GRUPPE
DATENVERARBEITUNG

PDV Berichte

Projekt Prozeßlenkung mit DV-Anlagen

KFK-PDV 75

Spezifikation CIMIC/1

Mai 1976

GESELLSCHAFT FÜR KERNFORSCHUNG MBH KARLSRUHE

PDV-Berichte

Die Gesellschaft für Kernforschung mbH koordiniert und betreut im Auftrag des Bundesministers für Forschung und Technologie das im Rahmen des 2. DV-Programms der Bundesregierung geförderte Projekt Prozeßlenkung mit Datenverarbeitungsanlagen (PDV). Hierbei arbeitet sie eng mit Unternehmen der gewerblichen Wirtschaft und Einrichtungen der öffentlichen Hand zusammen. Als Projektträger gibt sie die Schriftenreihe PDV-Berichte heraus. Darin werden Entwicklungsunterlagen zur Verfügung gestellt, die einer raschen und breiteren Anwendung der Datenverarbeitung in der Prozeßlenkung dienen sollen.

Der vorliegende Bericht dokumentiert Kenntnisse und Ergebnisse, die im Projekt PDV gewonnen wurden.

Druck und Verbreitung:
Gesellschaft für Kernforschung mbH
75 Karlsruhe 1, Postfach 3640
Printed in Western-Germany

KFK-PDV 75

Projekt Prozeßlenkung mit DV-Anlagen

Forschungsbericht KFK-PDV 75

Spezifikation CIMIC/1

44 Seiten

Mai 1976

ASME BERICHT

TITEL:

Spezifikation CIMIC/1

Mai 1976

Verfasser:

Mühlhahn — Firma ESG München

unter Mitwirkung von:

Eichenauer — Firma GPP München

Holleczek
Pelz
Rössler — III. Physik. Institut der Universität Erlangen

Ghassemi
Zeh — Institut für Regelungstechnik und Prozeßautomatisierung der Universität Stuttgart

Alt — Institut für Verfahrenstechnik und Dampfkesselwesen der Universität Stuttgart

Zusammenfassung

CIMIC/1 (Compiler Internal Machine Independent Code)
ist eine symbolische, rechnerunabhängige Zwischensprache
auf Assemblerniveau, die so ausgelegt wurde, daß im ASME-
PEARL-Subset/1 geschriebene Programme in ihr darstellbar
sind. Sie wurde entwickelt, um portable Compiler zu implemen-
tieren. Ein PEARL Programm wird durch den ASME-Stufe 1 -
Compiler zunächst in CIMIC/1 übersetzt und dieses CIMIC-Pro-
gramm dann mit einem Codegenerator in die Assemblersprache
des betreffenden Zielrechners übersetzt.

In dem vorliegenden Bericht wird diese Zwischensprache
CIMIC/1 spezifiziert.

Summary

CIMIC/1 (Compiler Internal Machine Independent Code)
is a symbolic computer independent intermediate language
of assembler level, designed to represent programs, written
in ASME-PEARL-Subset/1. It was developed to implement por-
table compilers. In a first step a PEARL program is trans-
lated by the ASME-Stufe 1 - Compiler into CIMIC/1 and in a
second step this CIMIC program is translated by a code
generator into the assembly language of the relevant target
computer.

In this report the intermediate language CIMIC/1 is specified.

0. Vorbemerkungen

Die folgende CIMIC-Beschreibung umfaßt diejenigen Sprachelemente, die erforderlich sind, den ASME-PEARL-Subset/1 in CIMIC darzustellen. Sie beruht auf der CIMIC- Spezifikation vom Mai 1973, die i.w. um Systemteil, Tasking und I/O erweitert wurde. Diese Erweiterungen wurden innerhalb der ASME in mehreren CIMIC-Seminaren gemeinsam erarbeitet.

Auf die Benutzung eines Stack wurde verzichtet. Die von W.M. Waite vorgeschlagene Erweiterung von CIMIC zur optimaleren Code-Erzeugung für Rechner, die nur über indirekte Adressierung verfügen und kein Indexregister haben, wurde ebenfalls nicht aufgenommen. Derartige Optimierungen sind u.a. für die Stufe 2 vorgesehen.

Referenz:

CIMIC/1, vorläufige Spezifikation, Mai 1973

1. STRUKTUR DER CIMIC-ANWEISUNGEN

1.1 Der abstrakte Rechner

Den CIMIC-Anweisungen liegt ein abstrakter Rechner zugrunde, dessen programmierbare Teile aus

- Akkumulator
- Indexregister
- Speicher

bestehen.

Der Akkumulator dient zur Aufnahme von Operanden aller im CIMIC zulässigen Datentypen und enthält das Ergebnis der mit diesen Daten und ggfs. Operanden im Speicher ausgeführten Operationen.

Das Indexregister kann INTEGER-Größen aufnehmen. Manipulationen dieser Größen sind mit speziellen Indexregisterbefehlen möglich.

Der Speicher enthält die Programmvariablen und Programmkonstanten. Die Adressierung des Speichers erfolgt über symbolische Adressen, zu denen feste und variable Offests addiert werden können.

1.2 Das Befehlsformat.

CIMIC hat folgendes Grundformat für Anweisungen:

Instr::= operation ␣ [mode] ␣ { reference ␣␣␣ } $

Die einzelnen Elemente einer Anweisung sind dabei durch genau ein Leerzeichen getrennt. Eine Anweisung wird durch das Zeichen "$" beendet.

