump aller Variablen der an der aktuellen Aufrufverschachtelung beteiligten Montageobjekte mit Ausnahme von a
BL-TEIL [(a)]	Dump aller Variablen des aktuellen Montageobjekts mit Ausnahme von a
BL-NICHTS (a)	Kein Dump mit Ausnahme von a
BL-KONSOL (a)	Dump von a auf dem Terminal
BL-BRINGE (a)	Wie KONSOL, jedoch ohne Start- und Endemeldung des BCPL-Dumps
BL-SETZE (a)	Die Werte der Variablen a werden umbesetzt; protokolliert die Variablen vor der Umbesetzung a: Dumpeinschränkung besteht aus einer geklammerten Folge von Montageobjektangaben, Prozedurangaben und/oder Variablenangaben. Mehrere Angaben durch Komma trennen.

Mehrere Spezifikationswerte sind durch Apostroph zu trennen; dabei ist zu beachten, daß pro Sprache nur die zuletzt gemachte Angabe Gültigkeit hat.

Das nachstehende BCPL-Programm wurde so abgeändert, daß es einen Speicherschutzalarm verursacht (A). Der Rückverfolger zeigt die aktuelle Programmverschachtelung - ausgehend von der fehlerhaften Adresse - auf (B). Dadurch ist bereits eine Einkreisung des Fehlers vorgenommen und zwar im Hauptprogramm STDHP, Quellzeile 200.
Die Besetzung der Dumpspezifikation: DUMP = BL-ALLES bewirkt, daß alle Variablen des Hauptprogramms STDHP und des Unterprogramms UP1 zum Fehlerzeitpunkt angelistet werden, getrennt nach statischen und dynamischen Variablen (C).

```
MONTAGEOBJEKT  STDHP
      10        NONREC 1: WRITE,READ
      20        EXTERNAL BL,EA:WRITE,READ
      30        GLOBAL &( START:1 &)
      40        START: &( EXTERNAL M
      50        NONREC 101M
      60        STATIC &( EHG=0 &)
      70        MANIFEST &( A2=5; A3=10 &)
      80        &( LET B1,B2,B3,A1=2,4,6,0
      90        AND V=VEC 10
      95        AND IA,IE = NIL,NIL
     100        AND SUB1(X,Y)=
     110        VALOF &( LET A,B=50,100
     120        IF (A+B)<(X+Y) RESULTIS X+Y
     130        RESULTIS A+B &)
     140        WRITE(6,[/'AUSGABE A=',I7/[,1,A1)
     145        WRITE(6,[/6(I7/)[,6,A1,A2,A3,B1,B2,B3)
     146        IA := 0
     147        IE := 20
     150        FOR I=IA TO IE
     160        &( A1:=A1 + M(B1+I*I)
     170        B2:=B2 + A1
     180        B3:=B2 * B2
     190        V!I:=SUB1(B2,B3)
     200        WRITE(6,[/'VEKTOR : ',I7/[,1,V!I)
     210        &) &) &)

MONTAGEOBJEKT  UP1
      10        EXTERNAL M
      20        NONREC 101M
      30        LET M(X) BE
      40        X:=X+1
```

```
¤ MONT.,MO=STDHP'UP1
¤ STARTE,STDHP,DUMP=BL-ALLES

START STDHP

AUSGABE A=0000000

0000000
0000005
0000010
0000002
0000004
0000006

0  VEKTOR : 0000150

1  VEKTOR : 0000210

2  VEKTOR : 0000612

3  VEKTOR : 0002970

4  VEKTOR : 0009900

5  VEKTOR : 0029756

6  VEKTOR : 0080940

7  VEKTOR : 0201152

8  VEKTOR : 0461720

9  VEKTOR : 0989030

10 VEKTOR : 1995156
```

```
(A) || FEHLER : SPEICHERSCHUTZ-ALARM.
(B) || RUECKVERFOLGER:
       1. SCPL=MP      (STDMP)          ADRESSE  1=  74   ZEILE  (200=1)
    || ENDE RUECKVERF (15.00)  0.12
```

(C)

(1) MO STDMP	(2)	(3)	(4)	(5)	(6)	(7)	(8)			(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	
START	90	2D11	1	000000	0	F				START	40	2D11	1	000000	0	F		1	000000	STATIC
ERG	90	2B2E	1	FFFFFF	0	T				SUB1	100	2B2F	1	000009	9	T		1	000009	STATIC
MO STDMP																				
B1	80	5802	1	000002	2	F				B2	80	5803	1	0007A2	1954	F	[			
B3	80	5804	1	9A4284	9818116	F				A1	80	5805	1	00021E	542	F				
V	90	5806	1	00580A	22538	F				CCCV	90	580A	1	000096	150	F		V10		DYNAMIC
	90	5808	1	000002	210	F	S	V11			90	580C	1	00032C	812	F		V12		DYNAMIC
	90	580D	1	00089A	2970	F		V13			90	580E	1	0026AC	9900	F	]	V14		DYNAMIC
	90	580F	1	007436	29750	F	]	V15			90	5810	1	013C2C	80940	F		V16		DYNAMIC
	90	5811	1	031100	201152	F	A	V17			90	5812	1	070B98	461720	F	!	V18		DYNAMIC
	90	5813	1	0F1760	989030	F	!	V18			90	5814	1	1E7194	1995156	F	!	V110		DYNAMIC
IA	95	5807	1	FFFFFF	0	T				IB	95	5808	1	000014	20	F				
I	190	5815	1	9A4A20	(9820070)	F														

```
   || ENDE PSEGCPLLDMP (0.01)  0.40

   ENDE STDMP  0.13

   ++++++OPERATORLAUF MIT FEHLER BEENDET: STDMP
```

(1) **Vereinbarte Namen im Quellprogramm**

(2) **Bezugnahme auf Quellstatement**

(3) **Adresse**

(4) **Typenkennung**

(5) **Wert sedezimal**

(6) **Wert dezimal**

(7) **Wert logisch (TRUE, FALSE)**

(8) **Wert als Text gedeutet**
(nicht druckbare Zeichen werden als _ abgebildet)

(9) (10) **Initialwert bei Prozedur- oder Markenvariable**

Interpretation des Beispiels: C

Die dynamischen Variablen belegen die Speicheradressen '5802' bis '5815'. Anfangs werden die explizit vereinbarten Variablen (B1, B2, B3, A1, Vektor V, IA und IE) gespeichert. Auf der Adresse '5806' steht der Adress-Verweis des Vektors V, wo dessen Elemente V ! 0 bis V ! 10 liegen.

Die implizit vereinbarte Laufvariable I belegt die Speicheradresse '5815'.

In der vom Rückverfolger ausgewiesenen Quellzeile 200 werden die Variablen I und V ! I angesprochen. Der Inhalt der Laufvariablen I enthält einen ungültigen Wert. Die Bezugnahme auf V ! I in Quellzeile 200 adressiert mithin auf eine unzulässige Speicherzelle (V ! 3820070) und verursacht den Speicherschutz-Alarm. Der Vektor V besitzt 11 Vektorelemente, die die entsprechenden Speicherzellen '580A' bis '5814' belegen; die Laufvariable I für I = 11 speichert den in Quellzeile 190 gelieferten Funktionswert auf die Speicheradresse, die dem 12. Vektorelement entsprechen würde, nämlich auf Adresse '5815'. Diese Adresse erklärt aber den R-Wert für die Laufvariable I, der nun überschrieben wird und die eigentliche Fehlerquelle bildet.

4.1 Spezifikationen des quellbezogenen Dumps

s. folgende Seiten

Modus:

BL-ALLES[(a)]	:	Dump der Variablen aller Montageobjekte mit Ausnahme von a
BL-NEST[(a)]	:	Dump aller Variablen der an der aktuellen Aufrufverschachtelung beteiligten Montageobjekte mit Ausnahme von a
BL-TEIL[(a)]	:	Dump aller Variablen des aktuellen Montageobjekts mit Ausnahme von a
BL-NICHTS (a)	:	Kein Dump mit Ausnahme von a
BL-KONSOL (a)	:	Dump von a auch auf dem Terminal
BL-BRINGE (a)	:	Dump von a auch auf dem Terminal, jedoch ohne Start- und Endemeldung des BCPL-Dumps
BL-SETZE (a)	:	Die Werte der Variablen a werden neu besetzt; protokolliert werden die Werte vor der Umbesetzung

a: Geklammerte Folge von Montageobjekt-, Prozedur- und/ oder Variablenangaben; mehrere Angaben innerhalb der Klammer durch Komma trennen.

Wirkung:

Der BCPL-Dump dient der Ausgabe der aktuellen Inhalte von Variablen ins Ablauf- und Terminalprotokoll. Die Leistungen des BCPL-Dumps werden durch die Angabe eines Dumpstrings in der Spezifikation DUMP der Kommandos STARTE, RECHNE und THSETZE, oder als Gesprächseingabe bei Kontrollereignissen, gesteuert. Erbracht werden können diese Leistungen jedoch nur, wenn bei der Übersetzung der Quelle die Spezifikation VARIANTE mit D oder GS besetzt war.
Die Modi KONSOL, BRINGE und SETZE sind nur bei dynamischem Start des Dumpoperators durch den Terminalbenutzer während eines Gesprächs sinnvoll.
Die Einschränkung a kann beliebig viele Angaben enthalten. Jede Angabe kann die drei Qualifizierungsstufen

- Montageobjekt
- Prozedur
- Variable

enthalten.
Die Qualifizierung braucht, wenn keine Namenskollision auftritt, nicht unbedingt vollständig zu sein. Verschiedene Qualifizierungsstufen werden durch runde Klammern getrennt, wobei, wenn nötig die Reihenfolge

Montageobjekt (Prozedur(Variable))

ist. Auf jeder Stufe des Klammergebirges können Einschränkungen, die gleiche Qualifizierungen in einer höheren Stufe haben, durch Komma getrennt angegeben werden. Sollen von einer Prozedur mehrere Variable qualifiziert werden und ist die Qualifizierungsstufe Prozedur vorhanden, so müssen in der Klammer hinter dem Prozedurnamen alle zu qualifizierenden Variablen dieser Prozedur aufgelistet werden. Will man alle Variablen einer Prozedur bzw. alle Prozeduren eines Montageobjekts qualifizieren, muß man eine leere Klammer hinter den Prozedur- bzw. Montageobjektnamen setzen.

Wirkung:

Qualifizierungsmöglichkeiten:

In den Modi NICHTS, KONSOL, BRINGE und SETZE sind zusätzlich zu den bisher möglichen Einschränkungen, auf der Qualifizierungsstufe "Variable" folgende spezielle Variablenspezifikationen zugelassen:

- Offsetspezifikation:

  Es wird ein BCPL-Element gedumpt, das relativ zur operatorrelativen Adresse einer Variablen liegt.
  Z. B.: BL-KONSOL (&&& A + 3)
  Gedumpt wird das Element mit der Adresse)&&&A(+3.
  Als Name wird &&&A angegeben. Zur Kennzeichnung, daß dies nicht der Originalname des gedumpten BCPL-Elements ist, wird zwischen Name und Zeilennummer ein * ausgedruckt.

- Bereichsspezifikation:

  Es wird ein Bereich relativ zum Variablenelement gedumpt.
  Z. B. : BL-KONSOL (&&&A(5 : 8))
  Gedumpt werden die vier BCPL-Elemente mit den Adressen)&&&A(+ 5 bis)&&&A(+ 8.
  Als Name wird wieder &&&A und wie oben ein * ausgedruckt.

- Vektorzugriff:

  Er ermöglicht den Zugriff auf einzelne oder (Bereich) mehrere BCPL-Elemente, die über einen Pointer adressiert sind (Vektoren, Tables).
  Z. B. : BL-KONSOL (A ! 6)
  Gedumpt wird das 7. Element eines Vektors, dessen Anfangsadresse in der Variablen A steht. Der Name des BCPL-Elements ist A. Als Kennzeichen, daß dies nicht der ursprüngliche Namen des gedumpten BCPL-Elements ist, wird zwischen Name und Zeilennummer ein * ausgedruckt.

- Ersetzungsspezifikation:

  Es wird eine Adreßersetzung durchgeführt (RV-Operation).
  Z. B. : !A
  Gedumpt wird das Element, dessen Adresse Inhalt der Variablen ist. Als Name wird wieder A und wie oben ein * ausgedruckt.

- Setz-Spezifikation (siehe formale Beschreibung):

  Ist auf ein BCPL-Element kein Schreibschutz gesetzt, so kann in allen speziellen Variableneinschränkungen, durch Anfügen von Gleichheitszeichen und gewünschtem Wert, der ursprüngliche Inhalt des Elements verändert werden. Im Protokoll erscheint der Wert des Elements vor der Umsetzung.

formal:

⟨BCPL-Dump⟩ ::= BL- { { ALLES / TEIL / NEST } [(⟨BL-Einschränkung⟩[,⟨BL-Einschränkung⟩]∞)] / { NICHTS / KONSOL / BRINGE / SETZE } (⟨spez. BL-Einschränkung⟩ [,⟨spez. BL-Einschränkung⟩]∞) }

⟨BL-Einschränkung⟩ ::= { ⟨Montageobjekteinschränkung⟩ / ⟨Prozedureinschränkung⟩ / ⟨Variableneinschränkung⟩ }

⟨spez. BL-Einschränkung⟩ ::= { ⟨Montageobjekteinschränkung⟩ / ⟨Prozedureinschränkung⟩ / ⟨spez. Variableneinschränkung⟩ }

⟨Montageobjekteinschränkung⟩ ::= ⟨Montageobjektnamen⟩ ()

⟨Prozedureinschränkung⟩ ::= { ⟨Prozedurnamen⟩ ( ) / ⟨MO-Namen⟩(⟨Prozedurliste⟩) }

⟨Prozedurliste⟩ ::= ⟨Prozedurnamen⟩ ( ) [,⟨Prozedurnamen⟩ ( )]∞

⟨Variableneinschränkung⟩ ::= { ⟨Variablenname⟩ / ⟨Prozedurname⟩(⟨Variablenliste⟩) / ⟨MO-Name (⟨Variablenliste⟩) / ⟨MO-Name (⟨Prozedurname⟩(⟨Variablenliste⟩)) }

⟨Variablenliste⟩ ::= ⟨Variablenname⟩[,⟨Variablenname⟩]∞

⟨spez. Variableneinschränkung⟩ ::= { ⟨Variableneinschränkung⟩ / ⟨Prozedurname⟩(⟨spez. Variablenliste⟩) / ⟨MO-Name (⟨spez. Variablenliste⟩) / ⟨MO-Name (⟨Prozedurname⟩(⟨spez. Variablenliste⟩)) }

⟨spez. Variablenliste⟩ ::= ⟨spez. Variablenspezifikation⟩[,⟨spez. Variablenspezifikation⟩]∞

⟨spez. Variablenspezifikation⟩ ::= { ⟨Variablenliste⟩ / ⟨Offsetspezifikation⟩ / ⟨Bereichsspezifikation⟩ / ⟨Vektorzugriff⟩ / ⟨Ersetzungsspezifikation⟩ / ⟨Setzspezifikation⟩ }

⟨Offsetspezifikation⟩ ::= ⟨Variablenname⟩ { [+] / - } { ⟨Dezimalkonstante⟩ / ⟨Hexadezimalkonstante⟩ }

⟨Bereichsspezifikation⟩ ::= ⟨Variablenname⟩ ({ [+] / - } { ⟨Dezimalkonstante⟩ / ⟨Hexadezimalkonstante⟩ } : { [+] / - } { ⟨Dezimalkonstante⟩ / ⟨Hexadezimalkonstante⟩ })

⟨Vektorzugriff⟩ ::= ⟨Variablenname⟩ ! { { [+] / - } { ⟨Dezimalkonstante⟩ / ⟨Hexadezimalkonstante⟩ } / ({ [+] / - } { ⟨Dezimalkonstante⟩ / ⟨Hexadezimalkonstante⟩ } : { [+] / - } { ⟨Dezimalkonstante⟩ / ⟨Hexadezimalkonstante⟩ }) }

⟨Ersetzungsspezifikation⟩ ::= ! ⟨Variablenname⟩

⟨Setzspezifikation⟩ ::= { ⟨Variablenname⟩ / ⟨Offsetspezifikation⟩ / ⟨Bereichsspezifikation⟩ / ⟨Vektorzugriff⟩ / ⟨Ersetzungsspezifikation⟩ } = { { [+] / - } { ⟨Dezimalkonstante⟩ / ⟨Hexadezimalkonstante⟩ } / ⟨Charakterkette⟩ }

Beispiele:

```
BL-ALLES (MO2 (PROZ1 (A, B)))
```

Dump der Variablen aller Montageobjekte mit Ausnahme der Variablen A und B der Prozedur PROZ1 im Montageobjekt MO2.

5 Kontrollereignisse

Kontrollereignisse (KE's) sind Hilfsmittel, mit denen ein Benutzer während des Programmlaufs sein Programm gezielt überwachen und steuern kann. Mit der Definition von Kontrollereignissen ('break points') ist es mithin möglich, Programmunterbrechungen an beliebig gewünschter Stelle im Objektlauf zu veranlassen, um bestimmte Kontroll-Anweisungen an den Operatorlauf geben zu können. Diese Gesprächsfähigkeit des Operators setzt voraus, daß im UEBERSETZE-Kommando die Spezifikation VARIANTE = GS besetzt wird; diese Besetzung beinhaltet gleichzeitig die Dumpfähigkeit des Operators.

Die für BCPL anwendbaren Kontrollereignisse (KE's) lassen sich in ereignisbezogene KE's und in frei wählbare KE's gliedern.

5.1 Ereignisbezogene und frei wählbare Kontrollereignisse

Ereignisbezogene KE's: sie bedürfen nicht der Vereinbarung. Sie sind immer wirksam, wenn die VARIANTE = GS ist und im STARTE-Kommando die KE's gesetzt sind, z. B.

␣STARTE, STDHP,..., AKTIV = ALLE␣.

Ereignisbezogene KE's sind:

START	zu Beginn eines Programms
ENDE	bei Ende eines Programms
FEHLER	bei logischen Programmfehlern, (z. B. EA-Fehler oder durch DYNKON erkannte Fehler)
ALARM	bei Hardware- und Ereignisalarmen (z. B. TK-Alarm, Zeitüberschreitung, Betriebsmittelüberschreitung)

(Die KE's FEHLER und ALARM sind immer aktiv, auch wenn im STARTE-Kommando AKTIV = KEINE vereinbart wird).