Die Wirkung einer CIMIC-Anweisung wird durch den Operationscode bestimmt.

operation::= { opcode/opcode,I }

Die zulässigen Operationscodes sind in Abschnitt 2 beschrieben.

Bei einigen Anweisungen (Sprunganweisung, ARGIS-Anweisung) kann dem Operationscode ",I" hinzugefügt werden. Das Zeichen "I" ist dabei die Abkürzung für "immediate". In diesem Fall wird die Operation nicht mit dem durch die Referenz bezeichneten Operanden sondern mit dessen Adresse ausgeführt.

Das Mode-Feld spezifiziert den Typ des Operanden . Operanden mit unterschiedlichem Mode können eine unterschiedliche Anzahl von Wörtern für ihre Darstellung im Zielrechner benötigen und eine bestimmte CIMIC-Anweisung kann in Abhängigkeit vom Mode zu verschiedenen Befehlsfolgen im Zielrechner führen. Die für jeden Operationscode zulässigen Modes sind in Abschnitt 2 angegeben.

Folgende Modes sind CIMIC/1 enthalten.

INT	für	INTEGER-Größen
REAL	"	REAL-Größen
BIT(length)	"	Bitketten
CHAR(length	"	Zeichenketten
DUR	"	Zeitintervalle
CLO	"	Uhrzeiten
SEMA	"	Semaphore-Variable
ADDR	"	Adressen
INSTR	"	Befehle
TASK	"	Tasks
PROC	"	Prozeduren
FORM	"	Formate
FILE	"	Files

sowie für Devices, Interrupts und Signale der jeweilige Typ (device-type, interrupt-type, signal-type). Die dafür zulässigen Schlüsselwörter sind abhängig vom Prozeßsystem und in den jeweiligen Benutzerhandbüchern beschrieben.

Das Referenz-Feld enthält Hinweise auf die Operanden in Form von symbolischen Namen. Es treten verschiedene Kategorien von Referenzen auf:

- Referenzen zu Programmkonstanten (reference 1)
- Referenzen zu Variablen und Feldern (reference 2)
- Referenzen zu Prozeduren (reference 3)
- Referenzen zu Tasks (reference 4)
- Referenzen zu Programmarken (reference 5)
- Referenzen zu Formatstrings (reference 6)
- Referenzen zu Interrupts, Signalen, Geräten, (reference 7)
- Referenzen zu Files (reference 8)

```
Reference: = { reference 1/reference 2/reference 3/
               reference 4/reference 5/reference 6/
               reference 7/reference 8/ array reference 1/
               array reference 2  / array reference 7 }
```

Programmkonstante sind entweder "normale" Konstanten oder VAL-Großen, d.h. im PEARL-Programm mit einem Namen versehene Konstanten bzw. Konstantenfelder. Infolgedessen wird zwischen 2 Arten von Referenzen zu Konstanten unterschieden.

```
reference 1::= {reference 1a/reference 1b }
```

Referenzen zu normalen Konstanten bestehen aus dem Schlüsselwort CONST, dem symbolischen Namen sowie der Angabe der Konstanten

```
reference 1a ::= CONST ␣ symbol (  ) ␣ type ␣ value
```

In CIMIC/1 werden 2 Typen von Konstantenangaben unterschieden:

type::= {A/E}

Der Wert einer Konstante vom Typ A ist unabhängig von der Zielmaschine. Die Darstellung der Konstanten erfolgt im PEARL-Format außer bei Characterstrings, die ohne äußere Apostroph dargestellt werden und bei denen ein Doppelapostroph im Inneren der Zeichenkette durch ein Apostroph ersetzt ist, sowie bei CLOCK- und DURATION-Konstanten, die in Sekunden dargestellt werden.

Beispiel:

```
ADD␣INT␣CONST␣S1 (  )␣A␣4711 $
```

Die INTEGER-Konstante 4711 wird zum Akkumulatorinhalt addiert.

```
LOAD␣BIT (7)␣CONST␣S2(  )␣A␣'1011010'B1$
```

Die Bitkette '1011010' wird in den Akkumulator geladen.

Der Wert einer Konstanten vom Typ E ist zielrechnerabhängig und mit der Adreßstruktur des Zielrechners verknüpft. Die Konstante wird dargestellt in Form eines Produktes einer ganzen Zahl mit einem MODE-Identifier. Der Wert eines solchen MODE-Identifier ist die Anzahl der Adreßeinheiten, die für die Darstellung des betreffenden MODE in dem jeweiligen Zielrechner benötigt werden.

Beispiel:

```
LOAD␣INT␣CONST␣S3 (  )␣ E␣3 * REAL $
```

Bei einem Rechner mit wortweiser Adressierung, der für die Darstellung von REAL-Größen 2 Wörter braucht, wird die Zahl 6 in den Akkumulator geladen.

Referenzen zu VAL-Größen bestehen aus einem der Schlüsselwörter

VLLOC für lokale VAL-Größen,
VLGLB für globale VAL-Größen,
VLPAR für VAL-Parameter,
VLMOD für VAL-Größen auf Modulebene,

aus dem symbolischen Namen, einem festen Offset und ggfs dem Zeichen "+"

reference 1b::= { VLLOC / VLGLB / VLPAR / VLMOD } ␣symbol([fix offset]) [+]␣␣

Das Symbol ergibt eine Basisadresse. Ein vorhandener fester Offset ist eine Konstante, die zu dieser Basisadresse addiert wird. Dieser Offset ist eine Konstante vom Typ E. Folgt ein Pluszeichen, so wird der Inhalt des Indexregisters als variabler Offset ebenfalls zur Basisadresse addiert.