Frei wählbare Kontrollereignisse: sie können auf zwei Arten vereinbart werden

- in der Quelle durch Prozeduraufruf (vgl. 5.6)
  oder
- sie können dynamisch während des Objektlaufs definiert werden (vgl. 5.3).

Wichtig!

Kontrollereignisse für BCPL können im Unterschied zu anderen Sprachen nicht im UEBERSETZE-Kommando vereinbart werden, sondern werden erst beim Start des Objektlaufs definiert bzw. dann, wenn sich das erste ereignisbezogene KE: START meldet, z.B.

```
:
GIB KOMMANDOS␣: STARTE, STDHP,..., AKTIV = ALLE␣.
START STDHP
STDHP * KE = START␣:        (es meldet sich das erste ereignisbezogene KE
                             START)
```

Wenn sich ein KE meldet, so können folgende Reaktionen vorgenommen werden:

Kommandos:

- Eingabe von Kommandos, die vorrangig ausgeführt werden; nach der Ausführung der Kommandos bleibt die KE-Anfrage weiterhin aktiv.

Anweisungen:

- Fortfahren im Programm, z.B. durch Eingabe einer Leeranweisung, der Anweisung WEITER oder BEENDE:

  STDHP * KE = START␣ : ␣ .
  oder
  STDHP * KE = START␣ : WEITER␣.
  oder
  STDHP * KE = START␣ : BEENDE␣.

- Definition von Kontrollereignissen, z. B.

  STDHP * KE = START␣ : BL-ZEILE (200)␣.

  die KE-Anfrage bleibt weiterhin aktiv, solange bis Leeranweisung oder eine andere Anweisung gegeben wird.

5.2 Anweisungen

Anweisungen beginnen im Gegensatz zu Kommandos nicht mit einem Fluchtsymbol. Sie können einzeln oder als Anweisungsfolge eingegeben werden. Die Kontrollereignisprozedur S&KEP meldet sich nach der Entgegennahme einer Anweisung erneut und zwar solange, bis eine leere Eingabe (oder WEITER) gemacht wird; die einzelnen Anweisungen werden bei einer Anweisungsfolge durch Semikolon getrennt, z. B.

:

STDHP * KE = START␣ : RUECK; BL-BRINGE (A) ␣.

Die wichtigsten Anweisungen sind:

1. Fortfahren an Unterbrechungsstelle

   Anweisung: leere Eingabe "␣."
   Der Operator wird an der Unterbrechungsstelle fortgesetzt.

Anweisung: WEITER␣.
Der Operator wird an der Unterbrechungsstelle fortgesetzt. Wenn die Anweisung als Reaktion auf eine Kontrollereignismeldung eines Alarms oder einer HALT-Anweisung am Terminal gegeben wird, erfolgt ein Weiterstart nach Alarm.

2. Sofortiges Beenden eines Operatorlaufs ohne Angabe von Dumps.

Anweisung: BEENDE␣.
Eventuell angemeldete Abschlußprozeduren werden der Reihe nach abgearbeitet. Der Operatorlauf wird ohne Endemeldung beendet.

Anweisung: OPSTOP␣.
Der Operatorlauf wird mit Endemeldung beendet.

3. Beenden eines Operatorlaufs mit den im Startsatz vorgesehenen Dumps.

Anweisung: OPABBRUCH[⟨Anzahl⟩]␣.
⟨Anzahl⟩ ::= natürliche Zahl zwischen 1 und 65535.
Ist das Montageobjekt S&BACKTRACE anmontiert, wird die angegebene Anzahl der letzten überwachten Schritte ausgegeben. Wird die Anzahl nicht angegeben, so wird implizit 20 eingesetzt. Darauf wird der Rückverfolger gestartet und die im Startsatz vorgesehenen Dumps werden ausgegeben. Der Operatorlauf wird mit Endemeldung beendet.

Nach allen im folgenden beschriebenen Anweisungen an einen gesprächsfähigen Operator wird nach Ausführung dieser an der Unterbrechungsstelle gewartet, bis eine weitere Anweisung eintrifft. Trat die Unterbrechung in Folge eines eingetretenen Kontrollereignisses ein, so wird die Meldung des Kontrollereignisses wiederholt. Eine Anfrage nach weiteren Anweisungen unterbleibt erst nach Eingabe der vorher beschriebenen Anweisungen.
Auf diese Weise ist es möglich, mit mehreren Anweisungen auf ein Kontrollereignis zu reagieren.
Zu den folgenden Anweisungen sei noch bemerkt, daß eine Ausgabe auf dem Drucker (Dumps, Tracing, etc.) im Gespräch nur erfolgt, wenn das Druck-

protokoll eingeschaltet war (Siehe Kommando DRPROTOKOLL, ZUSTAND = EIN).

4. Ausgabe der im Startsatz vorgesehenen Dumps ins Druckerprotokoll oder auf dem Terminal.

   Anweisung: DUMPE␣.
   Es wird der Rückverfolger gestartet und die vorgesehenen Dumps werden ins Druckerprotokoll ausgegeben.

   Anweisung: KDUMPE␣.
   Es wird der Rückverfolger gestartet und die vorgesehenen Dumps werden auf dem Terminal ausgegeben.

5. Start eines in der Anweisung spezifizierten Dumps

   Anweisung: [K] DUMPE (BCPL-Dump)␣.

Beschreibung der Dumps siehe Abschnitt 4 (Quelldump). Es wird der in der Anweisung spezifizierte Dump ausgeführt. Hierbei ist zu beachten, daß der Dump nur bei der Dumpanweisung KONSOL auf dem Terminal ausgegeben wird.

6. Verkürzte Schreibweise für den Start eines in der Anweisung spezifizierten Dumps der Modi ALLES, NICHTS, KONSOL, BRINGE und SETZE.

   Anweisung: [⟨Sprachspezifikation⟩-]⟨Dump-Modus⟩
   [(⟨Einschränkung⟩)]␣.

   ⟨Sprachspezifikation⟩ ::= BL für BCPL

   ⟨Dumpmodus⟩ ::= { ALLES | NICHTS | KONSOL | BRINGE | SETZE }

   ⟨Einschränkung⟩ ::= (siehe Abschnitt 4, Dumps)

Beim Binärdump ist die Angabe zur Sprachspezifikation immer erforderlich. Wird diese Angabe nicht gemacht, entscheidet der Sprachschlüssel des in der Aufrufverschachtelung zuerst gefundenen dumpfähigen Montageobjektes, welcher Dump gestartet wird.

z. B.: T-KONSOL (A, B)␣.
KONSOL (A, B)␣.

7. Information über die Aufrufverschachtelung eines Operators.

Anweisung: RUECKV␣.
Der Rückverfolgungsoperator wird gestartet.

8. Information über die momentane Registerbelegung.

Anweisung: REGISTER␣.
Der aktuelle Registerstand an der Unterbrechungsstelle des Programms wird ausgegeben.

9. Analyse eines Alarms

Anweisung: ANALYSE␣.
Über einen Alarm wird im Klartext informiert. Der Alarmkeller wird ausgegeben. Liegt kein Alarm vor, ist die Anweisung wirkungslos.

10. Trace-Steuerung

Anweisung: { TRACEEIN | KTRACEEIN | TRACEAUS | KTRACEAUS } [(⟨Art⟩ [, ⟨Art⟩]$^{\infty}$)]␣.

⟨Art⟩ ::= GOTO | CALL (vgl. Abschnitt 3, Tracing)

Die Anweisungen sind nur wirksam, wenn im UEBERSETZE-Kommando unter TRACE eine Überwachung vereinbart wurde. Im Grundzustand ist die Überwachung, wie im UEBERSETZE-Kommando definiert, eingeschaltet. Es kann nur die gesamte Überwachung jeweils einer Art geändert werden.

Mit TRACEEIN und TRACEAUS werden die angegebenen Arten der Überwachung ein- bzw. ausgeschaltet. Ist keine Art angegeben, gilt es für alle Arten. KTRACEEIN und KTRACEAUS haben dieselbe Wirkung, es wird zusätzlich die Überwachung auch auf dem Terminal protokolliert. Dabei ist zu beachten, daß die Protokollierung der gesamten Überwachung nur auf dem Drucker oder auf dem Drucker und dem Terminal erfolgt, je nach dem, ob die letzte Einschaltanweisung TRACEEIN oder KTRACEEIN hieß.

11. Backtrace-Steuerung

Anweisung: $\left\{\begin{matrix}\text{BTRACEEIN}\\ \text{BTRACEAUS}\end{matrix}\right\}$ ⊓.

Die Anweisungen sind nur wirksam, wenn im UEBERSETZE-Kommando unter TRACE eine Überwachung vereinbart wurde. Mit BTRACEEIN und BTRACEAUS wird das Backtracing ein- bzw. ausgeschaltet. Hierbei werden im Fehlerfall die letzten 20 überwachten Schritte ausgegeben. Es werden dabei alle unter TRACE im UEBERSETZE-Kommando vereinbarten Überwachungen berücksichtigt.

Anweisung: $\left\{\begin{matrix}\text{BTRACE}\\ \text{BACKTRACE}\end{matrix}\right\}\left[\left\{\begin{matrix}\text{(KO[,⟨Anzahl⟩])}\\ \text{(⟨Anzahl⟩[,KO])}\end{matrix}\right\}\right]$ ⊓.

⟨Anzahl⟩ ::= natürliche Zahl zwischen 1 und 65535

Die Anweisung ist nur wirksam, wenn zuvor Backtracing eingeschaltet wurde.
Es erfolgt eine einmalige Ausgabe der letzten 20 überwachten Schritte, oder wenn eine Angabe zu Anzahl gemacht wird, werden dem entsprechend viele Überwacherschritte ausgegeben. Durch den Zusatz KO erfolgt die Ausgabe auch auf Terminal.
Es besteht jederzeit die Möglichkeit, von Backtracing zum normalen Tracing oder umgekehrt überzugehen. Hierzu ist es nur nötig, die jeweilige Einschaltanweisung zu geben (TRACEEIN, KTRACEEIN oder BTRACEEIN).

12. Anweisung STOP

Diese Anweisung ist nur in Verbindung mit vordefinierten Reaktionen auf ein Kontrollereignis (Spezifikation KONTROLLE des STARTE-Kommandos bzw. Anweisung KONTROLLE)* sinnvoll. Die Anweisung bewirkt, daß sich trotz vordefinierter Reaktion, das Kontrollereignis am Terminal meldet. Man unterscheidet dabei zwei Fälle:

a) Löschen einer vorgegebenen Reaktion und Anhalten des Programms

```
␣STARTE, A=ALLE, KONTROLLE=KE1-REGISTER␣.
    :
STDHP*KE=START␣: KONT. (KE1-STOP)␣.
STDHP*KE=START␣:␣:
STDHP*KE=KE1␣:
```

Die vorgegebene Reaktion REGISTER auf das Kontrollereignis KE1 wird ignoriert und das KE meldet sich am Terminal.

b) Anhalten nach vordefinierter Reaktion

```
STDHP*KE=KE1␣: KONTR. (KE2-REG. (A)); STOP␣.
STDHP*KE=KE1␣:␣.
    :
1 00000000000A      A
STDHP*KE=KE2␣:
```

Die vordefinierte Reaktion wird ausgeführt und das Kontrollereignis KE2 meldet sich am Terminal.

*) vgl. 5.4 Vordefinierte Reaktion auf KE

5.3 Definition von Kontrollereignissen

In BCPL können an allen Stellen des Programmlaufs KE's vereinbart werden; es besteht zusätzlich die Möglichkeit, dynamische KE's zu definieren.

Beim Erreichen eines KE's kann man neue KE's definieren; als Bedingungen, die dabei zu KE's führen sollen sind angebbar:

- das Erreichen der 1. Anweisung der Zeile n (im aktuellen HP, im Unterprogramm UP) oder ein Sprung auf diese Anweisung soll ein KE auslösen.

  BL-ZEILE (UP(n))
  BL-ZEILE (n)

  Diese Schreibweise entspricht der in anderen Sprachen üblichen Definition von KE's bei der Übersetzung. Sie bietet darüber hinaus auch die Möglichkeit, einen Sprung auf eine Marke auch ohne Backtracing festzustellen.

- das Erreichen des Unterprogramms UP soll ein KE auslösen

  BL-KE (UP)

- der Zugriff auf eine Variable a oder ein Feld f eines Unterprogramms UP soll ein KE auslösen

  TEST * KE (UP ({a / f}))

  Damit kann man Zugriffe auf Variable und Felder überwachen.
  Dabei werden nicht nur die Referenzen über die angegebenen Namen (außer EXTERNAL vereinbarte Namen) geprüft, sondern alle Zugriffe auf zugeordnete Speicherbereiche (wodurch auch indirekte Zugriffe erkannt werden).

- nach Ablauf von n Befehlen im Objekt soll ein KE auftreten

  TEST * KZ (n)

  Damit kann man ein Programm für Schnappschüsse unterbrechen, sich einen groben Überblick über den Programmlauf verschaffen oder sich an den vermuteten Fehler herantasten. Durch das Ändern von n lassen sich die Abtaststellen verfeinern und erweitern.

Werden dynamische KE's definiert, so werden zunächst die externen Angaben in interne Angaben umgerechnet und damit der Überwacher versorgt. Voraussetzung für das dynamische Definieren von KE's ist, daß das Montageobjekt S&UEBERWACHE anmontiert wurde (Spezifikation TRACE im UEBERSETZE-Kommando oder (Zwangs-) Montage durch MONTIERE-Kommando in Spezifikation MO = ... ' S&UEBERWACHE).

Die Definition von Kontrollereignissen wird im Objektlauf vorgenommen (nicht im UEBERSETZE-Kommando).

5.4 Aktivieren und Passivieren von Kontrollereignissen

Anweisung: { KEAKTIV | AKTIV | KEPASSIV | PASSIV } (⟨KE-Angabe⟩ [,⟨KE-Angabe⟩]$^{\infty}$)□.

⟨KE-Angabe⟩ ::= ⟨KE-Bezeichnung⟩ [-⟨Aktivierungsdurchlauf⟩]

⟨KE-Bezeichnung⟩ :: { ⟨Buchstabe⟩ [{⟨Buchstabe⟩ | ⟨Ziffer⟩}]^{5} | ⟨Ziffer⟩ [⟨Ziffer⟩]^{5} }

⟨Aktivierungsdurchlauf⟩ ::= ⟨natürliche Zahl zwischen 1 und 65535⟩

Ist ein Operator mit Kontrollereignissen mit dem STARTE-Kommando ohne Besetzung der Spezifikation AKTIV gestartet worden, so sind alle definierten Kontrollereignisse passiv außer den ereignbezogenen KE's ALARM und FEHLER.

Die Angabe zum Aktivierungsdurchlauf bewirkt, daß das Kontrollereignis erst nach der angegebenen Anzahl von Durchläufen aktiv wird, unabhängig, ob die Anweisung AKTIV (KEAKTIV) oder PASSIV (KEPASSIV) lautet. Alle unter AKTIV oder KEAKTIV angeführten Kontrollereignisse ohne Durchlaufangabe werden sofort aktiv, wie alle unter PASSIV bzw. KEPASSIV angegebenen Kontrollereignisse sofort passiv sind.

Globales Aktivieren und Passivieren von Kontrollereignissen:

Anweisung:

$$\left\{\begin{matrix}\text{AKTIV}\\\text{PASSIV}\end{matrix}\right\}\left(\left\{\begin{matrix}\text{ALLE}\\\text{KEINE}\end{matrix}\right\}\left[(\langle\text{KE-Einschränkung}\rangle\ [,\langle\text{KE-Einschränkung}\rangle]^{\infty}\right]\right)\ \sqcup.$$

Es werden alle bzw. keine KE's mit Ausnahme der KE-Einschränkungen aktiviert bzw. passiviert.

z. B.

PASSIV (ALLE(⟨KE-Einschränkung⟩))
AKTIV (KEINE(KE-Einschränkung⟩)) } gleiche Wirkung

PASSIV (KEINE(⟨KE-Einschränkung⟩))
AKTIV(ALLE(⟨KE-Einschränkung⟩)) } gleiche Wirkung

Eintragung einer vordefinierten Reaktion auf ein Kontrollereignis:

Anweisung: KONTROLLE (⟨KE-Bezeichnung⟩ -⟨KE-Reaktion⟩ ⊔.

⟨KE-Bezeichnung⟩ ::= s.o
⟨KE-Reaktion⟩ ::= Anweisung

Bei Erreichen des durch KE-Bezeichnung spezifizierten Kontrollereignisses wird die in KE-Reaktion angegebene Anweisung ausgeführt.

Beispiel:

```
GIB KOMMANDOS␣:␣STARTE,STDHP,AKTIV=ALLE␣.

START STDHP
STDHP *KE= START␣:BL-KE(SUB1);BL-ZEILE(19o)␣.

ENDE PS&BCPLDUMP (6.o1)  o.32
STDHP *KE= START␣:KONTROLLE(SUB1-BL-BRINGE(1))␣.
STDHP *KE= START␣:␣.
STDHP *KE= 19o␣:␣.


  MO  STDHP
|                15o    1 FFFFFF

ENDE PS&BCPLDUMP (6.o1)  o.38
STDHP *KE= 19o␣:␣.


  MO  STDHP
|                15o    1 oooooi

ENDE PS&BCPLDUMP (6.o1)  o.33
STDHP *KE= 19o␣:BEENDE␣.
```

5.5 Beispiele für KE's und Anweisungen

```
GIB KOMMANDOS:MONT.,STDHP'UP1'S&UEBERWACHE.

ENDE MONTIERE (22.05)  2.66

GIB KOMMANDOS:STARTE,STDHP,AKTIV=KEINE(START).

START STDHP
STDHP *KE= START:BL-ZEILE(150).
STDHP *KE= START:BL-ZEILE(190);BL-ZEILE(200);DRUCK.
STDHP *KE= START
STDHP *KE= START:AKTIV(ALLE(200)).
STDHP *KE= START:.
STDHP *KE= 150:BL-BRINGE'GE(IA,IE,I).


  MO  STDHP
IA                95   1 FFFFFF
IE                95   1 000014

ENDE PS&BCPLDUMP (6.01)  0.40
STDHP *KE= 150:.
STDHP *KE= 190:.
STDHP *KE= 190:BL-KE(SUB1);AKTIV(200-10).

ENDE PS&BCPLDUMP (6.01)  0.38
STDHP *KE= 190:.
STDHP *KE= SUB1:RUECKV.

RUECKVERFOLGER:
  2. TAS-MO       S&UEBERWACHE  ADRESSE   A-    2
  1. BCPL-HP      STDHP         ADRESSE   1-   64   ZEILE    190-1

ENDE PS&RUECKVERF (15.00)  0.20
STDHP *KE= SUB1:TKOP.,ENG&BCPL4,PROT.=KO,ZEILE=190.
   000190  V!!:=SUB1(B2,B3)

ENDE TKOPIERE (7.01)  0.08

GIB WEITERES VORRANGKOMMANDO:.

VORRANGKOMMANDOS AUSGEFUEHRT
STDHP *KE= SUB1:BL-ZEILE(160).
STDHP *KE= SUB1:='BL-BRINGE(I).


  MO  STDHP
I                150   1 000001

ENDE PS&BCPLDUMP (6.01)  0.38
STDHP *KE= SUB1:BL-BRINGE(IA,IE,J,&&&V(0:10)).


  MO  STDHP
&&&V          *     90   1 000096
              *     90   1 3FFFFF
              ...      10 MAL
IA                  95   1 FFFFFF
IE                  95   1 000014
I                  150   1 000001

ENDE PS&BCPLDUMP (6.01)  0.47
STDHP *KE= SUB1:.
STDHP *KE= 160:.
STDHP *KE= 190:.
```

I ist in Zeile 150 noch nicht vereinbart, deshalb nicht angelistet.

```
STDHP *KE= SUB1¤:BL-BRINGE(STDHP(I))¤.


  MO  STDHP
I                15o    1 ooooo2

ENDE PS&BCPLDUMP (6.o1)  o.46
STDHP *KE= SUB1¤:¤.
STDHP *KE= 16o¤:¤.
STDHP *KE= 19o¤:¤.
STDHP *KE= SUB1¤:¤.
STDHP *KE= 16o¤:BL-BRINGE(I)¤.


  MO  STDHP
I                15o    1 ooooo4

ENDE PS&BCPLDUMP (6.o1)  o.43
STDHP *KE= 16o¤:BL-KE(STDHP(I))¤.

ENDE PS&BCPLDUMP (6.o1)  o.38
STDHP *KE= 16o¤:¤.
STDHP *KE= I¤:BL-BRINGE(I)¤.


  MO  STDHP
I                15o    1 ooooo4

ENDE PS&BCPLDUMP (6.o1)  o.32
STDHP *KE= I¤:¤.
STDHP *KE= I¤:¤.
STDHP *KE= 19o¤:¤.
STDHP *KE= SUB1¤:¤.
STDHP *KE= I¤:PASSIV(I-1o)¤.
STDHP *KE= I¤:¤.
STDHP *KE= 16o¤:¤.
STDHP *KE= 19o¤:¤.
STDHP *KE= SUB1¤:¤.
STDHP *KE= I¤:¤.
STDHP *KE= I¤:KEPASSIV(I)¤.
STDHP *KE= I¤:¤.
STDHP *KE= 16o¤:¤.
STDHP *KE= 19o¤:¤.
STDHP *KE= SUB1¤:¤.
STDHP *KE= 16o¤:¤.
STDHP *KE= 19o¤:BL-BRINGE(I,SUB1,&&&V(1:1o,¤'),V!o)¤.


  MO  STDHP
SUB1             1oo    1 oooooo9

  MO  STDHP
V                *     9o    1 oooo9E
&&&V             *     9o    1 oooeD8
                 *     9o    1 ooo32C
                 *     9o    1 ooeB9A
                 *     9o    1 oo26AC
                 *     9o    1 oo743C
                 *     9o    1 o13C2C
                 *     9o    1 3FFFFF
                ...          4 MAL
I                15o    1 oooooo7

ENDE PS&BCPLDUMP (6.o1)  o.46
STDHP *KE= 19o¤:BEENDE¤.
```

Beispiel für Marken als dynamische KE's:

- BCPL-Programm

```
      :
M1: X:= Y
      :
      :
M2: A:= V ! I
      :
      :
GIB KOMMANDOS␣ : ␣ STARTE, STDHP, AKTIV = ALLE␣.

START STDHP
STDHP * KE = START␣: BL-WEITER (STDHP(PROZ1(! M1))) ␣.
STDHP * KE = M1␣: ␣.
STDHP * KE = M1␣: BL-KE (! M2) ␣.
STDHP * KE = M1␣: ␣.
STDHP * KE = M2␣: BEENDE␣.
```


5.6 Durchreichung von S&KEP-Leistungen an höhere Programmiersprachen

Um dem Benutzer, der sich einer höheren Programmiersprache bedient, Leistungen der Kontrollereignisprozeduren innerhalb des Programms verfügbar zu machen, wurden Unterprogramme geschaffen, die als Bindeglieder zwischen Benutzerprogrammen und der Programmiersystemkomponente S&KEP dienen. Diese Leistungen werden nur zugänglich, wenn das Montageobjekt S&KEP anmontiert wurde und alle Kontrollereignisse aktiviert sind AKTIV = ALLE (STARTE-Kommando). Die Montage von S&KEP kann zur Übersetzungszeit durch die Angabe VARIANTE = GS oder KV oder im MONTIERE-Kommando durch eine Angabe MO = ... 'S&KEP erreicht werden.

Die Leistungen die erbracht werden können sind:

1. Kontrollereignis melden
2. Kontrollanweisung ausführen
3. Unterprogrammaufruf als Kontrollanweisung vereinbaren.
   Dabei muß ein Kontrollanweisungsname definiert werden.
   Er ist grundsätzlich an folgende Syntax gebunden:

   ⟨Anweisungsname⟩ ::= ⟨Buchstabe⟩[⟨Buchstabe⟩]^{11}

Deklaration:

EXTERNAL BL. PS: { BL.KE / BL.KA / BL.KP }

Typbeschreibung der einzelnen Parameter:

⟨string⟩ ::= BCPL-String

⟨proc⟩ ::= NONREC-Prozedurvariable beliebiger Klasse, die eine parameterlose Prozedur beschreibt.

Aufrufe der Kontrollereignisprozedur:

BL.KE (⟨string⟩)	Meldet Kontrollereignis "string" am Terminal
BL.KA (⟨string⟩)	Die Kontrollanweisung "string" wird ausgeführt
BL.KP (⟨string⟩,⟨proc⟩)	"string" wird als Kontrollanweisung vereinbart. Die Angabe dieser privaten Anweisung bei beliebigen Kontrollereignissen führt dann zum Aufruf der Prozedur "proc".

OBJEKTROUTINEN

OBJEKTROUTINEN

1. Die Routine BL. LISTE

1.1. Allgemeine Beschreibung

Dynamische und statische Vektoren besitzen die Eigenschaft, daß ihre Länge bereits zur Compilezeit festgelegt und während des Objektlaufs nicht veränderbar ist.
Durch die Routine BL. LISTE ist es möglich, Speicherbereiche (= Pseudovektoren) zu definieren, deren Länge während des Objektlaufes erstmalig definiert wird und dynamisch verändert werden kann.

Im Deklarationsteil muß diese Routine wie folgt vereinbart sein:

```
EXTERNAL BCPL. CLIST: BL. LISTE
NONREC 15: BL. LISTE
```

Die verschiedenen Aufrufarten von BL. LISTE dienen einerseits der Definition und Längenveränderung der Pseudovektoren, andererseits der Löschung einzelner oder aller derartiger Speicherbereiche.

1.2. Aufrufarten von BL. LISTE

1.2.1. Erstaufruf (Definitionsaufruf)

Syntax:

BL. LISTE (NR, LNG, LVL)

Bezeichnung:

- NR : Nummer des Pseudovektors, der durch diesen Aufruf definiert wird ($1 \leq NR \leq 100$)
- LNG : geforderte Länge des Pseudovektors in BCPL-Elementen
- LVL : Adresse der Pseudo-Vektorvariablen, in der die Anfangsadresse des Pseudovektors gehalten wird.

Wirkung:

Beim Erstaufruf für einen Pseudovektor mit der Nummer NR wird ein Speicherbereich reserviert, der aus LNG BCPL-Elementen besteht. Seine Anfangsadresse wird im BCPL-Element mit der Adresse LVL abgelegt.

1.2.2. Verändern der Pseudovektorlänge

Syntax:

BL. LISTE (NR, LNG)

Bezeichnung:

Parameterbeschreibung siehe L 1.1.2

Wirkung:

Der Pseudovektor NR erhält eine neue Länge von LNG BCPL-Elementen. Dies kann eine Verlängerung oder Verkürzung des ursprünglichen Vektors bedeuten.

1.2.3. Löschen eines Pseudovektors

Syntax:

BL. LISTE (NR, 0)

Bezeichnung:

Parameterbeschreibung siehe L 1.1.2

Wirkung:

Der Pseudovektor mit der Nummer NR wird gelöscht. Der nächste BL. LISTE Aufruf für einen Speicherbereich mit dieser Nummer muß ein Erstaufruf sein.

1.2.4. Löschen aller Pseudovektoren

Syntax:

BL. LISTE (0)

Bezeichnung:

Parameterbeschreibung siehe L 1.1.2

Wirkung:

Alle Pseudovektoren werden gelöscht. Ein weiterer Aufruf mit der Routine BL. LISTE muß ein Erstaufruf sein.

1.3. Ablage der Pseudovektoren

Die durch BL. LISTE vereinbarten Pseudovektoren liegen im Speicher dicht gepackt und in Abhängigkeit ihrer Nummer in aufsteigender Reihenfolge hintereinander. Jede Veränderung innerhalb dieses Gesamtbereiches, welche z. B. durch

- Einschieben (Erstdefinition)
- Verlängern
- Verkürzen
- Löschen

von Pseudovektoren verursacht werden kann, hat in den meisten Fällen auch eine Verschiebung anderer Pseudovektoren zur Folge (Einhaltung der NR-Sortierfolge!). Die Anfangsadressen der betroffenen Pseudovektoren werden in diesem Fall ohne Einfluß des Benutzers in den zugehörigen Vektorvariablen LVL korrigiert. Die rückgemeldeten Adressen sind stets gerade, d.h. ein Pseudovektor wird bezüglich seiner Länge in BCPL-Elementen stets auf Ganzwortgrenze aufgerundet.

Regeln:

a) Es dürfen maximal 100 Pseudovektoren definiert werden.

b) Die Nummern der Pseudovektoren müssen der Bedingung genügen:

$$1 \leq NR \leq 100$$

1.4. Zugriff auf die Elemente eines Pseudovektors

Mit dem Vektoroperator "!" kann auf die einzelnen Elemente zugegriffen werden (siehe auch C 5.1).

Achtung:

Auf die einzelnen Vektorelemente darf nur über Relativadressen zugegriffen werden.

Implementierung TR 440:

a) Die Gesamtlänge aller Pseudovektoren muß ≤ 20 K Ganzworte (= 40 K BCPL-Elemente) sein.

b) Ein Pseudovektor beginnt immer auf Ganzwortgrenze.

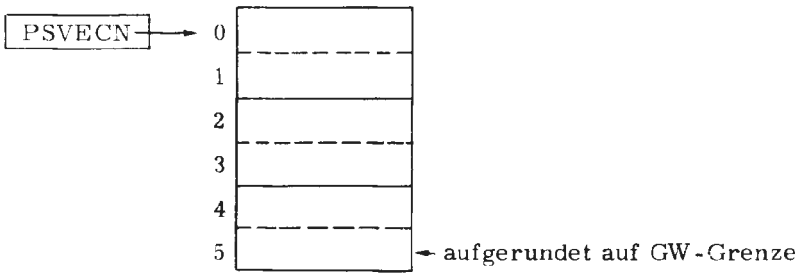
c) Die für einen Pseudovektor geforderte Länge in BCPL-Elementen (= LNG) wird immer auf Ganzwortgrenze aufgerundet.

d) Alle Pseudovektoren und nachgeforderten Speicherbereiche haben die Vorbesetzung

$H 3FFFFF

1.5. Beispiele zu BL. LISTE

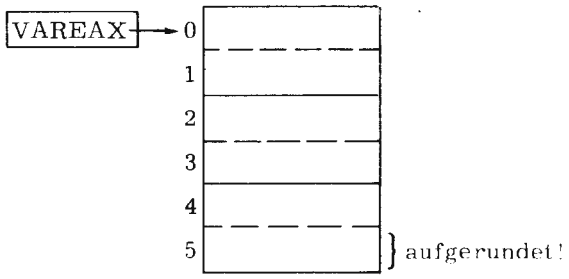
a)
```
LET LNG = NIL
:
READ (5, "I3", 1, LV LNG)      // LNG sei 5
BL. LISTE (10, LNG, LV PSVECN)
```



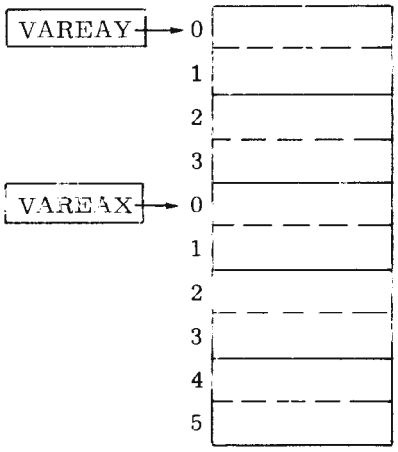
b)
```
EXTERNAL LNG1, LNG2, LNG3    // LNG1 sei 5
.                            // LNG2 sei 4
:                            // LNG3 sei 7
BL. LISTE (7, LNG1, LV VAREAX)
BL. LISTE (5, LNG2, LV VAREAY)
:
BL. LISTE (6, LNG3, LV VAREAZ)
```

Pseudovektorablage

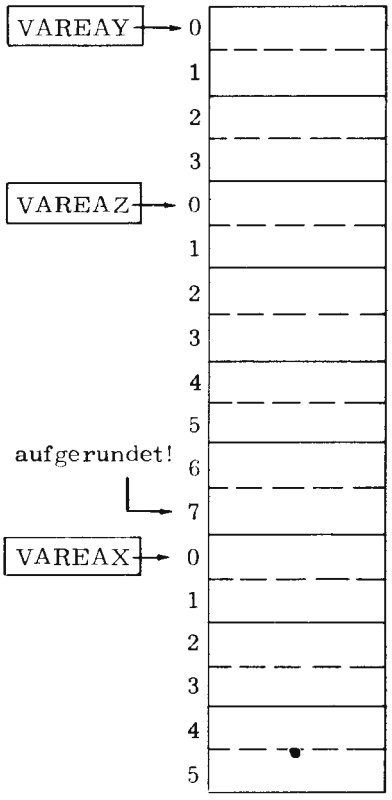
Nach 1. Aufruf BL. LISTE:



Nach 2. Aufruf BL. LISTE:



Nach 3. Aufruf BL. LISTE:



c) Die Länge eines Pseudovektors wird eingelesen, der zugehörige Pseudovektor erzeugt und mit Eingabedaten belegt. Jedes Element wird mit einem Suchwort verglichen und bei Übereinstimmung in einen zweiten Pseudovektor eingetragen. Der zweite Pseudovektor wird bei wenigstens einer gefundenen Übereinstimmung auf seine echt benötigte Länge verkürzt, der erste Pseudovektor gelöscht.

```
GLOBAL  &( ST:1  &)
EXTERNAL  BL.EA:READ,WRITE
NONREC  1:READ,WRITE
EXTERNAL  RCPL.CLIST:BL.LISTE
NONREC  1:BL.LISTE
&T:  &'
LET I,J,LAENGE,MASK1=-1,0,NIL,NIL
AND ERSTLI,ZWEITLI=NIL,NIL
READ(8,[12[,1,LV LAENGE)
BL.LISTE(1,LAENGE+1,LV ERSTLI)
BL.LISTE(2,LAENGE+1,LV ZWEITLI)
&(LIES  I:=I+1
READ(8,[A3[,1,LV ERSTLI!I)
      &)LIES  REPEATUNTIL  I=LAENGE
READ(8,[-3[,1,LV MASK1)
FOR I=0 TO LAENGE
&(FOR   IF ERSTLI!I=MASK1
         &(TRUE   ZWEITLI!J:=I
                  J:=J+1  &)TRUE
                  &)FOR
IF  J=0  &(NO   L.LISTE(0);WRITE(9,['KEINE UEBEREINSTI
MMUNG'/[,0)
                  &)NO
          &(YES   BL.LISTE(2,J);     BL.LISTE(1,0)
                    FOR I=0 TO J-1
             WRITE(9,[A3,2X,'GEFUNDEN IN LISTENELEMENT
 MIT RELATIVADRESSE ',N/[,2,MASK1,ZWEITLI!I)
                  &)YES
    FINISH  &)
```

Zugehörige Konsoleingabe und Konsolausgabe

1)

```
                  :04:AAABBBAAACCCAAAM.:AAAM.
AAA  GEFUNDEN IN LISTENELEMENT MIT RELATIVADRESSE 0
AAA  GEFUNDEN IN LISTENELEMENT MIT RELATIVADRESSE 2
AAA  GEFUNDEN IN LISTENELEMENT MIT RELATIVADRESSE 4
```

2)

```
                  :03M.M:1112223334444M.:555M.
KEINE UEBEREINSTIMMUNG
```

2. Die Routine UNPACKSTR

Deklaration:

```
EXTERNAL  BCPL.UTIL: UNPACKSTR
    NONREC 9:        UNPACKSTR
```

Syntax:

UNPACKSTR (V,S)

Bezeichnung:

S : BCPL-String

V : Vektor

Wirkung:

Der BCPL-String S wird gestreckt indem die Stringzeichen oktadenweise rechtsbündig in den Vektorelementen des Vektors V abgelegt werden. Die rechtsbündigen Oktaden werden mit führenden Ignores (Nulloktaden) abgelegt. Das erste Vektorelement enthält die Stringlänge.