Als Prozedurparameter und bei der Ein- und Ausgabe treten Referenzen auf VAL-Felder auf. Sie enthalten eines der Schlüsselwörter VLLOC, VLGLB, VLPAR, VLMOD, das Symbol, eine Feldlänge und einen Stern zur Kennzeichnung eines ganzen Feldes im Unterschied zur Referenz auf ein Feldelement:

array-reference 1 ::= { VLLOC / VLGLB / VLPAR / VLMOD } ␣symbol ([size]) * ␣␣

Referenzen zu Variablen und Feldelementen enthalten eines der Schlüsselwörter

LOCAL	für lokale Variable/Feldelemente
GLOBAL	für globale " "
PARAM	für formale Parameter
ARG	für aktuelle Parameter
MODUL	für Größen auf Modulebene

sowie ein Symbol, einen festen Offset und ggfs. das Pluszeichen

reference 2 ::= { LOCAL / GLOBAL / PARAM / ARG / MODUL } ␣ symbol ([fix offset]) [+] ␣␣

Beispiel:

```
LDX␣INT␣LOCAL␣I ( )␣␣$
MPX␣INT␣CONST␣S1 ( )␣E␣REAL $
LOAD␣REAL␣LOCAL␣ S2 (3*REAL) +␣␣$
```

Der erste Befehl lädt den Wert der INTEGER-Variablen I in das Indexregister. Im zweiten Befehl wird I multipliziert mit der Anzahl der Adreßeinheiten, die für die Darstellung einer REAL-Größe im betreffenden Rechner benötigt werden. Das Ergebnis dieser Multiplikation befindet sich im Indexregister. Der dritte Befehl lädt die REAL-Größe S2 (I+4).

Analog zur array-reference 1 gibt es Referenzen zu Variablenfeldern:

```
                                  ⎧LOCAL ⎫
array-reference 2 =               ⎪GLOBAL⎪ ␣ symbol    ( size )✳␣␣
                                  ⎪PARAM ⎪
                                  ⎩MODUL ⎭
```

Referenzen zu Prozeduren bestehen aus einem der Schlüsselwörter

MODUL	für lokale Prozeduren
GLOBAL	für globale Prozeduren
STD	für Standardprozeduren
REENT	für lokale reentrant Prozeduren
RGLOB	für globale reentrant Prozeduren

dem symbolischen Namen der Prozedur, der Anzahl der Parameter und dem Resultat-Mode bei Funktionsprozeduren

```
                  ⎧MODUL ⎫
                  ⎪GLOBAL⎪
reference 3::=    ⎨STD   ⎬symbol ([number of param])␣ ⎧      ⎧INT          ⎫ ⎫
                  ⎪REENT ⎪                           ⎪R␣    ⎪REAL         ⎪ ⎪
                  ⎩RGLOB ⎭                           ⎪      ⎪BIT(length)  ⎪ ⎪
                                                     ⎨      ⎨CHAR(length) ⎬ ⎬
                                                     ⎪      ⎪CLO          ⎪ ⎪
                                                     ⎩      ⎩DUR          ⎭ ⎭
                                                        ␣
```

Referenzen zu Tasks enthalten eines der Schlüsselwörter

MODUL	für lokale Tasks
GLOBAL	für globale Tasks
RESD	für residente lokale Tasks
RS LB	für residente globale Tasks

das Symbol und ggfs die Priorität, falls eine Priorität im zugehörigen PEARL-Statement angegeben war.

reference 4::= { MODUL / GLOBAL / RESD / RSGLB } ␣symbol ([priority]) ␣␣

Referenzen zu Programmarken bestehen aus dem Schlüsselwort CODE und dem symbolischen Namen der Marke:

reference 5::= CODE␣symbol ([fix offset]) [+] ␣␣

Referenzen zu Formatstrings enthalten das Schlüsselwort VLLOC und den symbolischen Namen des Formatstrings:

reference 6 ::= VLLOC ␣ symbol ( ) ␣␣

Die Referenz 7 umfaßt Referenzen zu Interrupts, Signalen und Geräten. Sie enthalten die Richtung des Informationsflusses (I für Eingabe, O für Ausgabe, IO für Ein- und Ausgabe), den symbolischen Namen sowie bei Device-Arrays ggfs einen "fix offset" und bei einem "variable offset" das Zeichen "+".

reference 7 ::= { I / O / IO } symbol ([fix offset]) [+] ␣␣

entsprechend bei Referenzen zu Device-Arrays:

array reference 7 :: = { I / O / O } ␣symbol (size) ␣␣

Im Gegensatz zu den bisherigen Offsets ist der Referenz 7-Offset nur eine Integer-Größe (d.h. kein Ausdruck der Form integer mode). Das 3. Element eines Device-Arrays hat also die Bezeichnung symbol (2).

Referenzen zu Files enthalten das Schlüsselwort VLLOC und den symbolischen Namen

reference 8 ::= VLLOC␣symbol ( )␣␣

Bei einigen CIMIC-Anweisungen, die als Operanden die Inhalte der Register bearbeiten, ist kein Referenzfeld vorhanden.

2. CIMIC-ANWEISUNGEN

2.1 Transferbefehle

Transferbefehle dienen zum Laden und Abspeichern von Akkumulator und Indexregister.

LOAD : Laden des Operanden in den Akkumulator

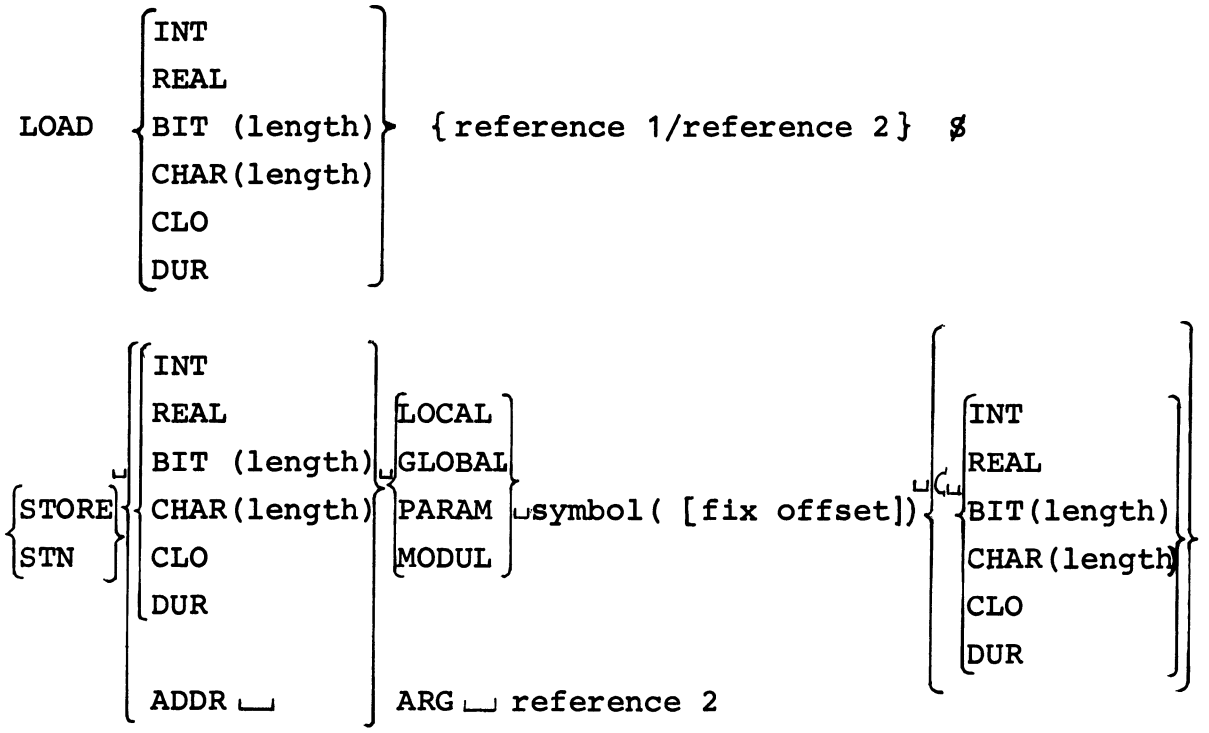
STORE : Abspeichern des Akkumulators in die angegebene Operandenadresse

STN : wie STORE, der Inhalt des Akkumulators muß jedoch erhalten bleiben.

LDX : Laden des Operanden in das Indexregister

STX : Abspeichern des Indexregisters in die angegebene Operandenadresse.

Bei den Speicherbefehlen STORE und STN kann nach der Referenz auf den Operanden noch eine 2. Modeangabe stehen, die den Mode des Akkumulators angibt, sofern dieser verschieden ist vom Mode des Operanden. Damit können entsprechende Konvertierungen vorgenommen werden.



LDX INT {reference 1/reference 2} ¢

STX INT reference 2 ¢

Beispiel:

```
LOAD  DUR  CONST  S1(   )  A 100.8SEC¢
STORE DUR  LOCAL  S2(   )   ¢
LOAD  REAL CONST  S3(   )  A 10.9¢
STORE REAI LOCAL  S4(3*REAL)   ¢
```

2.2 Arithmetische Befehle

2.2.1 Zahlarithmetik

In CIMIC/1 sind folgende Operationen mit REAL- und INTEGER-Größen zugelassen:

- ABS: Bildung des Absolutwertes des Akkumulators
- NEG: Negierung des Akkumulators
- ADD: Addition des Operanden zum Inhalt des Akkumulators
- SUB: Subtraktion des Operanden vom Inhalt des Akkumulators
- MPY: Multiplikation des Operanden mit dem Inhalt des Akkumulators
- DIV: Division des Inhalts des Akkumulators durch den Operanden
- EXP: Exponentiation - der Operand ist der Exponent
- ADX: Addition des Operanden zum Indexregister
- SBX: Subtraktion der Operanden vom Indexregister
- MPX: Multiplikation des Operanden mit dem Indexregister
- ADDAX: Addition von Akkumulator und Indexregister, das Ergebnis befindet sich im Indexregister

Nach Ausführung der jeweiligen Operation befindet sich das Ergebnis im Akkumulator (außer bei ADDAX und den Indexbefehlen) der Operand bleibt unverändert.