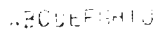
Regel:

Der BCPL-String darf nicht mehr als 255 Zeichen enthalten.

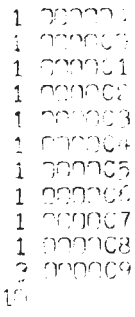
Beispiel:

```
LET STRADR=VEC 3
AND STRPU=NIL
AND ZEICHANZ=NIL
MANIFEST  &(  BPCH=8    // BITS PER CHARACTER
              BPW=24    // BITS PER WORD
                      &)
MANIFEST  &( LNG=SLCT BPCH:(BPW-BPCH)  &)
READ(8,[S10[,1,STRADR)
ZEICHANZ:=LNG OF STRADR
BL..ISTE(1,ZEICHANZ,LV STRPU)
UNPACKSTR(STRPU,STRADR)
FOR I=0 TO ZEICHANZ
WRITE(9,[Z/[,1,LV STRPU!I)
WRITE(9,[N/[,1,ZEICHANZ)
```

Konsoleingabe:



Konsolausgabe:



3. Die Routine PACKSTR

Deklaration:

EXTERNAL BCPL.UTIL: PACKSTR
NONREC 9 : PACKSTR

Syntax:

PACKSTR (S, V)

Bezeichnung:

V : Vektor

S : BCPL-String

Wirkung:

V muß einen gestreckten String enthalten. Dabei enthält das Element V ! 0 die Länge L des Strings, V ! 1, ... bis V ! L enthalten je eine Zentralcode-Oktade rechtsbündig.
Die beiden linken Oktaden aller Vektorelemente sind Ignores (Nulloktaden). Die jeweils rechten Oktaden von V ! 0 ... bis V ! L werden dicht gepackt ab S ! 0, S ! 1,.. abgelegt (3 Oktaden je BCPL-Element), so daß S ein String im BCPL-Format ist. Das Resultat von PACKSTR ist die Stringlänge in BCPL-Elementen.

Regel:

a) S muß ein Vektor von Mindestlänge L/CHPW + 1 sein (CHPW = Anzahl der Zeichen pro BCPL-Element = 3 beim TR 440).

b) Die Längenangabe L im Vektor V darf 255 nicht überschreiten.

Beispiel:

```
LET ENTVEC=VEC 10
AND BCPLST=VEC 3
AND LNG=NIL
FOR I=1 TO 10
READ(8,[A1[,1,LV ENTVEC!I)
ENTVEC!0:=10
LNG:=PACKSTR(BCPLST,ENTVEC)
FOR I=0 TO 3
WRITE(9,[Z/[,1,LV BCPLST!I)
WRITE(9,[S10/[,1,BCPLST)
WRITE(9,[N/[,1,LNG)
```

Konsoleingabe:

```
?123456789
```

Konsolausgabe:

```
 1 B4B0B1
 1 B2B3B4
 1 B5B6B7
 1 B8B9C0
?123456789
4
```

Ab ENTVEC ! 1 werden auf einen Vektor Zeichen im A1-Format eingelesen.
ENTVEC ! 0 wird mit der Anzahl Zeichen belegt.
Die führenden Ignores in den eingelesenen Vektorelementen (für PACKSTR) werden durch das A1-Format bereits eingesetzt.

4. Die Routinen NLEVEL und NLONGJUMP

4.1. Allgemeine Einführung

Es ist in BCPL verboten, mit einem GOTO-Statement eine Prozedur zu verlassen, um z. B. in einer Unterprogrammverschachtelung aus einer Prozedur auf eine Marke in einer anderen Prozedur oder aus einer Prozedur auf eine Marke des Hauptprogramms zu springen. Um dennoch derartige Sprünge zu ermöglichen, wurden die beiden Routinen NLEVEL und NLONGJUMP geschaffen.
Sie müssen im BCPL-Programm wie folgt deklariert werden.

```
EXTERNAL BCPL.LEVEL: NLONGJUMP, NLEVEL
NONREC 9: NLONGJUMP, NLEVEL
```

4.2. NLEVEL

Syntax:

NLEVEL (V)

Bezeichnung:

V : Vektor

Wirkung:

Existieren innerhalb einer Prozedur oder dem Hauptprogramm Marken, auf die auch aus anderen untergeordneten Prozeduren gesprungen werden soll, muß in diesem, die Marken enthaltenden Programmteil (Prozedur oder Hauptprogramm) ein Vektor V der Länge 3 deklariert werden.
Durch einen Aufruf der Prozedur NLEVEL (V) in dem die Marken enthaltenden Programmteil wird nun der Vektor V mit Verwaltungsinformation gefüllt. Diese erlaubt in den folgenden untergeordneten Prozeduren einen Rücksprung mit NLONGJUMP.

4.3. NLONGJUMP

Syntax:

NLONGJUMP (V,M)

Bezeichnung:

V : Vektor

M : Sprungmarke

Wirkung:

Der Aufruf der Routine NLONGJUMP bewirkt, daß auf die Marke M gesprungen wird.

Regel:

In einer Aufrufverschachtelung darf mit NLONGJUMP nur in eine hierarchisch übergeordnete (noch nicht beendete) Prozedur gesprungen werden, da bei einem Sprung in eine hierarchisch untergeordnete Prozedur deren Verwaltungsinformation zu diesem Zeitpunkt noch unbekannt ist.
In dieser Prozedur muß der Vektor V durch einen Aufruf von NLEVEL (V) initialisiert worden sein.

Beispiele:

a)

```
GLOBAL &( ST:1 &)
EXTERNAL BL.EA:WRITE
NONREC 1:WRITE
EXTERNAL BCPL.LEVEL:NLEVEL
NONREC 9:NLEVEL
ST:  &(
LET P() BE
&(P  EXTERNAL VERWALT,MARKE
   STATIC &( VERWALT=TABLE 0 REP 3  &)
EXTERNAL UP:PROZ
NLEVEL(VERWALT)
PROZ()
WRITE(9,['KEIN RUECHSPRUNG AUF DIESE ZEILE'/[,0)
MARKE:  WRITE(9,['RUECKSPRUNG AUF DIESE ZEILE'/[,0)
RETURN  &)P
P()
FINISH  &)
```

} Haupt-Programm

```
EXTERNAL BCPL.LEVEL:NLONGJUMP
NONREC 9:NLONGJUMP
EXTERNAL  :PROZ
&(  LET PROZ() BE
&(PROZ  EXTERNAL VERWALT,MARKE
    NLONGJUMP(VERWALT,MARKE)
RETURN  &)PROZ
&)
```

} Montageobjekt UP

Konsolausgabe:

```
RUECKSPRUNG AUF DIESE ZEILE
```

b)

```
GLOBAL  &( BEG:1  &)
EXTERNAL BL.EA :WRITE
NONREC 1:WRITE
EXTERNAL BCPL.LEVEL:NLEVEL,NLONGJUMP
NONREC 9:NLEVEL,NLONGJUMP
EXTERNAL WEITE,STORE,UP2
STATIC  &(  STORE=TABLE 0   REP 3  &)
NLEVEL(STORE)
BEG:  &(
LET UP1() BE
&(UP1
NONREC 15:UP2
WRITE(9,['IN UP1 GESPRUNGEN'/[,0)
UP2()
WRITE(9,['AUF DIESE ZEILE KEIN RUECKSPRUNG'/[,0)
RETURN    &)UP1
LET UP2()  BE
&(UP2  EXTERNAL WEITE,STORE
WRITE(9,['IN UP2 GESPRUNGEN'/[,0)
NLONGJUMP(STORE,WEITE)
WRITE(9,['DIESE ZEILE WIRD NICHT DURCHLAUFEN'/[,0)
RETURN  &)UP2
UP1()
WRITE(9,['DIESE ZEILE WIRD NICHT DURCHLAUFEN'/[,0)
WEITE:  WRITE(9,['SPRUNG AUS UP2 AUF DIESE ZEILE'/[,0)
FINISH  &)
```

Konsolausgabe:

```
IN UP1 GESPRUNGEN
IN UP2 GESPRUNGEN
SPRUNG AUS UP2 AUF DIESE ZEILE
```

INTRINSICS

INTRINSICS

1. Allgemeine Beschreibung

Die BCPL-Intrinsics wurden geschaffen, um einerseits den Anforderungen einer assemblernahen Systemprogrammierung gerecht zu werden und andererseits den Programmierkomfort beim Anschluß von Codeprozeduren zu unterstützen.

Die Intrinsics müssen wie folgt deklariert werden:

EXTERNAL BL. INTR : ⟨name⟩, ⟨name⟩, . . .
NONREC 2: ⟨name⟩, ⟨name⟩, . . .
⟨name⟩ ist der Name eines Intrinsics

Beachte:

Aus oben genannten Gründen wird bei der Erklärung der INTRINSICS weitgehenst Kenntnis der Assemblersprache TAS 440 vorausgesetzt.

2. MOVE

Syntax:

MOVE (LNG, ADRQUELL, ADRZIEL)

Bezeichnung:

LNG : Feldlänge in BCPL-Elementen

ADRQUELL: Anfangsadresse der zu transportierenden Information

ADRZIEL : Anfangsadresse des Zielgebietes

Wirkung:

MOVE bewirkt einen Transport von LNG BCPL-Elementen. Die Anfangsadresse der zu transportierenden Information wird durch den Parameter ADRQUELL angegeben, die Anfangsadresse des Zielgebietes durch ADRZIEL. Es ist dabei gleichgültig, welcher Art die BCPL-Elemente sind (z. B. Variable, Vektoren, Strings). Setzt sich der Informationsblock aus verschiedenartigen BCPL-Elementen zusammen, ist für deren passende Ablage vor dem Transport zu sorgen. Ebenso muß dafür gesorgt werden, daß die entsprechende Anzahl von BCPL-Elementen im Zielgebiet zur Verfügung steht (evtl. Überschreiben von Information!).

Regeln:

a) Die Anfangsadresse des Quell- und Zielgebietes sowie die Länge müssen gerade sein.

b) Tables, Strings und Vektoren werden vom Compiler stets auf gerader Adresse beginnend angelegt.

Beispiel:

Siehe 3.

3. SAVEREG

Syntax:

SAVEREG ( )

Wirkung:

Das Intrinsic SAVEREG ( ) wird als Funktion aufgerufen. Der Aufruf wird auf den TAS-Befehl QCR (siehe auch TR 440 Befehlslexikon) abgebildet. Als Funktionswert wird die Anfangsadresse des QCR-Blocks zurückgemeldet, der die Register in folgender Reihenfolge enthält.

Sei n der zurückgelieferte Funktionswert (die Anfangsadresse des QCR-Blocks).

Es gilt:

	$\langle n \rangle_{\text{Typenkennung}}$	:= 2+ ⟨M⟩	// M = Markenregister
n ! 0 →	$\langle n \rangle_{1-24}$	:= ⟨B⟩	// B = Bereitadressenregister
n ! 1 →	$\langle n \rangle_{25-32}$	:= ⟨K⟩	// K = Merklichterregister
	$\langle n \rangle_{33-40}$	:= ⟨Y⟩	// Y = Schiftzähler
	$\langle n \rangle_{41-48}$	:= ⟨U⟩	// U = Unterprogrammregister
n ! 2, n ! 3 → ⟨n+2⟩		:= $TK_{\text{von A;}}$ ⟨A⟩	// A = Akkumulator
n ! 4, n ! 5 → ⟨n+4⟩		:= $TK_{\text{von Q;}}$ ⟨Q⟩	// Q = Quotientenregister
n ! 6, n ! 7 → ⟨n+6⟩		:= $TK_{\text{von D;}}$ ⟨D⟩	// D = Multiplikandenregister
n ! 8, n ! 9 → ⟨n+8⟩		:= $TK_{\text{von H;}}$ ⟨H⟩	// H = Hilfsregister
n ! 10 → ⟨n+10⟩		:= 2; ⟨T⟩ , 0	// T = Prüfregister

Regeln:

a) Bei Aufruf von SAVEREG wird das U-Register verändert.

b) Erfolgt der Aufruf von SAVEREG aus einer rekursiven Funktion, wird das B-Register zerstört.

Beispiel:

```
LET QCRBLOCK=VEC 11
AND R.BB,R.M,R.SH,R.UP,R.AL,R.AR,R.QL,R.QR,R.DL,R.DR=0
 REP 10
MANIFEST  &(   MASK1=SLCT 8:16:1
               MASK2=SLCT 8:8:1
               MASK3=SLCT 8:0:1  &)
MOVE(12,SAVEREG(),LV QCRBLOCK!0)
R.BB:=QCRBLOCK!0
R.M:=MASK1 OF QCRBLOCK
R.SH:=MASK2 OF QCRBLOCK
R.UP:=MASK3 OF QCRBLOCK
R.AL:=QCRBLOCK!2
R.AR:=QCRBLOCK!3
R.QL:=QCRBLOCK!4
R.QR:=QCRBLOCK!5
R.DL:=QCRBLOCK!6
R.DR:=QCRBLOCK!7
```

4. EXECUTE

Syntax:

EXECUTE (ADR)

Bezeichnung:

ADR : Adresse eines BCPL-Elements

Wirkung:

ADR ist die Adresse einer Speicherzelle, die einen Befehl enthalten muß. Dieser Befehl wird ausgeführt (wird abgebildet auf T-Befehl in TAS).

Beispiel:

```
EXTERNAL  BL.INTR : EXECUTE
          NONREC  2 : EXECUTE
GLOBAL $( BEFWORT: 100   $)      // wird in Codeprozedur belegt
EXECUTE (LV  BEFWORT)
```

5. SETSW

Syntax:

SETSW (BOOL, SWNUMB)

Bezeichnung:

BOOL : logischer Ausdruck

SWNUMB: Wahlschalter

Wirkung:

Dem Programmierer stehen 8 Wahlschalter zur Verfügung, die er setzen oder löschen kann. Mit SETSW wird der durch SWNUMB bezeichnete Wahlschalter auf den durch BOOL angegebenen Wert (TRUE oder FALSE) gesetzt.

Die Wahlschalter sind abwicklerrelativ und können zum besseren Austausch von Information zwischen Operatoren benutzt werden.

(Abbildung auf SSR 1 8 bzw. SSR 1 12).

Regel:

SWNUMB muß sein:

$$1 \leq \text{SWNUMB} \leq 8$$

Beispiel:

```
SETSW (F(X), WAHL ! 3)
```

6. SW

Syntax:

SW (SWNUMB)

Bezeichnung:

SWNUMB : Wahlschalter

Wirkung:

SW wird als Funktion aufgerufen. Der Funktionswert des Aufrufes ist TRUE, wenn der Wahlschalter mit der Nummer SWNUMB gesetzt ist. Ist er gelöscht, wird als Funktionswert FALSE zurückgeliefert.

Regel:

SWNUMB muß sein:

$$1 \leq \text{SWNUMB} \leq 8$$

Beispiel:

```
GO TO SW(WAHL!F( )) → M1, M2
```

7. SETSL

Syntax:

SETSL (BOOL, LIGHTNUMB)

Bezeichnung:

BOOL : logischer Ausdruck

LIGHTNUMB : Merklicht

Wirkung:

Dem Programmierer stehen 8 Merklichter zur Verfügung, die er setzen und löschen kann.

Mit SETSL wird das durch LIGHTNUMB bezeichnete Merklicht auf den durch BOOL ausgegebenen Wert (TRUE oder FALSE) gesetzt. Die Merklichter können in Codeprozeduren durch eine Reihe von TAS-Befehlen angesprochen werden.

Regel:

LIGHTNUMB muß sein:

$$1 \leq \text{LIGHTNUMB} \leq 8$$

Beispiel:

```
EXTERNAL MERK1, TIME
SETSL (TIME( ) > MAX → TRUE, FALSE, MERK1)
```

8. SL

Syntax:

SL (LIGHTNUMB)

Bezeichnung:

LIGHTNUMB : Merklicht

Wirkung:

SL wird als Funktion aufgerufen. Der Funktionswert des Aufrufes ist TRUE, wenn das Merklicht mit der Nummer LIGHTNUMB gesetzt ist. Ist es gelöscht, wird als Funktionswert FALSE zurückgeliefert.

Regel:

LIGHTNUMB muß sein:

$$1 \leq \text{LIGHTNUMB} \leq 8$$

TR 440:

Die Merklichter können in Codeprozeduren durch eine Reihe von TAS-Befehlen angesprochen werden.

Beispiel:

```
EXTERNAL MERKF
MERKF (SL (M ! I), I)
```

9. SSR

Syntax:

SSR (PL, PR, ADR)

Bezeichnung:

PL : Linksadreßteil eines SSR-Befehls

PR : Rechtsadreßteil eines SSR-Befehls

ADR : Adresse des Versorgungsblocks

Wirkung (TR 440):

Mit Aufruf des Intrinsics SSR ist es möglich, auch auf BCPL-Ebene, SSR-Befehle zu geben (siehe auch BS3-Systemdienste).
Die Parameter PL und PR sind Links- und Rechtsadreßteil des SSR-Befehls, ADR ist die Adresse des Versorgungsblocks.

Der Aufruf SSR (. . .) wird abgebildet auf die TAS-Befehlsfolge

```
TCB   ADR,
SSR   PL    PR,
```

Regel:

Die Fehleradresse muß in derselben Prozedur liegen, in der die SSR-Routine gerufen wurde.

Wird auf der Fehleradresse das Intrinsic SAVEREG gerufen, und werden die Registerinhalte zur Fehlerprüfung verwendet, so kann bei rekursiven Prozeduren das B-Register zerstört sein.

Dumpen eines Speicherbereichs:

```
MANIFEST  $( TEIL = $H0; ALLES = $H3 $)
.
.
.
LET DUMPE ( AA, LNG, MODUS)
$(        //  AA, LNG  gerade
    LET  VB   = VEC 3
    AND  QCR = NIL
    VB ! 0,   VB : 1  :=  FAUS, MODUS
    VB ! 2,   VB : 3  :=  AA,   AA + LNG
    SSR  ( 6, 0, VB)
    .
    .
    .
    FAUS  :  QCR = SAVERE G (   )
                 .
                 .     //  Fehleranalyse
                 .
                                          $)
```

10. SETTK

Syntax:

SETTK (ADR, TK)

Bezeichnung:

ADR : Adresse eines BCPL-Elements

TK : Typenkennung

Wirkung (TR 440):

Das durch ADR bezeichnete BCPL-Element erhält die Typenkennung TK.