{ABS / NEG} {INT / REAL} ␣␣␣␣

{ADD / SUB / MPY / DIV / EXP} ␣ {INT / REAL} ␣ { reference 1/reference 2 } ⊄

{ADX / SBX / MPX} ␣ INT ␣ { reference 1/ reference 2 } ⊄

ADDAX ␣INT ␣␣␣␣⊄

Beispiel:

Berechnung von A + 7*C, wobei A und C REAL-Größen sind.

```
LOAD␣ REAL ␣LOCAL ␣C(  )␣␣$
MPY ␣ REAL ␣CONST ␣S1(  )␣A ␣7 $
ADD ␣ REAL ␣LOCAL ␣A(  )␣␣$
```

2.2.2 Bitkettenoperationen

Die CIMIC-Bitkettenoperationen sind:

NOT: Inversion der Bitkette im Akkumulator

AND: Verknüpfung von Akkumulator und Operand durch logisches UND (Konjunktion)

OR: Verknüpfung von Akkumulator und Operand durch logisches ODER (Disjunktion)

EXOR: Verknüpfung von Akkumulator und Operand durch exclusives ODER.

CONC: Verkettung der Bitkette im Akkumulator mit dem Operanden.

LBIT: Der Operand a ist eine INTEGER-Größe mit $1 \leq a \leq \pi$, wo π die Länge der Bitkette im Akkumulator ist. Das Ergebnis ist das a-te Bit des Akkumulatorinhalts von links.

RBIT: Analog LBIT. Das Ergebnis ist das a-te Bit des Akkumulatorinhalts von rechts.

SHIFT: Schift des Akkumulators nach rechts, wenn der Operand negativ ist bez. links, wenn der Operand positiv ist.

Nach Ausführung der obigen Operationen befindet sich das jeweilige Ergebnis im Akkumulator, ein Operand bleibt unverändert.

```
NOT ␣ BIT(length)␣␣␣$

{AND }
{OR  } BIT(length)␣{reference 1/reference 2}  $
{EXOR}
{CONC}

{LIBIT}
{RBIT } ␣INT␣{reference 1/ reference 2}  $
{SHIFT}
```

2.2.3 Zeichenkettenoperationen

Die CIMIC-Anweisung CONC erlaubt die Verkettung von Zeichenketten:

```
CONC ␣ CHAR(length)␣{reference 1/reference 2 }  $
```

2.2.4 Clock- und Duration-Operation

In CIMIC sind Addition und Subtraktion von Clock und Duration- Operanden zu Clock- und Duration-Größen im Akkumulator zulässig, und zwar:

```
CLO - CLO, das Ergebnis ist DUR
CLO ± DUR,   "      "     "  CLO
DUR + CLO,   "      "     "  CLO
DUR ± DUR,   "      "     "  DUR
INT * DUR,   "      "     "  DUR
DUR * INT,   "      "     "  DUR
```

```
{ADD} ␣ {DUR} ␣{reference 1/ reference 2}  $
{SUB}   {CLO}

MPY ␣ DUR{␣reference 1/ reference 2}  $
```

Beispiel:

```
LOAD ␣ CLO ␣ CONST ␣ S1(  ) ␣ A ␣ 12:0:0 ∅
ADD ␣  DUR ␣ CONST ␣ S2(  ) ␣ A ␣ 30 MIN  ∅
```

2.3 Vergleichsbefehle

Ein Vergleichsbefehl vergleicht den Inhalt des Akkumulators mit dem im Vergleichsbefehl angegebenen Operanden. Entsprechend dem Ergebnis des Vergleichs werden die zutreffenden der folgenden Bedingungen "gleich", "ungleich", "größer gleich", "größer als", "kleiner gleich" und "kleiner als" gesetzt. Diese Bedingungen können durch einen nachfolgenden bedingten Sprungbefehl abgefragt werden. Die Bedingungen werden gelöscht durch den ersten Befehl, der verschieden von einem bedingten Sprungbefehl ist.

```
CMP␣ {INT          }␣{reference 1/ reference 2}  ∅
     {REAL         }
     {BIT(length)  }
     {CHAR(length  }
     {CLO          }
     {DUR          }
```

2.4. Sprungbefehle

Folgende Operationen sind zulässig.

JMP: unbedingter Sprung
JEQ: Sprung, wenn Bedingung "gleich" gesetzt ist
JNE: Sprung, wenn Bedingung "ungleich" gesetzt ist
JGE: Sprung, wenn Bedingung "größer gleich" gesetzt ist
JGT: Sprung, wenn Bedingung "größer als" gesetzt ist
JLE: Sprung, wenn Bedingung "kleiner gleich" gesetzt ist
JLT: Sprung, wenn Bedingung "kleiner als" gesetzt ist

Der Operand des Sprungsbefehls ergibt die Sprungadresse. Für die Darstellung von Sprungbefehlen gibt es in CIMIC/1 zwei Möglichkeiten.

opcode, I ␣ INSTR␣CODE␣symbol( )␣␣$
wobei symbol eine durch LOC definierte Marke ist und direkt die Sprungadresse darstellt, oder
opcode ADDR CODE symbol ([fix offset]) [+]␣␣$
wobei symbol ([fix offset]) [+] die Referenz zu einem Speicherplatz ist, der die Sprungadresse enthält.

Beispiel 1:

```
LOC␣␣␣M99 (  )␣␣$
      .
      .
      .
      .
      .
JMP,I ␣ INSTR ␣ CODE␣ M99(  )␣␣$
```

Es wird zur Marke M99 gesprungen

Beispiel 2:

```
SPACE␣ ADDR ␣ CODE␣ S1(  ) ␣ A ␣ M99 $
LOC␣␣␣M99(  )␣␣$
JMP ␣ ADDR␣CODE ␣ S1(  )␣␣ $
```

Es wird ebenfalls zur Marke M99 gesprungen

2.5 Deklarationen

2.5.1 Marken

Zur Definition von Programmarken dient der Pseudo-Befehl LOC:

```
LOC␣␣␣symbol(  )␣␣$
```

In einem CIMIC-Programm dürfen mehrere Markendefinitionen unmittelbar aufeinander folgen. Die verschiedenen Symbole bezeichnen dann die gleiche Adresse.

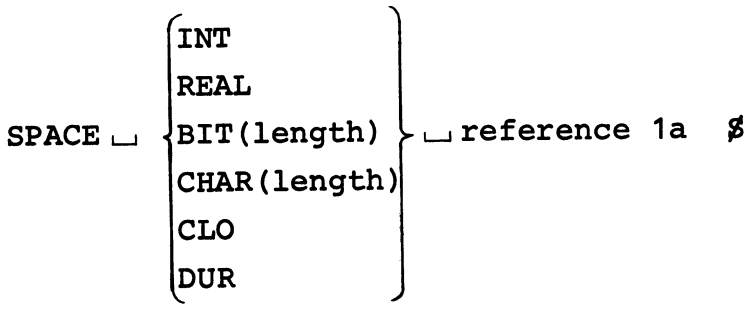
Beispiel:

```
LOC ␣␣␣M1  (  )␣␣$
LOC ␣␣␣M2 (   )␣␣$
LOAD␣ INT␣LOCAL␣S1(  )␣␣$

JMP,I␣INSTR␣CODE ␣ M1(  )␣␣ $

JMP,I␣INSTR␣CODE␣M2  (  )␣␣$
```

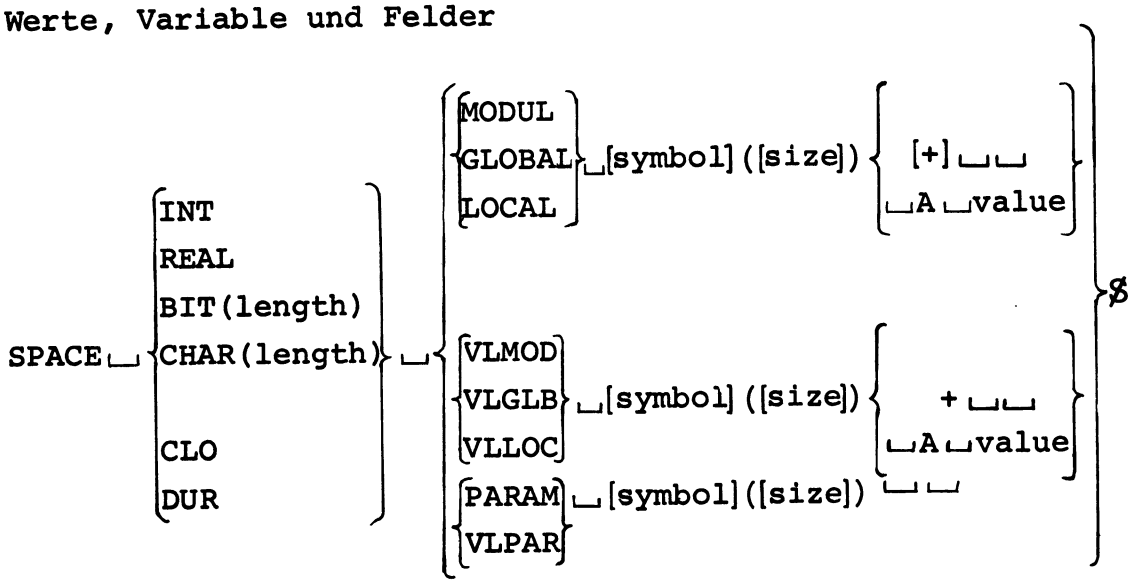
Bei beiden Sprungbefehlen wird das Programm mit der LOAD-Anweisung fortgesetzt.

2.5.2 Konstante

Zur Reservierung von Speicherplätzen für Konstante dient die SPACE-Anweidung mit einer Referenz 1a



2.5.3 Werte, Variable und Felder



Durch die SPACE-Anweisung werden für die durch "size" gegebene Anzahl von Elementen des angegebenen Mode Speicherplätze reserviert. Wenn keine "size"-Angabe vorhanden ist, handelt es sich um eine einfache Variable bzw. Wert. Eine nachfolgende Konstante bedeutet, daß die Größe initialisiert werden soll. Bei einem Feld werden dann alle Feldelemente mit der gleichen Konstante initialisiert. Sollen verschiedene Feldelemente mit unterschiedlichen Anfangswerten belegt werden, muß die SPACE-Anweisung das Zeichen "+" enthalten. Es folgen dann weitere SPACE-Anweisungen ohne "symbol", die den einzelnen Feldelementen Anfangswerte zuordnen.

Bei Werten (Kategorie VLMOD, VLGLB, VLLOC) muß eine Konstante angegeben werden.

Beispiel 1:

```
SPACE␣INT␣LOCAL␣S1(  )␣␣$
```

Deklaration einer einfachen Variablen ohne Initialisierung.

Beispiel 2:

```
SPACE ␣ REAL ␣ LOCAL ␣ S2(10*REAL)␣␣$
```

Deklaration eines Feldes von REAL-Größen der Länge 10.

Beispiel 3:

```
SPACE ␣ DUR ␣ LOCAL ␣ S3(  )␣ A ␣10MIN $
```

Deklaration einer Duration-Variablen mit Initialisierung

Beispiel 4:

```
SPACE␣BIT(10)␣LOCAL␣S4(5*BIT(10)␣A␣'1001001011'B1$
```

Deklaration eines Feldes von Bitketten. Alle Feldelemente werden mit der gleichen Bitkette vorbesetzt.