Beispiel:

```
EXTERNAL  BL. INTR :  SETTK
NONREC 2           :  SETTK
BEFEHL.SI          := $HAC0000
SETTK ( LV  BEFEHL.SI,   2)
```

11. TK

Syntax:

TK (ADR, TK)

Bezeichnung:

ADR : Adresse eines BCPL-Elements

TK : Typenkennung

Wirkung:

TK wird als Funktion aufgerufen.
Der Funktionswert des Aufrufes ist TRUE, wenn das durch ADR bezeichnete BCPL-Element die Typenkennung TK besitzt.
Ist die Typenkennung ungleich TK, wird als Funktionswert FALSE zurückgeliefert.

Beispiel:

```
EXTERNAL  BL.INTR  :  TK
EXTERNAL  TASMO, BCPL MO
NONREC 2           :  TK
NONREC 11          :  TASMO
VARIAB             := TASMO (    )

BCPLMO  ( TK (LV VARIAB, 0) →
              RESULTIS  VARIAB,
              VALOF  $ ( SETTK (LV VARIAB, 0),
                         RESULTIS VARIAB $)
                                          )
```

Eine BCPL-Funktion BCPLMO wird aufgerufen. Die Typenkennung des zu übergebenden Parameters wird auf 0 abgeprüft. Ist sie nicht 0, wird die Typenkennung auf 0 gesetzt.

12. US

Syntax:

US (ADRQUELL, ADRUSTAB, ADRZIEL)

Bezeichnung:

ADRQUELL : Adresse eines Ganzwortes

ADRUSTAB : Adresse einer Umschlüsseltabelle

ADRZIEL : Adresse eines Ganzwortes

Wirkung:

Mit dem Aufruf US wird das durch ADRQUELL bezeichnete Ganzwort oktadenweise umgeschlüsselt und in dem durch ADRZIEL bezeichneten Ganzwort abgelegt.

Umgeschlüsselt wird nach einer Umschlüsseltabelle, deren Anfangsadresse mit ADRUSTAB anzugeben ist.

Die Umschlüsseltabelle muß wie folgt aufgebaut sein:

Im ersten Viertelwort steht rechtsbündig das Zeichen (im Zentralcode), in das das Zeichen mit dem Binärwert 0 umgeschlüsselt werden soll, im 2. Viertelwort steht rechtsbündig jenes Zeichen, in das das Zeichen mit dem Binärwert 1 umgeschlüsselt werden soll usw.

Allgemein:

Im n-ten Viertelwort muß rechtsbündig das Zeichen stehen, in das das Zeichen mit dem Binärwert n-1 umgeschlüsselt werden soll (siehe auch TR 440 Befehlslexikon).

Beispiel:

```
EXTERNAL  BL. INTR:  US
NONREC 2:  US
MANIFEST $(   ZEICHANZ = 46   $)  // Anzahl umzuschlüsselnder
                                     Zeichen
LET  I = -1  AND
WANDEL  =  TABLE  $H0C00C1,...,$H093078  // Definition der Um-
                                            schlüsseltabelle
ALTLIST  =  VEC  100  AND  // 300 Zeichen sind zu wandeln
NEULIST  =  VEC  100
FOR  I     =  0  TO  100
READ  (5,  "A3",  1,    LV  ALTLIST ! I)
FOR  I     =  0  TO  100  BY  2
US  (  LV  ALTLIST ! I,  WANDEL,  LV  NEULIST ! I  )
```

300 Zeichen (100 Vektorelemente enthalten je 3 Zeichen) sollen gemäß einer vorgegebenen Umschlüsseltabelle gewandelt werden.

STRINGHANDLING ROUTINEN

STRINGHANDLING ROUTINEN

1. Einführung

In den zuvor beschriebenen Kapiteln wurden Strings stets im Sinne von BCPL-Strings angesprochen (siehe G 4.3).
Zusätzlich existiert ein BCPL-Prozedurensatz, der mit Strings arbeitet, deren interne Ablage (Stringhandling-Format) sich von der der BCPL-Strings unterscheidet. Dieser Prozedurensatz ermöglicht umfassende Stringmanipulationen, die sich auf das Erzeugen, Verändern, Analysieren, Vergleichen und Wandeln von Stringhandling Strings erstrecken (nahezu äquivalenter Prozedurensatz wird in FORTRAN geboten).

1.1. Allgemeine Stringhandling Konventionen

Der Gebrauch des Stringhandling Prozedurensatz setzt die Beachtung und Erfüllung folgender Anforderungen voraus:

a) Jede, in einem Programm aufgerufene Stringhandling Routine (oder Function), muß über eine EXTERNAL-Vereinbarung (siehe E 5.) dem Benutzerprogramm zugänglich gemacht werden.

Beispiel:

```
EXTERNAL BSTRL, BKET, BERS
```

b) Die Prozeduren dürfen nicht rekursiv verwandt werden, sonst erfolgt Fehlerabbruch.

c) Werden in einem Prozeduraufruf Parameter auch zum Rückmelden bestimmter Stringzustandsdaten benutzt - wird also der Wert der auf Parameterposition stehenden Variablen während der Prozedurausführung verändert - müssen diese Parameter als LV-Parameter übergeben werden.

d) Der letzte Parameter einer Prozedur mit variabler Parameterzahl (BNULLST, BKET, BTKET, BKCASSE) ist stets mit FALSE anzugeben.

e) Die von einer Prozedur gegebenenfalls benötigten Speicher für Listen sind auf der Aufrufzeile bereitzustellen und auf Parameterposition zu übergeben (BLISTE, BLBISZ, BLSOLZ, BIPOSZ, BIPOSZN).

1.2. Bedeutung und aktuelle Versorgung der formalen Parameter

Bei den einzelnen Prozedurbeschreibungen werden die zugehörigen Prozedurparameter hinsichtlich ihrer Bedeutung erläutert.
Folgende Tabelle gibt einen Überblick über sämtliche Formalparameter in Bezug auf ihre Bedeutung und den Typ des entsprechenden aktuellen Parameters.

Bei formalen Parametern vom Typ RV wird während der Prozedurausführung stets mit den R-Werten der zugehörigen aktuellen Parametern gearbeitet.

Bei formalen Parametern vom Typ LV wird auf aktueller Parameterposition stets die Übergabe einer Adresse erwartet.

Name	Bedeutung	Deutung des aktuellen Parameters
ZS	Zielstring	RV
B	BCPL-String	RV
R	Replikator	RV
A	entweder Stringadresse	RV
	oder Variable (abhängig vom letzten Parameter)	LV
W	Feldwerte	RV
N	Anzahl	RV
S	String	RV
P{Z, S}	Zeichenposition im angegebenen (Ziel-) string, kann in Prozedur geändert werden	LV
T	Teilstring, definiert durch drei Parameter S, P, N	-
I	Intervall der Buchstaben- oder Ziffernfolge oder Sonderzeichen, als BCPL-String	RV
K	Zeichenklasse, als positive Zahl	RV
NA	Name einer Liste	RV

Name	Bedeutung	Deutung des aktuellen Parameters
Z	Zeiger auf ein Element einer Liste	LV
PEND	Endeparameter bei Prozeduren mit variabler Parameterzahl	RV

2. Begriffsbestimmungen

2.1. String

Ein String ist eine lückenlose Folge von Zentralcodeoktaden außer der Nul-Oktade, die immer auf Ganzwortgrenze beginnt und dessen Länge intern verwaltet wird.
Um einen String zu adressieren, genügt deshalb die Angabe einer Stringvariablen, die dessen Adresse enthält. Beim Anlegen von Strings muß der Benutzer dafür sorgen, daß eine Stringvariable vor ihrem erstmaligen Gebrauch entweder dynamisch (über Vektordefinition, siehe E 1.2) oder statisch (über TABLE, siehe G 5.6) als solche definiert wird. Der angebotene Speicherplatz muß auf gerader Adresse beginnen. Jeder String besitzt eine Länge welche die Anzahl der im String enthaltenen Zeichen beschreibt. Jedes Zeichen eines Strings besitzt eine Positionsnummer, die bei 1 beginnt und in Schritten von 1 steigt. Für je 6 Zeichen sind 2 BCPL-Elemente anzulegen.

2.1.1. Definition eines Strings

Die Variable S enthalte eine gerade Adresse, dann kann mit S im Stringhandling Format gearbeitet werden, wenn eine der folgenden Bedingungen erfüllt ist:

a) S kommt vorher in einer der Prozeduren BSTRL, BSTRA, BNULLST auf Zielstringposition vor.

b) Der als S vereinbarte Speicherbereich wird vorher durch eine Eingabe-Anweisung mittels G-Format beschrieben.

c) Der als S vereinbarte Vektor (dynamisch) wird vorher durch eine Eingabe-Anweisung mittels G-Format beschrieben.

2.1.2. Definition eines Teilstrings

Gegeben sei ein String der Länge n. Ein Teilstring ist ein Ausschnitt dieses Strings. Er wird durch folgendes Wertetripel beschrieben:

S : Anfangsadresse des Gesamtstrings
P : Position innerhalb des Strings, bei der der Teilstring beginnt
N : Länge des Teilstrings (Anzahl Zeichen)

In Abhängigkeit der Besetzung von N gelten für die Teilstringlänge folgende Festlegungen:

a) $N < 0$ Der Teilstring umfaßt vom Zeichen mit der Nummer P alle Zeichen bis zum Ende des Strings. Dies sind die Zeichen mit den Positionsnummern
P, P + 1, P + 2 ... n (n = Länge des Gesamtstrings)

b) $N > 0$ (und $P + N - 1 \leq n$)
Der Teilstring umfaßt die Zeichen des Strings von der Positionsnummer P bis zur Positionsnummer P + N - 1

c) $N = 0$ Der Teilstring ist leer.

Beispiele:

Gegeben sei ein String mit der Anfangsadresse ST.
Die Zeichenfolge im String lautet:

DIES ␣ IST ␣ EIN ␣ STRING

Folgende Teilstrings werden beschrieben:

		erzeugter Teilstring
a)	(ST, 9, - 1)	␣ EIN ␣ STRING
b)	(ST, 6, 7)	IST ␣ EIN
c)	(ST, 8, 0)	Der Teilstring ist leer

3. Prozedurbeschreibungen

3.1. Erzeugen von Strings

3.1.1. String aus BCPL-String

Syntax:

BSTRL (ZS, B, R)

Bezeichnung:

ZS : Zielstring

B : Adresse eines BCPL-Strings oder Stringkonstante

R : Replikator

Aufrufform:

Routine

Wirkung:

Der durch B adressierte BCPL-String wird R-mal hintereinander ab der durch ZS definierten Adresse im Stringhandling Format abgelegt.

Beispiele:

a) LET FELD = TABLE 0,0,0,0,0,0
BSTRL (FELD, "ABCDEFGH", 2)
Erzeugter Zielstring: ABCDEFGHABCDEFGH

b) LET ZIELSTR = VEC 4
BSTRL (ZIELSTR, "1111", 3)
Erzeugter Zielstring: 111111111111

3.1.2. String aus A-Format-Text

Syntax:

BSTRA (ZS, A, W, N)

Bezeichnung:

ZS : Zielstring

A : Variable oder Vektoradresse (abhängig von N)

W : Anzahl Zeichen pro Halbwort

N : 0 (A = Variable) oder Anzahl Halbworte (A = Vektoradresse)

Aufrufform:

Routine

Wirkung:

Mit der Prozedur BSTRA können Zeichenfolgen, die zuvor im A-Format auf Variable oder Vektoren eingelesen wurden, in Stringhandling Strings gewandelt werden.

Ist A eine Variable (N = 0), werden ihre rechtsbündigen W Zeichen ab der durch ZS bezeichneten Adresse als Stringhandling String abgelegt.

Ist A eine Vektoradresse (N > 0), werden von den ersten N Halbworten des Vektors alle W rechtsbündigen Zeichen eines Vektorelementes ab der durch ZS bezeichneten Adresse als Stringhandling String abgelegt.

Beispiele:

a) Ein im A-Format eingelesener Vektor wird in einen Stringhandling-String gewandelt.

Daten: ␣␣A␣␣-␣␣F␣␣O␣␣R␣␣M␣␣A␣␣T

```
LET ADR=TABLE 0 REP 3 AND
A=VEC 7
READ(8,[8A3[,8,LV A!0,LV A!1,LV A!2,LV A!3,LV A!4,LV A!5,
LV A!6,LV A!7)
BSTRA(ADR,A,1,8)
WRITE(9,[G18/[,1,ADR)
```

Ausgabe:

␣␣A␣␣-␣␣F␣␣O␣␣R␣␣M

b) Eine im A-Format eingelesene Variable wird in einen Stringhandling-String gewandelt.

Daten: LADER

```
          LET T=TABLE 0,0 AND
BVAR=NIL
READ(8,[A5[,1,LV BVAR)
BSTRA(T,BVAR,2,0)
WRITE(9,[G3/[,1,T)
```

erzeugter Zielstring:

```
ER
```

c) Einlesen eines variabel langen Satzes und wandeln in einen Stringhandling-String.

```
GLOBAL  &(  ST:1  &)
EXTERNAL BL.EA:READ,WRITE
NONREC 1:READ,WRITE
EXTERNAL BSTRA
MANIFEST  &(  EIN =8  &)
MANIFEST &( S.RECEND=21;SATZMAX=100;CHPW=3  &)
ST:  &(
LET I=0
AND V=VEC  SATZMAX
AND STRING=VEC SATZMAX/CHPW
&(SCHL  I:=I+1
READ(EIN,[C0[,1,LV V!I)
&)SCHL  REPEATUNTIL  V!I=S.RECEND
BSTRA(STRING,LV V!1,1,I)
FOR I=0 TO 6
WRITE(9,[Z/[,1,LV STRING!I)
WRITE(9,[G36/[,1,STRING)
WRITE(9,[N/[,1,I)
FINISH  &)
```

Konsoleingabe und Ausgabe:

```
          Π:SAG'Π.Π: MIRΠ.Π: WO
Π.
 3 D2C0C6
 3 6121AF
 3 CCC8D1
 3 21AFD6
 2 CE1500
 2 000000
 1 3FFFFF
SAG'! MIR! WO!
14
```

Die Ausrufezeichen im Ausgabestring stehen für die Zentralcode-oktaden '21' und '15', die beim Einlesen im C0-Format für Eingabeende und NEWLINE als Funktionswert zurückgeliefert werden (siehe auch C0-Format, H 3.11).

3.1.3. Nullstring

Syntax:

BNULLST (ZS1, [ZS2,...] PEND)

Bezeichnung:

ZSi : String

PEND : FALSE

Aufrufform:

Routine

Wirkung:

Die durch ZS adressierten Strings sind nach Aufruf leere Strings. Der letzte Parameter PEND muß mit FALSE besetzt sein.

Beispiel:

```
LET F1 = TABLE 0,0
LET F2 = VEC 1
BNULLST (F1, F2, FALSE)
```

3.2. Verändern von Strings

3.2.1. Verketten von Strings

Syntax:

BKET (ZS, S1, [S2,...] PEND)

Bezeichnung:

ZS : Zielstring

Si : String

PEND : FALSE

Aufrufform:

Routine

Wirkung:

An den durch ZS adressierten String werden alle angeführten Strings Si in der im Prozeduraufruf vorgegebenen Reihenfolge angekettet.
Der letzte Parameter PEND muß mit FALSE besetzt sein.

Fehlermeldung:

Falls einer der Strings oder der Zielstring nicht definiert ist.

Beispiele:

a) Vier Stringhandling-Strings werden miteinander verkettet.

```
LET Z=VEC 7
AND X=VEC 1
AND Y=VEC 1
AND U=VEC 1
AND PAR=FALSE
BSTRL(Z,"ADAM",1)
BSTRL(X,"EVA",1)
BSTRL(Y,"KAIN",1)
BSTRL(U," UND ",1)
PKET(Z,U,X,U,Y,PAR)
```

erzeugter Zielstring:

```
ADAM␣UND␣EVA␣UND␣KAIN
```

b) Drei Stringhandling-Strings werden miteinander verkettet.

Daten: HIMMEL ␣ SACRAMENTO ␣ LUJAKRUZITUERKEN

```
LET PAR=FALSE
AND V1=VEC 4
AND V2=VEC 3
AND V3=VEC 11
READ(NUM,"G13",1,V1)
READ(NUM,"G10",1,V2)
READ(NUM,"G12",1,V3)
PKET(V3,V1,V2,PAR)
WRITE(9,"G25/",1,V3)
```

erzeugter Zielstring und Ausdruck:

```
KRUZITUERKEN HIMMEL SACRAMENTO LUJA
```

3.2.2. String aus Teilstring

Syntax:

[KET (ZS, PZ, T1, [T2,...] PEND)

Bezeichnung:

ZS : Zielstring

PZ : Zeichenposition im Zielstring

Ti : Teilstring (drei Parameter, siehe N 2.1.2)

PEND: FALSE

Aufrufform:

Routine

Wirkung:

Ab dem ersten Zeichen hinter (!) der Zeichenposition PZ des Zielstrings werden die Teilstrings in der angegebenen Reihenfolge nacheinander angekettet. Das Ende des letzten Teilstrings ist auch das Ende des Zielstrings Der so entstandene neue Zielstring kann kürzer oder länger als der ursprüngliche Zielstring sein.

Die Zeichenposition wird während des Verkettungsvorgangs mitgezählt. Nach Bearbeitungsende enthält die Positionsvariable PZ die Länge des neuen Zielstrings. Der letzte Parameter PEND muß mit FALSE besetzt sein.

Regel:

Auf Parameterposition muß für die Zeichenpositionsvariable deren Adresse übergeben werden (LV).

Das gleiche gilt für die Positionsvariable bei der Teilstringbeschreibung.

Fehlermeldungen:

a) wenn die angegebenen Zeichenpositionen der Teilstrings außerhalb der angegebenen Strings liegen,

b) wenn die Anzahl der Teilstringzeichen im zugehörigen String nicht vorgefunden wird.