Beispiel 5:

```
SPACE␣INT␣MODUL␣S5(6*INT)+␣␣$
SPACE␣INT␣MODUL␣(2*INT)␣A␣1  $
SPACE␣INT␣MODUL␣(1*INT)␣␣$
SPACE␣INT␣MODUL␣(1*INT)␣A␣3  $
SPANE␣INT␣MODUL␣(1*INT)␣␣$
SPACE␣INT␣MODUL␣(1*INT)␣A␣2  $
```

Deklaration eines INT-Feldes der Länge 6. Die ersten 2 Feldelemente werden mit der Zahl 1 vorbesetzt, das 3. Element wird nicht initialisiert, das 4. Element wird mit der Zahl 3 vorbesetzt, das 5. Element wird nicht initialisiert und das 6. Element bekommt den Anfangswert 2.

2.5.4 Adreßfelder

Adreßfelder treten beim Prozeduraufruf in Form von Adressen der aktuellen Parameter (Kategorie ARG) und bei Markenvariablen als Feld von Programmlabel (Kategorie CODE) auf.

```
SPACE  ADDR {ARG }␣[symbol]([size]) { +␣␣      } $
            {CODE}                  {␣A␣value}
```

Beispiel:

```
SPACE␣ADDR␣ARG␣L1(2*ADDR)+␣␣$
SPACE␣ADDR␣ARG␣(1*ADDR)␣A␣S1 $
SPACE␣ADDR␣ARG␣(1*ADDR)␣A␣S2 $
```

Es wird ein Adreßfeld der Länge 2 deklariert, das die Adressen von aktuellen Parametern enthält. Die Adresse des 1. aktuellen Parameters ist S1, die des 2. aktuellen Parameters S2.

2.5.5 Semaphore-Variable

```
SPACE␣SEMA␣{GLOBAL}␣symbol(  ){␣A␣value}  $
           {MODUL }                {␣␣  }
```

Semaphore-Variable können mit einer Integer-Konstanten initialisiert werden.

2.5.6 Formate

Ein PEARL-Formatstring wird in CIMIC dargestellt durch ein Feld vom Mode FORM, wobei jedes Feldelement initialisiert wird mit einem CIMIC-Formatelement.

```
CIMIC-format-element::={[(integer)]   ( | ) |
                        [(integer)] c-format-element|
                        [(integer)] g-format-element}

SPACE␣FORM␣VLLOC␣symbol (number of CIMIC-format-elements
                         *FORM)+␣␣$
SPACE FORM VLLOC (FORM) A CIMIC-format-element $
  :
```

Beispiel:

Die PEARL-Formatanweisung

```
DRUCK: FORMAT (PAGE(3),F(6),X(5),(4)(F(7),X(2),A(5)))
```

wird in CIMIC folgendermaßen dargestellt:

```
SPACE␣FORM␣VLLOC␣DRUCK(10—FORM)+␣␣$
SPACE␣FORM␣VLLOC␣(FORM)␣A␣(  $
        "                   A␣PAGE(3)  $
        "                   A␣F(6)  $
        "                   A␣X(5)  $
        "                   A␣(4)(  $
        "                   A␣F(7)  $
        "                   A␣X(2)  $
        "                   A␣A(5)  $
        "                   A␣)  $
        "                   A␣)  $
```

2.5.7 Systemteil

Der Pearl-Systemteil wird in CIMIC durch eine Folge von SPACE-Anweisungen dargestellt. Für jede Endstelle existiert eine SPACE-Anweisung, die den Typ der Endstelle, die Richtung des Informationsflusses, den symbolischen Namen, und eine Beschreibung des Anschlußweges enthält.

```
SPACE␣ {interrupt-type} ␣ {I} ␣symbol(  )␣␣connection $
       {signal-type   }    {O}
       {device-type   }    {O}
```

```
connection ::={✳integer....}    {✳integer, integer}
```

bzw. bei Arrays:

```
SPACE␣ {interrupt-type} ␣ { I} ␣symbol(number of devices)+␣␣$
       {signal-type   }   { O}
       {device-type   }   {IO}

SPACE␣ {interrupt-type} ␣ { I} ␣(1)␣␣connection $
       {signal-type   }   { O}
       {device-type   }   {IO}
   ⋮
```

Die zulässigen Endstellentypen sind abhängig von der Prozeßkonfiguration und in den jeweiligen Benutzerhandbüchern beschrieben.

2.5.8 Filedeklarationen

Filedeklarationen haben die Form

SPACE␣FILE␣reference 8 ¢

2.6 Prozeduren und Tasks

2.6.1 Prozedurkörper

Die erste Anweisung des Prozedurbody ist eine BEGIN-Anweisung:

BEGIN ␣ PROC ␣ reference 3 ¢

Auf die BEGIN-Anweisung folgen SPACE-Anweisungen für die formalen Parameter (Kategorie PARAM bzw. VLPAR bei Wertübergabe) und danach für die prozedurinternen Größen (Kategorie LOCAL).

Die Liste der Deklarationen wird abgeschlossen durch eine LINK-Anweisung:

LINK␣PROC␣reference 3 ¢

Sie stellt das Linkage zu dem aufrufenden Programm her (Parameterübertragung, Rücksprung). Ihre Wirkung hängt von der Unterprogrammorganisation des jeweiligen Zielrechners ab.