Beispiele:

a) Aus zwei Stringhandling-Strings werden zwei Teilstrings verkettet.

```
LET ZPZ,ZP1,ZP2=16,6,99
AND ZS1=VEC 10
AND ZS2=VEC 11
BSTRL(ZS1,'DER TEUFEL LEBT NICHT IM PARADIES',1)
BSTRL(ZS2,'FAST IMMER IST ALLES IN DER HOELLE',1)
BTKET(ZS1,LV ZPZ,ZS2,LV ZP1,6,ZS2,LV ZP2,-1,FALSE)
```

erzeugter Zielstring:

```
DER TEUFEL LEBT IMMER IN DER HOELLE
```

ZPZ = 25

b) Aus zwei Stringhandling-Strings werden zwei Teilstrings verkettet.

```
LET ZIEL1=VEC 6
AND ZIEL2=VEC 5
 AND P1,P2,P3=4,5,15
 BSTRL(ZIEL1,'DER KAIN ISST HUMMER',1)
BSTRL(ZIEL2,'WER LEBT DENN DORT',1)
 BTKET(ZIEL1,LV P1,ZIEL2,LV P2,5,ZIEL2,LV P3,-1,FALSE)
```

erzeugter Zielstring:

```
DER LEBT DORT
```

P_1 = 13

3.2.3. Ersetzen eines Teilstrings

Syntax:

BERS (ZS, PZ, T)

Bezeichnung:

ZS : Zielstring

PZ : Zeichenposition im Zielstring

T : Teilstring (drei Parameter, siehe N 2.1.2)

Aufrufform:

Routine

Wirkung:

Im Zielstring werden ab der Zeichenposition PZ (einschließlich) die folgenden Zeichen durch die Zeichen des Teilstrings ersetzt.
Dabei werden genau soviel Zeichen ersetzt, wie der Teilstring besitzt.
Der Zielstring kann dadurch verlängert werden.
Ein leerer Teilstring hat keine Wirkung. Die Zeichenposition wird während des Verkettungsvorgangs mitgezählt. Nach Bearbeitungsende ist der Wert der Positionsvariablen PZ gleich der Position des letzten ersetzten Zeichens im Zielstring.

Regel:

Auf Parameterposition muß für die Positionsvariable PZ deren Adresse übergeben werden (LV).
Das gleiche gilt für die Zeichenpositionsvariable bei der Teilstringbeschreibung.

Fehlermeldungen:

a) Wenn die Zeichenposition des Teilstrings oder des Zielstrings außerhalb des zugehörigen Strings liegt,

b) wenn die Anzahl der Teilstringzeichen im zugehörigen String nicht vorgefunden wird.

N

Beispiele:

a) Im String STRING werden ab Position 14 die folgenden Zeichen durch einen Teilstring des Strings TEIL ersetzt.

```
LET Z1,Z2,J,POS=14,5,35,19
AND STRING=TABLE 0 REP 11
AND TEIL=TABLE 0 REP 0
AND ST=TABLE 0 REP 19
BSTRL(STRING,'ADAM UND EVA IM PARADIES',1)
BSTRL(TEIL,'SIE LEBEN NICHT MEHR LANGE',1)
BERS(STRING,LV Z1,TEIL,LV Z2,16)
```

erzeugter String:

```
ADAM UND EVA LEBEN NICHT MEHR

Z1 = 22
```

b) Beispiel a) wird fortgesetzt

```
BSTRL(ST,'NICHTS GEHT UEBER STRINGHANDLING',1)
BERS(ST,LV J,STRING,LV POS,-1)
```

erzeugter Zielstring:

```
NICHTS GEHT UEBER STRING NICHT MEHR

J = 35
```

3.2.4. Einfügen eines Teilstrings

Syntax:

BEINFG (ZS, PZ, T)

Bezeichnung:

ZS : Zielstring

PZ : Zeichenposition im Zielstring

T : Teilstring (drei Parameter, siehe N 2.1.2)

Aufrufform:

Routine

Wirkung:

Der durch T beschriebene Teilstring wird hinter der durch PZ bezeichneten Zeichenposition im Zielstring eingefügt.
Die Zeichenposition wird während des Einfügungsvorganges mitgezählt.
Nach Bearbeitungsende ist der Wert der Positionsvariablen gleich der Position des ersten Zeichens nach der Einfügung.

Regel:

Auf Parameterposition muß für die Positionsvariable PZ deren Adresse übergeben werden (LV).
Das gleiche gilt für die Zeichenpositionsvariable bei der Teilstringbeschreibung.

Fehlermeldungen:

a) Wenn die Zeichenposition des Teilstrings oder des Zielstrings außerhalb des zugehörigen Strings liegt,

b) wenn die Anzahl der Teilstringzeichen im zugehörigen String nicht vorgefunden wird.

Beispiele:

In String Z1 wird bei der Position 14 ein Teilstring aus Z2 eingefügt.

```
LET V=TABLE 0,0
AND Z1=VEC 20
AND Z2=VEC 20
AND ZAEHL,POS=13,1
BSTRL(Z1,[ADAM UND EVA IM PARADIES[,1)
BSTRL(Z2,[LEBEN NICHT MEHR[,1)
BEINFG(Z1,LV ZAEHL,Z2,LV POS,12)
WRITE(9,[GBB ,'POSITION DES ERSTEN ZEICHENS NACH EINFUEGU
NG= ',N/[,2,Z1,ZAEHL)
```

Ausdruck:

```
ADAM UND EVA LEBEN NICHT IM PARADIES
POSITION DES ERSTEN ZEICHENS NACH EINFUEGUNG= 26
```

3.3. Analysieren von Strings

3.3.1. Länge eines Strings

Syntax:

BLNG (S)

Bezeichnung:

S : String

Aufrufform:

Funktion

Wirkung:

Der durch S adressierte String wird auf seine Zeichenanzahl untersucht. Die Anzahl dieser Untersuchung wird als Funktionswert geliefert. Ist der zu untersuchende String ein Nullstring, ist auch der Funktionswert 0.

Fehlermeldung:

Wenn der String nicht definiert ist.

Beispiele:

a)
```
LET  J = NIL
AND STR = TABLE  0,0,0,0,0,0,0,0
BSTRL (STR, "OMA␣UND␣OPA", 2)
J : = BLNG (STR)
   J  →  22
```

b)
```
LET  N = NIL
AND  V = TABLE  0,0
BNULLST (V, FALSE)
N : = BLNG (V)
   V  →  0
```

3.3.2. Vereinbare Zeichenklassen

Syntax:

BKLASSE (I1, K1, [I2, K2, ...]PEND)

Bezeichnung:

I : Zeichenintervall oder Sonderzeichen
K : Klassenzahl
PEND : FALSE

Aufrufform:

Routine

Wirkung:

Mit der Prozedur BKLASSE ist es möglich, bestimmten Zeichen (- gruppen) und Sonderzeichen Klassenzahlen zuzuordnen, über die in der Prozedur BLESZ (siehe I 2.1.2) eingelesene Zeichen bezüglich ihrer Klassenzugehörigkeit analysiert werden können.

Parameter I:

Er beschreibt die Zeichen (- intervalle) und Sonderzeichen, denen eine Klasse zugeordnet werden soll, in folgender Form (Beispiele):

"Z1 - Z2" alle Zeichen von Zeichen Z1 bis Zeichen Z2, dabei müssen entweder beide Zeichen Buchstaben (z. B. "A - K") oder beide Zeichen Ziffern sein (z. B. "0 - 7")

" Z " ein Zeichen, welches Buchstabe, Ziffer oder Sonderzeichen ist (z. B. "8" oder "Y" oder "•")

" * Z" ein Zeichen, daß über eine Ersatzdarstellung (siehe G 4.3) angesprochen werden muß (z. B. "*" " oder "**")

Parameter K:

Den so definierten Zeichen (- gruppen) wird durch den folgenden Parameter K eine numerische Klasse zugeordnet. Die Klasse 0 wird nach einem BKLASSE-Aufruf all den Zeichen zugeordnet, die im Aufruf explizit keiner Klasse zugeordnet wurden.
Fehlt die Parameterliste ganz, wird jedem Zeichen die Klasse 0 zugeordnet.
Der letzte Parameter PEND muß mit FALSE besetzt sein.

Fehlermeldungen:

a) Falls der BCPL-String nicht den vorgegebenen Syntaxforderungen genügt.

b) Falls die Klassenzahl negativ ist.

Beispiele:

a) BKLASSE ("0-9", 1, "A-S", 2, "T", 3 "U-Z", 2, FALSE)

Den Ziffern 0-9 wird die Klasse 1, den Zeichen A bis S und U bis Z die Klasse 2, dem Zeichen T die Klasse 3 zugeordnet.
Alle anderen Zeichen haben die Klasse 0.

b) BKLASSE ("+", 1, "-", 2, "/", 3, "**", 4, "* S", 5, FALSE)

Ergibt folgende interne Klassenzuordnung:

Klasse 1:	Zeichen +	
Klasse 2:	Zeichen -	
Klasse 3:	Zeichen /	
Klasse 4:	Zeichen *	siehe Ersatzdarstellungen (G 4.3)
Klasse 5:	Zeichen ␣	
Klasse 0:	alle anderen Zeichen	

3.3.3. Lies ein Zeichen und gib Klassenwert

Syntax:

BLESZ (ZS, S, PS)

Bezeichnung:

ZS : Zielstring
S : String
PS : Zeichenposition im String S

Aufrufform:

Funktion

Wirkung:

Es wird ein Zeichen aus dem String S gelesen, dessen Position innerhalb des Strings durch den Parameter PS angegeben wird.
Dieses Zeichen wird erstes und alleiniges Zeichen im Zielstring ZS.
Der Funktionswert des Aufrufes ist gleich dem Klassenwert des gelesenen Zeichens, der ihm durch die Prozedur BKLASSE zugeordnet wurde.
Nach Bearbeitungsende ist der Wert der Positionsvariablen PS die um 1 erhöhte Leseposition, wenn der String noch nicht zu Ende ist. Bei Stringende enthält diese Variable eine negative Zahl.

Regel:

Auf Parameterposition ist für die Positionsvariable PS deren Adresse zu übergeben (LV).

Beispiel:

Ein Eingabestring wird zeichenweise untersucht und jedem Zeichen wird der durch BKLASSE definierte Klassenwert zugeordnet.

Eingabestring S: AX3/+ZN/0YH

```
LET ZS=TABLE 0,0
AND PS,ZAEHL,I=NIL REP 3
AND KLASSEN=VEC 30
AND S=VEC 3
BKLASSE('A-W',1,'X',2,'Y-Z',1,'1-9',3,FALSE)
READ(S,'S11',1, S)
FOR I= 1 TO 30 DO
&( PS:=1
KLASSEN!I:=BLESZ(ZS,S,LV PS)
IF PS LS 0 &( ZAEHL:=I;BREAK &)
WRITE(S,'ZEICHEN= ',C1,'  PS= ',N,'  WERT= ',N/('3,ZS,PS
,KLASSEN!I)
&)
```

Ausdruck:

```
ZEICHEN= A   PS= 2   WERT= 1
ZEICHEN= X   PS= 3   WERT= 2
ZEICHEN= 3   PS= 4   WERT= 3
ZEICHEN= /   PS= 5   WERT= 3
ZEICHEN= +   PS= 6   WERT= 0
ZEICHEN= Z   PS= 7   WERT= 1
ZEICHEN= N   PS= 8   WERT= 1
ZEICHEN= /   PS= 9   WERT= 0
ZEICHEN= 0   PS= 10  WERT= 0
ZEICHEN= Y   PS= 11  WERT= 1
```

3.3.4. Baue Liste auf

Syntax:

BLISTE (NA, B)

Bezeichnung:

NA : Name einer Liste

B : BCPL-String

Aufrufform:

Routine

Wirkung:

Der BCPL-String wird zeichenweise gelesen. Nacheinander wird jedes gelesene Zeichen in der Liste NA (in je einem Ganzwort) derart abgelegt, daß die Liste für die Funktionen BLBISB, BLSOLZ, BIPOSZ und BIPOSZN verarbeitungsgerecht vorliegt.

Regel:

Die Liste NA muß als TABLE oder Vektor mindestens doppelt so viele BCPL-Elemente umfassen, wie der BCPL-String Zeichen enthält.

Beispiel:

```
BCPL-String B  ", ; ! . "

        LET  LIST = TABLE  0,0,0,0,0,0,0,0
        BLISTE  (LIST, B)
```

Nach Aufruf enthält die Liste LIST die Zeichen , ; ! . in je einem Ganzwort.

3.3.5. Lies bis Zeichen in Liste auftritt

Syntax:

BLBISZ (ZS, S, PS, NA)

Bezeichnung:

ZS : Zielstring

S : String

PS : Positionsnummer in der Liste NA

NA : Name einer Liste

Aufrufform:

Funktion

Wirkung:

Die Zeichen des Lesestrings S werden ab Position PS solange im Zielstring ZS aufgereiht, bis eines der gelesenen Zeichen in der mit der Prozedur BLISTE vereinbarten Liste der Trennzeichen vorkommt. Das Trennzeichen selbst wird nicht mit in den Zielstring übernommen.

Die Funktion BLBISZ liefert als Funktionswert die Positionsnummer des Trennzeichens in der Liste NA. Wird der String bis zum Ende gelesen und keine Übereinstimmung mit einem der in der Liste stehenden Zeichen gefunden, ist der Funktionswert 0.

Nach Bearbeitungsende ist der Wert der Positionsvariablen PS bei gefundenem Trennzeichen gleich der Positionsnummer des ersten Zeichens hinter dem Trenner im Lesestring.

Der Wert von PS ist negativ, wenn für keines der Zeichen im Lesestring eine Übereinstimmung mit dem Trennzeichen in der Liste NA gefunden wird oder wenn der Trenner das letzte Zeichen im Lesestring ist.

Regel:

Auf Parameterposition muß für die Positionsvariable PS deren Adresse übergeben werden (LV).

Fehlermeldungen:

a) Falls der Zielstring nicht definiert ist,

b) falls die angegebene Zeichenposition nicht im Lesestring liegt,

c) falls die Liste NA undefiniert ist.

Beispiel:

Im folgenden Beispiel werden zunächst die als BCPL-String eingelesenen Zeichen durch den BLISTE-Aufruf einzeln in je einem Ganzwort (BLISTE-gerecht) abgelegt, was durch das Format des folgenden WRITE-Befehls berücksichtigt wird. Die folgenden Statements untersuchen einen String nach den in einem weiteren BLISTE-Aufruf vorgegebenen Zeichen. Als String werden durch einen BLBISZ-Aufruf all die Zeichen aufgereiht, die nicht mit den durch BLISTE vereinbarten Zeichen übereinstimmen.

Eingabestring STRING: +*/-

```
LET ST=VEC 4
AND Y=TABLE 0 REP 12
AND TR=VEC 3
AND POS,I1=1,NIL
AND STRING=VEC 1 AND LIST=TABLE 0 REP 8
READ(8,[S4[,1,STRING)
BLISTE(LIST,STRING)
WRITE(9,[441/[,4,LIST!1,LIST!3,LIST!5,LIST!7)
BSTRL(ST,[DER ZUG. DAS AUTO[,1)
BLISTE(TR,[ .[)
FOR I=1 TO 5
&(FOR  I1:=BLBISZ(Y,ST,LV POS,TR)
WRITE(9,[I3,2X,C10,2X,I4/[,3,I1,Y,POS)  &)FOR
```

Ausdruck:

```
+*/-
001  DER          0005
002  ZUG          0009
001               0010
001  D S          0014
000  AUTO         -017
```

3.3.6. Lies solange Zeichen aus Liste auftreten

Syntax:

BLSOLZ (ZS, S, PS, NA)

Bezeichnung:

ZS : Zielstring

S : String

PS : Zeichenposition im String

NA : Name einer Liste

Aufrufform:

Funktion

Wirkung:

Die Zeichen des Lesestrings ab der Zeichenposition PS werden im Zielstring aufgereiht solange das gelesene Zeichen in der Liste NA vorkommt. Die Anzahl der in den Zielstring gelesenen Zeichen wird als Funktionswert von BLSOLZ zurückgeliefert.
Die Zeichenpositionsvariable enthält nach Aufruf entweder die Position des unbekannten, nicht in der Liste NA gefundenen Zeichens, oder bei Stringende einen negativen Wert.

Regel:

Auf Parameterposition muß für die Zeichenpositionsvariable deren Adresse übergeben werden (LV).

Fehlermeldungen:

a) Falls der Zielstring nicht definiert ist,
b) falls die angegebene Zeichenposition nicht im Lesestring liegt,
c) falls die Liste NA undefiniert ist.

Beispiel:

Der vorgegebene String ST1 wird durch BLSOLZ auf die Zeichen untersucht, die in dem vorangehenden BLISTE-Aufruf definiert wurden.
Es werden bei einem BLSOLZ-Aufruf all die Zeichen als String aufgereiht, die auch im String des BLISTE-Aufrufs vorkommen. Das erste Zeichen des Strings, welches nicht in der Liste vorkommt, beendet den BLSOLZ-Aufruf.

```
LET LIST=VEC 51
AND ST1=VEC 8
AND ZS=VEC  25
AND POS,ANZ= 0,NIL
BLISTE(LIST,[ABCDEFGHIJKLMNOPQRSTUVWXYZ[)
BSTRL(ST1,[HEUTE;MORGEN.UEBERMORGEN![,1)
&(   POS:=POS+1
ANZ:=BLSOLZ(ZS,ST1,LV POS,LIST)
WRITE(9,['ZEICHENANZAHL= ',N,'  GELESENER STRING= ',G1
8/[,2,ANZ,ZS)
WRITE(9,[19X,'POSITION DES UNBEKANNTEN ZEICHENS= ',N/[
,1,POS)
   &)  REPEATUNTIL  POS=BLNG(ST1)
```

Ausdruck:

```
ZEICHENANZAHL= 5   GELESENER STRING= HEUTE
                   POSITION DES UNBEKANNTEN ZEICHENS= 6
ZEICHENANZAHL= 6   GELESENER STRING= MORGEN
                   POSITION DES UNBEKANNTEN ZEICHENS= 13
ZEICHENANZAHL= 11  GELESENER STRING= UEBERMORGEN
                   POSITION DES UNBEKANNTEN ZEICHENS= 25
```

3.3.7. Bestimme erstes Auftreten eines Zeichens

Syntax:

BIPOSZ (T, NA, Z)

Bezeichnung:

T : Teilstring (drei Parameter, siehe Kap. N 2.1.2)

NA : Name einer Liste

Z : Listenzeiger

Aufrufform:

Funktion

Wirkung:

Im Teilstring T wird ab der angegebenen Zeichenposition untersucht, ob die Zeichen mit einem in der Liste angegebenen Zeichen identisch sind. Das erste Zeichen, welches sowohl im Teilstring als auch in der Liste vorkommt, beendet die Suche. In diesem Fall wird als Funktionswert diese Zeichenposition gezählt vom Teilstringanfang, übergeben.
Wird keines der im Teilstring enthaltenen Zeichen in der Liste gefunden, ist der Funktionswert 0. Bei erfolgreicher Suche zeigt die Zeichenpositionsvariable des Teilstrings auf das gefundene Zeichen, bezogen auf den Stringanfang; andernfalls bleibt sie unverändert.
Der Listenzeiger Z weist bei erfolgreicher Suche auf die Position des Zeichens in der Liste, andernfalls bleibt auch er unverändert.

Regel:

Auf Parameterposition der Zeichenpositions- und Listenzeigervariablen muß jeweil deren Adresse übergeben werden (LV).

Fehlermeldungen:

a) Wenn die angegebene Zeichenposition nicht im String liegt,

b) wenn die Zeichenliste nicht definiert ist.

Beispiele:

a) Der String A wird nach den Zeichen untersucht, die in der Liste LI definiert wurden. Das erste gemeinsame Zeichen beendet die Suche.

```
LET LI=VEC 3
AND A=VEC 6
AND K,IP,ILI=NIL,3,NIL
BSTRL(A,[HAUPTSATZ, NEBENSATZ.[,1)
BLISTE(LI,[,.[)
K:=BIPOSZ(A,LV IP,-1,LI,LV ILI)
WRITE(9,[315/[,3,LI,'H'E'K,IP,ILI)
```

Ausdruck:

```
0000,0001000001
```

b)

```
LET PF,K,ZEIG=2,NIL,1000
AND F=VEC 5
AND NA=VEC 3
BSTRL(F,[A B C F . G[,1)
BLISTE(NA,[.![)
K:=BIPOSZ(F,LV PF,7,NA,LV ZEIG)
WRITE(9,[N,2X,N,2X,N/[,3,K,PF,ZEIG)
```

Ausdruck:

```
0  2  0
```

3.3.8. Bestimme erstes Nichtauftreten eines Zeichens

Syntax:

BIPOSZN (T, NA)

Bezeichnung:

T : Teilstring (drei Parameter, siehe N 2.1.2)

NA : Name einer Liste

Aufrufform:

Funktion

Wirkung:

Im Teilstring wird ab der angegebenen Zeichenposition untersucht, ob die folgenden Zeichen mit einem der Zeichen in der Liste NA identisch sind. Die Suche wird abgebrochen, wenn ein Zeichen nicht in der Liste gefunden wird.

Der Funktionswert ergibt sich aus der Zeichenposition des nicht in der Liste gefundenen Zeichens, vom Teilstringanfang gezählt.

Sind alle im Teilstring stehenden Zeichen auch in der Liste vertreten, ist der Funktionswert 0.

Wurde im Teilstring ein Zeichen gefunden, das nicht in der Liste steht, enthält die Zeichenpositionsvariable nach dem Aufruf die Zeichenposition dieses Zeichens, gezählt vom Stringanfang, sonst bleibt sie unverändert.

Regel:

Auf Parameterposition der Zeichenpositionsvariablen muß deren Adresse übergeben werden (LV).

Fehlermeldungen:

a) Wenn die angegebene Zeichenposition nicht im String liegt,

b) wenn die Zeichenliste nicht definiert ist.

Beispiele:

a) In einem String werden alle Nichtziffernzeichen gesucht.

```
LET ZIF=VEC 19
AND M=VEC 4
AND KP,K1,K2=1,NIL REP 2
BSTRL(M,[3.14159265359,[,1)
BLISTE(ZIF,[0123456789[)
K1:=BIPOSZN(M,LV KP,-1,ZIF)
WRITE(9,[N,3X,N/[,2,K1,KP)
KP:=KP+1
K2:=BIPOSZN(M,LV KP,-1,ZIF)
WRITE(9,[N,3X,N/[,2,K2,KP)
```

Ausdruck:

```
2   2
12  14
```

b) Im folgenden Beispiel werden aus einem String nur die Ziffern, Vorzeichen und Blanks gesucht, als String aufgelistet und ausgegeben.

```
LET POINT,POS,VAR=0,1,NIL
AND STRING=VEC 20
AND AUFLIST=VEC 20
AND NOZIF=VEC 53
BLISTE(NOZIF,[=ABCDEFGHIJKLMNOPQRSTUVWXYZ[)
BNULLST(AUFLIST,FALSE)
BSTRL(STRING,[WERT=-90  POSITION=100  ERGEBNIS=10000[,
1)
&(REP
VAR:=BIPOSZN(STRING,LV POS,-1,NOZIF)
IF VAR=0  BREAK
WRITE(9,[N,3X,N/[,2,VAR,POS)
RTKET(AUFLIST,LV POINT,STRING,LV POS,1,FALSE)
IF POS=BLNG(STRING)  BREAK
POS:=POS+1
&)REP   REPEAT
WRITE(9,[G25/[,1,AUFLIST)
```

Ausdruck:

```
6    6
1    7
1    8
1    9
1    10
10   20
1    21
1    22
1    23
1    24
10   34
1    35
1    36
1    37
1    38
-90  100  10000
```


3. 4. Vergleichen von Strings

3. 4. 1. Identität von Strings

Syntax:

BIDENT (S1, S2)

Bezeichnung:

S1: } String
S2: }

Aufrufform:

Funktion

Wirkung:

Die Strings S1 und S2 werden auf Identität bezüglich ihrer Länge und Zeichen untersucht. Das Ergebnis der logischen Funktion ist TRUE, wenn S1 und S2 identisch sind, sonst FALSE.

Fehlermeldung:

Falls einer der Strings nicht definiert ist.

Beispiele:

a) BSTRL (STR1, "KALAUER", 1)
BSTRL (STR2, "KALAUER␣", 1)
ERG: = BIDENT (STR1, STR2)

Ergebnis: ERG → FALSE

b) BSTRL (S1, "ARI", 1)
BSTRL (S2, "ARI", 1)
ERG: = BIDENT (S1, S2)

Ergebnis: ERG → TRUE

3.4.2. Teilstring in Teilstring enthalten

Syntax:

BITEIL (T1, T2)

Bezeichnung:

T1: } Teilstring (je drei Parameter, siehe Kap. N 2.1.2)
T2: }

Aufrufform:

Funktion

Wirkung:

Die Funktion prüft, ob der Teilstring T2 identisch im Teilstring T1 enthalten ist. Wenn ja, wird als Funktionswert die Zeichenposition in T1 (relativ zum Teilstringanfang) übergeben, ab der der Teilstring T2 in T1 beginnt. Ergibt sich keine Identität, ist der Funktionswert 0. Die Zeichenpositionsvariable in T1 enthält bei Identität nach dem Aufruf die Position des Zeichens ab der der Teilstring T2 beginnt (relativ zum Stringanfang). Ist keine Identität vorhanden, bleibt die Zeichenpositionsvariable in T1 unverändert.

Regel:

Auf Parameterposition der Zeichenpositionsvariablen in T1 und T2 müssen deren Adressen übergeben werden (LV).

Beispiele:

a) Es wird geprüft, ob der vorgegebene Teilstring in K2 identisch im vorgegebenen Teilstring von K1 enthalten ist.

```
LET PT1,PT2=3,6
AND K1=VEC 4
AND K2=VEC 9
BSTRL(K1,[STRINGHANDLING[,1)
BSTRL(K2,[EINE HAND WAESCHT DIE ANDERE[,1)
POS:=BITEIL(K1,LV PT1,10,K2,LV PT2,4)
WRITE(9,[N,2X,N/[,2,POS,PT1)
```

Ausdruck:

```
5  7
```

b)

```
LET P1,P2,POS=2,5,NIL
AND A=VEC 3
AND B=VEC 3
BSTRL(A,[OMA UND OPA[,1)
BSTRL(B,[OPA UND OMA[,1)
POS:=BITEIL(A,LV P1,9,B,LV P2,6)
WRITE(9,[N,2X,N/[,2,POS,P1)
```

Ausdruck:

```
0  2
```

3.5. Wandeln von Strings

3.5.1. Wandeln auf A-Format

Syntax:

BAFORM (A, S, N)

Bezeichnung:

A : Variable oder Vektoradresse

S : String

N : N = 0 (→ A ist Variable)

N ≠ 0 (→ A ist Vektoradresse)

Aufrufform:

Routine

Wirkung:

Ist N = 0, werden die Zeichen des Strings A formatgerecht in der Variablen A abgelegt.

Ist N ≠ 0, werden die Zeichen des Strings A formatgerecht ab der durch A bezeichneten Vektoradresse abgelegt.

Regel:

Ist das Ablageziel eine Variable (N = 0), muß auf Parameterposition die Adresse der Variablen übergeben werden.

Fehlermeldung:

Wenn der String zu lang ist

Beispiele:

```
LET VEK=VEC 5
AND VT2=VEC 8
AND A=NIL
AND VT1=VEC 1
BSTRL(VT1,[ST9[,1)
BAFORM(LV A,VT1,0)
WRITE(9,[A3/[,1,A)
BSTRL(VT2,[A1 B2 C3 [,1)
BAFORM(VEK,VT2,3)
WRITE(9,[5(Z/)[,5,LV VEK!0,LV VEK!1,LV VEK!2,LV VEK!3,LV
VEK!4)
WRITE(9,[9(A1/)[,9,VEK!0,VEK!1,VEK!2,VEK!3,VEK!4,VEK!5,V
K!6,VEK!7,VEK!8)
```

Ausdruck:

```
ST?
 1 0000C0
 1 0000B1
 1 0000AF
 1 0000C1
 1 0000B8
A
1

B
2

C
3
```

Der String VT1 wird in A-Format gewandelt und mit entsprechendem Formatschlüssel ausgegeben.
Der String VT2 wird in A-Format gewandelt und mit entsprechendem Formatschlüssel ausgegeben.

Achtung:
Werden Vektoren von Stringhandling in A-Format gewandelt, wird jeweils ein Zeichen A-formatgerecht in ein Vektorelement rechtsbündig abgelegt (siehe dieses Beispiel Ausgabe im Z-Format).

3.5.2. Wandeln in BCPL-String

Syntax:

BSTRB (B, S)

Bezeichnung:

B : BCPL-String
S : String

Aufrufform:

Routine

Wirkung:

Der durch S adressierte String wird nach BCPL-Stringkonventionen (Länge des Strings im ersten Stringzeichen) ab der durch B bezeichneten Adresse abgelegt.

Fehlermeldung:

Wenn der durch B adressierte BCPL-String mehr als 155 Zeichen aufnehmen soll.

Beispiel:

Ein Stringhandling-String wird in einen BCPL-String gewandelt.

```
LET SH=TABLE    0 REP 5
AND B=VEC 5
BSTRL(SH,[STRINGTEXTENDE [,1)
WRITE(9,[G15/[,1,SH)
BSTRB(B,SH)
WRITE(9,[S15/[,1,B)
```

Ausdruck:

```
STRINGTEXTENDE␣
STRINGTEXTENDE␣
```

ANHANG

ANHANG

1. Liste der reservierten Worte

AND	MANIFEST
BE	NE
BEGIN	NEQV
BREAK	NIL
BY	NONREC
CASE	NOT
COND	OF
DEFAULT	OR
DO	REM
END	REP
ENDCASE	REPEAT
EQ	REPEATUNTIL
EQV	REPEATWHILE
ET	RESULTIS
EXTERNAL	RETURN
FALSE	RSHIFT
FINISH	RV
FOR	SLCT
GE	STATIC
GLOBAL	SWITCHON
GOTO	TABLE
GR, GT	TEST
IF	THEN
INTO	TO
LE	TRUE
LET	UNLESS
LOGAND	UNTIL
LOGOR	VALOF
LS, LT	VEC
LSHIFT	VEL
LV	WHILE

2. Alternative Darstellung durch Sonderzeichen

Reserviertes Wort	Alternative Darstellung
BEGIN	‡(oder &(
COND	→
END	‡) oder &)
EQ	=
ET, LOGAND	/ \
GE	> =
GR, GT	>
LE	< =
LOGOR, VEL	\ /
LS, LT	<
LSHIFT	<<
LV	∂
NE	\ =
NOT	\
OF	::
RSHIFT	>>
RV	!

STICHWORTVERZEICHNIS

Zur Systemprogrammierung in BCPL

S

Programmierung von Operatoren

Allgemeines

An jedes BCPL-Programm, das nicht mit Version R übersetzt ist, wird das Rahmenprogramm BCPL&R anmontiert, das die Programmanfangs- und Endebehandlung durchführt und im Normalfall Dump- und Rückverfolgungsoperator startet.

Mit dem Rahmenprogramm bestehen folgende Kontakte:

Globalzelle 0 enthält die Adresse des Startsatzes, der Aufbau des Startsatzes ist operatorspezifisch.

Globalzelle1 enthält die Startadresse des BCPL-Programms, das ist die Adresse des Befehls im BCPL-Programm, der nach erfolgter Anfangsbehandlung angesprungen wird.
Im Normalfall wird die Globalzelle 1 durch eine Markenvereinbarung im BCPL-Programm initialisiert.

```
GLOBAL      $( START:1  $)
  :
  :
START : ......
```

Globalzelle 2 enthält die Adresse der Programmendebehandlung. Die FINISH-Anweisung wird auf einen Sprung auf diese Globalzelle abgebildet.

FINISH ≙ GØTØ Globalzelle 2

Die Globalzelle 2 wird vom Rahmenprogramm initialisiert.

Globalzelle 3 enthält die Adresse einer Fehlerabschlußroutine. Diese Routine wird mit einem Parameter FS versorgt und bewirkt:

a) bei FS = 1
   Dump der Variablengebiete und Beenden des Operatorlaufs mit Fehler.
b) bei FS = 2
   Beenden des Operatorlaufs mit Fehler ohne Ausgabe eines Dumps.

Wird ein BCPL-Programm mit VERSION = R übersetzt, so wird kein Rahmenprogramm mitanmontiert und der Benutzer kann sich ein eigenes Rahmenprogramm schreiben.
Dabei sind die Konventionen wie oben einzuhalten.
D.h.
Globalzelle 0 muß mit der Adresse des Startsatzes,
Globalzelle 2 und 3 mit der Adresse der Ende- bzw. Fehlerbehandlung initialisiert werden.

Übergang Rahmenprogramm - BCPL-Programmteil

Bevor das BCPL-Programm aufgerufen wird, müssen für die Freispeicherverwaltung die Anfangszeiger gesetzt werden.
Dies kann durch das Assemblermakro INITSTACK ([BASIS]) geschehen.
BASIS ist dabei die gerade Adress-Konstante eines Speicherbereichs, der dem BCPL-Programm als Freispeicher zur Verfügung steht.

Die Länge des benötigten Speicherbereichs ist dabei vom dynamischen Programmablauf abhängig und kann durch Versuch bestimmt werden.
Werden nur nichtrekursive Prozeduren gerufen und erfolgt der Übergang vom Assemblerrahmen zum BCPL-Programm durch einen Prozeduraufruf, so braucht kein Freispeicher zur Verfügung gestellt werden.
Bis zur Integration des Makros ist die Initialisierung der Freispeicherverwaltung von Hand vorzunehmen.
Das geschieht durch die Befehle:

```
TCB       BASIS,
XC        8,
ZX      4   11,
STACK = ASP JENACHDEM/G --FREISPEICHER--
BASIS = STACK/A,
```

Wird kein Freispeicher benötigt, so können diese Befehle entfallen, es empfiehlt sich jedoch wegen der besseren Fehlerdiagnostik, die Zeiger mit der Adresse eines schreibgeschützten Bereichs zu initialisieren.
Bei einem nicht gewollten Zugriff auf den Freispeicher wird dann ein Alarm ausgelöst.

Der Übergang vom Rahmen ins BCPL-Programm kann auf zwei Arten erfolgen

a) durch Sprung auf das Startlabel des BCPL-Programms, dies geschieht durch die Befehlfolge.
   SE Globalzelle 1

b) Durch Aufruf einer rekursiven oder nichtrekursiven BCPL-Prozedur, wobei die Prozedurvariable durch eine GLOBAL oder EXTERNAL-Deklaration im TAS-Programm bekannt gemacht wurde. (vgl. I 2).
   Der Prozeduraufruf geschieht durch das Makro CALL.
   Dabei ist das Hauptprogramm wie eine Prozedur zu betrachten.
   Diese Prozedur ist als rekursiv anzunehmen, falls rekursive Prozeduren am Programm beteiligt sind.
   Beispiel vgl. bei I 3.2.8.
   vgl. Beispiel B 17 + 18

Makro CALL

Über das Makro kann von einer Assemblerroutine oder vom Hauptprogramm aus eine BCPL-Prozedur aufgerufen werden.

Syntax:
CALL (<fnvar>, <quzon>, <zizon>, <dynsp>, <p1>, <p2>, ...)

<fnvar>	TAS-Variablenname, die zugeordnete Variable muß die Anfangsadresse der BCPL-Prozedur enthalten.
<quzon>	Nonrecklasse der rufenden Prozedur, ist die rufende Prozedur rekursiv, so ist <quzon> = 0 zu setzen. Sonst muß quzon eine zweistellige Zahl sein.
<zizon>	Nonrecklasse der gerufenen Prozedur analog zu <quzon>
<dynsp>	Auf gerade Zahl aufgefundete Anzahl der von der rufenden Prozedur belegten dynamischen Variablen. Dabei sind Parameter als dynamische Variable mitzuzählen. Im Normalfall ist <dynsp> identisch mit der Anzahl der Parameter (aufgerundet).
<p1>, <p2> ...	Parameter, die an die gerufene Prozedur übergeben werden. <pi> kann sein - der Name einer Assemblerhalbwortvariablen oder Konstanten - oder ein Halbwortliteral.

vgl. Beispiel B 18

1210.1. Einleitung

In dieser Komponente werden die Unterprogrammanschlußkonventionen für BCPL-Objekte beschrieben (TR440 Halbwortimplementierung).

1210.2. Benutzungsbeschreibung

1210.2.1. Externe Schnittstellen

siehe auch 18.5.1200.3.4

- Aufruf-Befehlsfolgen
  - rekursive Prozeduren
    Der Aufruf erfolgt mit dem Befehl
    SFBE adrv,
    wobei adrv eine Adreßvariable ist, die die Adresse der Ansprungstelle enthält;
  - nichtrekursive Prozeduren
    Der Aufruf erfolgt mit dem Befehl
    SUE adrv
    mit gleicher Bedeutung von adrv
- Versorgung
  - rekursive Prozeduren
    Bei rekursiven Prozeduren wird im RA die Adreßdifferenz des von der rufenden Prozedur benötigten Stacks mitgegeben. Der prozedurrelative Stackpointer (Indexzelle 8) zeigt auf den Anfang des Stackbereichs der rufenden Prozedur (dies ist immer eine GW-Adresse). Dieser Stackpegel muß also (falls überhaupt) in der gerufenen Prozedur aktualisiert werden. Eventuell

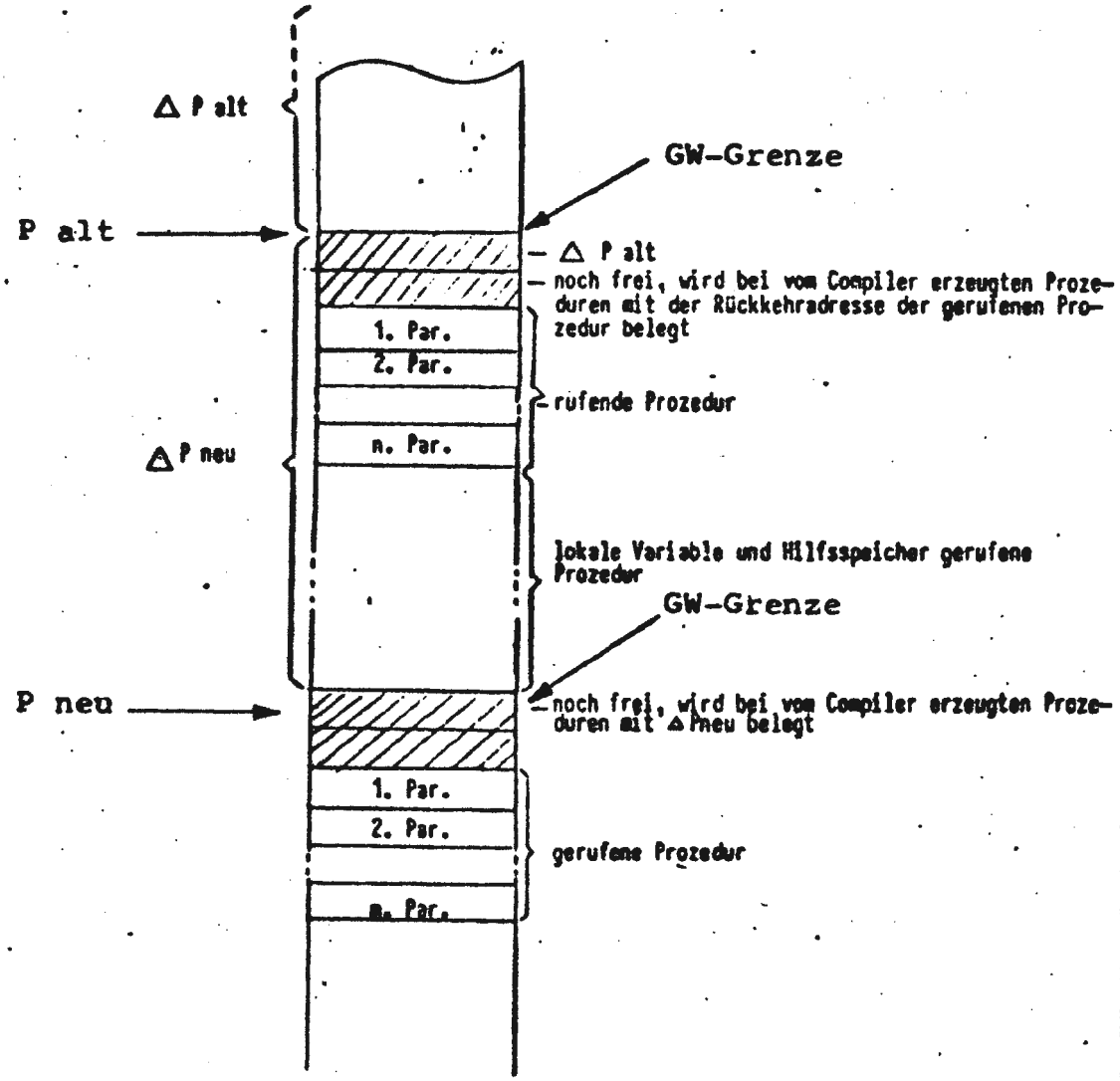
vorhandene Aktualparameter sind von der rufenden Prozedur vor dem Aufruf in den Stackbereich der gerufenen Prozedur abgelegt worden (und zwar mit ihrem Wert).

Ablageort: 1. Parameter: Wert aktualisierter

Stackpegel + 2

2. Parameter: Wert aktualisierter

Stackpegel + 3 usw.

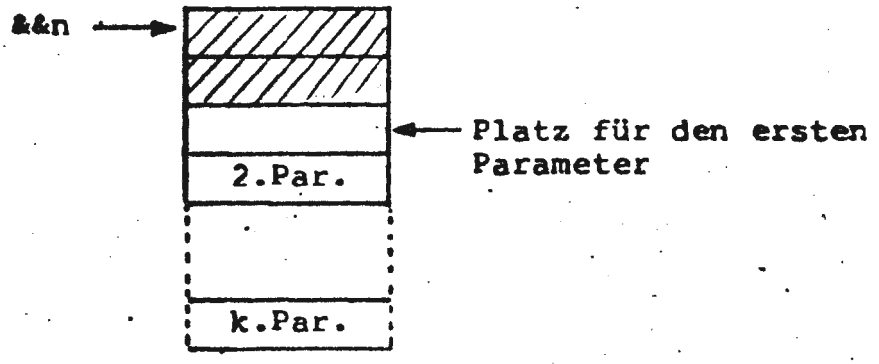


Das Bild beschreibt den Stackzustand nach dem Prozeduraufruf.

- nichtrekursive Prozeduren

  Die Parameter 2 und folgende stehen in einem Pseudostack (Commonzone, die auf GW-Grenze beginnt mit dem Namen &&n, wobei n die Zahl ist, die bei der Nonrecdeklaration als Klassenzahl angegeben wurde) beginnend mit der Relativadresse 3.

  Der 1. Parameter wird im RA übergeben



- Zugriffe auf den Globalvektor

  Der Globalvektor ist eine Commonzone mit dem Namen ZGLØBAL. Er beginnt auf GW-Grenze; die im BCPL-Programm angegebenen Nummern sind die Relativadressen in dieser Commonzone

- Zugriffe auf BCPL-Elemente, die als External gekennzeichnet sind

  Wird der erste in der BCPL-External-Anweisung aufgelistete Namen von dem nächsten durch ein Kolon getrennt, so ist er ein MO-Name und alle anderen sind Kontaktnamen bezgl. dieses MO-Namens; anderenfalls sind alle aufgelisteten Namen MO-Namen.

- Funktionswerte

  Der Funktionswert maximal 24 Bit lang wird rechtsbündig in RA übergeben.

- Makro's in der Öffentlichen Bibliothek

Die in den vorangegangenen Punkten beschriebene Schnittstelle sollte als nicht fixiert betrachtet werden. Deshalb wurde eine Reihe von Makros geschaffen, die es dem Benutzer ermöglichen sollen, eine Schnittstellenänderung hinzunehmen, ohne eine Programmänderung durchführen zu müssen.

Makroaufruf für

- Eingang

- rekursiv

RECENTRY (<Name>),

produziert die Befehlsfolge

*<NAME = SU <ENTRYCODE>, wobei <ENTRYCODE>
die folgende Befehlsfolge ist:

MFU	8,	(Index 8 enthält den Stackpegel)
TBC	1,	(Rückkehradresse retten)
MRX AC	8,	(Erhöhen Stackpegel um Inhalt RA)
C2	0,	(Retten der Stackpegeldifferenz)

- nicht rekursiv

NRECENTRY (<NAME [,<PARAMETERZAHL>, <ZØNEN NUMMER>]),

produziert die Befehlsfolgen

- * <NAME> = ASP 0/B, (wenn nur <NAME> als Parameter)
- * <NAME> = C2 && <ZØNENNUMMER> +2 (sonst)

- Deklaration des Pseudostacks

  Notwendig, da in einer Übersetzungseinheit TAS nicht mehrere Deklarationen einer CZØNE erlaubt.

  NRCZØNE (<ZØNEN-NR.>[,<PARAMETERZAHL>]),

  Das Makro deklariert eine Common-Zone mit dem Namen &&<ZØNEN-NR.>. und der Länge <PARAMETERZAHL>+2

- Rückkehr

  - rekursiv

    RECRET ( ),

    Das Makro erzeugt einen absoluten Sprung auf die Befehlsfolge

    E TCB 8, (Laden ΔP)
    RX BCN 8, (Korrektur des Pegels)
    MAB SE 1, (Rücksprung)

  - nichtrekursiv

    Das Makro erzeugt den Befehl

    MU S 0,

- Benutzung eines Parameters

  Die Benutzung eines Parameters wird mit dem Makro

  USEPARAM (<CODE>, <PARAMETERPØSITIØN>[,
  <ZØNEN-NR.>])

  ermöglicht; dabei ist CØDE ein TAS-Befehl, dessen Adreßteil den spezifizierten Parameter adressiert. Die Zonen-Nr. entfällt für eine rekursive Prozedur.

- Wechsel zwischen rekursiven und nichtrekursiven Prozeduren

Bei den vier verschiedenen Möglichkeiten wird folgender Code erzeugt:

- Fall 1 (rec ruft rec auf):

```
BA Deltap -- Deltap ist die Länge des zu kel-
             lernden Stack-Bereichs der ru-
             fenden Routine --
SFBE Adresse
```

- Fall 2 (nonrec ruft nonrec auf):

```
SUE Adresse
```

- Fall 3 (rec ruft nonrec auf):

```
XBA Deltap -- Deltap siehe unter Fall 1 --
XC NRP     -- Deltap wird für die Dauer des
              nonrec-Aufrufs gerettet in In-
              dexzelle 11 --
           -- diese beiden Befehle können ggf.
              durch ZX Deltap NRP ersetzt wer-
              den --
SUE Adresse
```

- Fall 4 (nonrec ruft rec auf):
(siehe dazu genauer: 13.3.2.5, Seiten 34, 34.1, 34.2)

```
XB NRP     -- Die Länge des zu kellernden Stack-
              bereichs der über nonrec liegenden
              rec wird geholt --
TBC && Nr. -- im nonrec-Stack Nr. gerettet, da
              nämlich die gerufene rec ihrer-
              seits nonrec aufrufen könnte --
RX BC P    -- die bei "rec ruft nonrec" nicht er-
              folgte Erhöhung von Index 8 wird
              nachgeholt --
           -- an dieser Stelle sind die Parame-
              ter für den Aufruf zu laden, In-
              dex 8 enthält den unteren Level --
```

```
BA          -- Pseudodeltap für Pseudostackbedarf
               der nonrec, dieser Wert ist 2 --

SFBE Adresse

TCB && Nr.-- Deltap kann wieder in --
XC NRP      -- Index 11 gespeichert werden --
RX BCN P    -- die für den rec-Aufruf nachgeholte
               Erhöhung von Index 8 zum nonrec-
               Aufruf wird rückgängig gemacht --
```

KENNEN SIE SCHON UNSERE ?

A

INTRINSICS

Zusammen mit dem neuen Compiler sollen einige Intrinsics und Makros für Codeanschlüsse zur Verfügung gestellt werden. Die Intrinsics sind zunächst durch Prozeduren realisiert, sie sollen später bereits im Compiler durch Erzeugung von inline-code verarbeitet werden. Die Externalfassung ist dabei nicht so mächtig wie die spätere Fassung, zum einen werden beim Aufruf Register verändert (stets das U-Register, bei Parameterversorgung mindestens auch das A-Register, bei Aufruf aus rekursiven Funktionen zusätzlich noch das B-Register), zum anderen gibt es keine "optionalen" Parameter, da die gerufene Routine nicht die Zahl der mitgegebenen Parameter erkennen kann.

Folgende Deklaration ist notwendig:

```
EXTERNAL  BL.INTR: MØVE,...
NØNREC    2: MØVE,...
```

Im Folgenden sind die zunächst vorgesehen Intrinsics und Makros beschrieben.

1) Intrinsics

a) MØVE (lng, adrquell, adrziel)

Transport von adrquell nach adrziel mit B, BZ bzw. WTV,
lng ist die Länge des Feldes in BCPL-Elementen (also HW-Längen!); die Adressen müssen gerade sein
Beispiel:
GLØBAL $(ZWEIGW : 100 $)

A

```
LET  V = VEC 4 AND  W =  VEC  4
.
.
.
MØVE (4, LV ZWEIGW, V)
```

b) MØVEB(lng, adrquell, adrziel)
Tansport von adrquell nach adrziel mit B bzw. WTR sonst wie MØVE

c) CØMPARE ("relop", lng, adropl, adropr)

Die Funktion meldet die Werte true oder false zurück, dabei werden lng-Halbworte für den Vergleich herangezogen, der Vergleich wird von links nach rechts durchgeführt (wie üblich!). Die Länge und die Adressen müssen gerade sei, relop ist der Vergleichsoperator, es sind zunächst (für die external-Fassung) nur die Ausdrücke EQ, NE, LE, LS, GE, GR zugelassen. COMPARE entspricht in einer laxen Schreibweise etwa dem Vergleich

adropl relop adropr

Beispiel: (wie oben)

```
.
.
.
CØMPARE ("EQ", 4, LV ZWEIGW, V)
```

d) SEARCH (adrsuchw, adrtabanf, dehn, adrmask)
Die durch adrtabanf bezeichnete Tabelle wird mit dem TDM durchsucht, die Tabelle muß also die entsprechende Gestalt haben.
Die Routine meldet die Adresse des gefundenen Wortes im Erfolgsfalle zurück, ansonsten ist die Rückmeldung negativ und ergibt invertiert die Adresse des Wortes, das zum Abbruch führte.

A

Evtl. anstehende Alarmzeiger sind gelöscht. Die Dehnung dehn ist in HW-Schritten anzugeben, adrsuchw und adrmask müssen gerade sein.

e) LOGSEARCH (adrsuchw, adrtabanf, tablng, adrmask)
wie bei SEARCH, nur wird die mit adrtabanf beschriebene Tabelle mit dem TLOG durchsucht, die Tabelle muß dementsprechend aufgebaut sein.

Die Searchroutine erscheint nur auf die Befehle TLOG bzw. TDM zugeschnitten zu sein, sie können jedoch noch durch Erweiterung um den Parameter "länge" auf allgemeine Strukturen erweitert werden.

f) SEMAPHØRE(adr)
adr ist die Adresse eines GW, das GW wird gelöscht, sein Inhalt wird als Resultat zurückgemeldet. Wird der Funktionswert abgespeichert, so wird nur das rechte HW berücksichtigt. Abgebildet wird das Intrinsic auf den Befehl BL.

g) SAVEREG (adr)
Der Inhalt der Register wird (mittels QCR) ab dem durch adr bezeichneten Speicherplatz abgelegt. Die Externalfassung ist eine parameterlose Funktion SAVEREG( ) und als Funktionswert wird die Anfangsadresse des QCR-Blockes zurückgemeldet. Ein nachfolgender Aufruf überschreibt den alten Inhalt.

h) EXECUTE (adr)
adr zeigt auf eine Speicherzelle, die einen Befehl enthält, dieser Befehl wird (durch T) ausgeführt. Bei der External-Fassung wird das B-Register zerstört.

i) ABS(expression)
Funktionswert ist der Absolutbetrag des expressions

j) SAA(), SAT()
Funktionswert ist true, falls ein Alarm anstand, die Alarme werden mit SAA bzw. SAT abgefragt und gelöscht.

k) SETSW (bool, swnumb1 , swnumb2,...)
der durch swnumbi bezeichnete Wahlschalter wird auf den durch bool angegebenen Wert (=true oder false) gesetzt.
Bei der Externalfassung sind nur ein Parameter swnumb erlaubt. (swnumb=1,...,8:Wahlschalter 1 bis 8
swnumb= 9,...,24: Zusatzwahlschalter

l) SETSL (bool, lightnumb1 , light...)
das entsprechende Merklicht wird gelöscht oder gesetzt.

m) SL (numb) und

n) SW (numb) fragen dabei Merklichter und Wahlschalter ab.
Funktionswert ist true, falls das entsprechende Bit gesetzt ist.

o) SSR (n, m, adr)
entspricht der Befehlsfolge

TCB adr, SSR n m

p) SETTK (adr, tk)
das durch adr bezeichnete Wort erhält die Typenkennung tk.

q) TK (adr, tk)
meldet den Wert true zurück, falls die durch adr bezeichnete Variable die Typenkennung TK besitzt.

r) US (adrquell, adrustab, adrziel)
Es wird das durch adrquell bezeichnete GW geholt, oktadenweise umgeschlüsselt und auf das durch adrziel bezeichnete GW abgelegt.

s) SYS (n, regbl)
regbl ist die Adresse eines Speicherbereiches der aufgebaut ist wie der QCR-Block, von dort werden die Register vor dem Aufruf des SSR mit QBR geladen und nach Aufruf des SSR dort wieder abgelegt. Der Unterprogrammordnungszähler wird auf den alten Wert restauriert. n wird Adresse des SSR-Befehls.

t) BLEI (A, P);
A = Quad Speicheradresse
P = Wert zwischen 0 und 255
entspricht BLEI p; Wert in A!0 und A!1