Es folgen die ausführbaren Anweisungen des Prozedurkörpers. Zugriffe zu den formalen Parametern sind dabei von der Kategorie PARAM bzw. VLPAR.

Die RETURN-Anweisung ist der aktuelle Rücksprung aus einer Prozedur:

RETURN␣PROC␣reference 3 $

Ein RESULT-Parameter bei einer Funktionsprozedur wird im Akkumulator dem aufrufenden Programm übergeben

Der Prozedurkörper wird abgeschlossen durch eine End-Anweisung:

END ␣ PROC␣reference 3 $

Beispiel:

Das folgende Beispiel zeigt eine Prozedur, die nach einem vereinfachten Algorithmus die Quadratwurzel einer positiven reellen Zahl berechnet.

```
BEGIN␣ PROC ␣ LOCAL ␣ SQRT(1)␣R␣REAL $
SPACE␣ REAL ␣ VLPAR ␣ S1(  )␣␣$
SPACE␣ REAL ␣ LOCAL ␣ S2(  )␣␣$
SPACE␣ REAL ␣ LOCAL ␣ S4(  )␣␣$
LINK ␣ PROC ␣ LOCAL ␣ SQRT(1)␣R ␣REAL $
LOAD ␣ REAL ␣ VLPAR ␣ S1(  )␣␣$
STORE ␣ REAL ␣ LOCAL ␣ S2(  )␣␣$
LOC␣␣␣M1(  )␣␣$
LOAD ␣ REAL ␣ VLPAR ␣ S1(  )␣␣$
DIV ␣ REAL␣LOCAL␣S2(  )␣␣$
ADD ␣ REAL␣LOCAL␣S2(  )␣␣$
DIV ␣ REAL␣CONST␣S3(  )␣A␣2 $
STORE␣REAL␣LOCAL␣S4(  )␣␣$
CMP␣REAL␣LOCAL␣S2(  )␣␣$
JNE,I␣INSTR␣CODE␣M2(  )␣␣$
LOAD␣REAL␣LOCAL␣S2(  )␣␣$
RETURN␣PROC␣LOCAL␣SQRT(1)␣R␣REAL $
LOC␣␣␣M2(  )␣␣$
LOAD␣ REAL␣ LOCAL␣S4(  )␣␣$
STORE␣REAL␣ OCAL␣S2 ␣␣ $
JNE,I␣INSTR␣CODE␣M1(  )␣␣$
END␣PROC␣LOCAL␣ SQRT(1)␣ R␣REAL $
```

2.6.2 Prozeduraufruf

Der Aufruf einer Prozedur wird durch eine Folge von CIMIC-Anweisungen dargestellt.

Die Folge beginnt mit der CALL-Anweisung:

CALL␣PROC␣reference 3 $

Die aktuellen Parameter werden mit den Anweisungen ARGIS, ARGIS,I bzw. ARGIS,S übergeben.

```
[ARGIS  ]
{ARGIS,I} ␣ [INT      ]
[ARGIS,S]   [REAL     ]
            [CLO      ] {reference 1/array-reference 1/
            [DUR      ]
            [BIT(length)] reference 2/array-reference 2} $
            [CHAR(length)]
```

Welche dieser Anweisungen für die Parameterübergabe genommen wird, hängt vom aktuellen Parameter (Variable, Konstante, Ausdruck, Feldelement) und vom formalen Parameter (PARAM oder VLPAR) ab (siehe Abbildung).

Die Parameterübergabe wird abgeschlossen durch die Anweisung CEND

CEND␣ PROC␣reference 3 $

aktueller Parameter	formaler Parameter: PARAM	formaler Parameter: VLPAR
Variable	ARGIS,I mode reference 2 $	ARGIS mode reference 2 $
Konstante und VAL-Größe	—	ARGIS mode reference 1 $
Ausdruck	—	ARGIS mode reference 2 $ Bemerkung: Die Berechnung des Parameters wird vor dem Prozeduraufruf ausgeführt und der Wert in einer Hilfszelle abgelegt. Die "reference 2" bezieht sich auf diese Hilfszelle.
VAL-Feldelement	—	ARGIS mode reference 2 $ Bemerkung: wie Ausdruck
variable Feldelement	ARGIS, S mode reference 2 $ Bemerkung: Die Berechnung der Adresse des Feldelementes wird vor dem Prozedur-Aufruf ausgeführt und die Adresse in einer Hilfszelle abgelegt. Die "reference 2" bezieht sich auf diese Hilfszelle, ihr Mode und Kategorie die des Feldelements.	ARGIS mode reference 2 $ Bemerkung: wie Ausdruck
VAL-Feld	—	ARGIS mode array-reference 1 $
variabl. Feld	ARGIS,I mode array-reference 2 $	ARGIS mode array-reference 2 $

2.6.3 Taskkörper

Die erste Anweisung des Taskbody ist die BEGIN-Anweisung:

BEGIN ␣ TASK ␣ reference 4 ¢

Auf die BEGIN-Anweisung folgen SPACE-Anweisungen für die lokalen Größen. Die Liste der Deklarationen wird abgeschlossen durch eine LINK-Anweisung:

LINK ␣ TASK ␣ reference 4 ¢

Der Taskbody wird abgeschlossen durch die END-Anweisung:

END ␣ TASK ␣ reference 4 ¢

2.7 Spezifikationen

2.7.1 Spezifikation von Werten

SPEC ␣ { INT / REAL / BIT(length) / CHAR(length) / CLO / DUR } ␣ VLGLB ␣ symbol ([size]) ␣ ␣ ¢

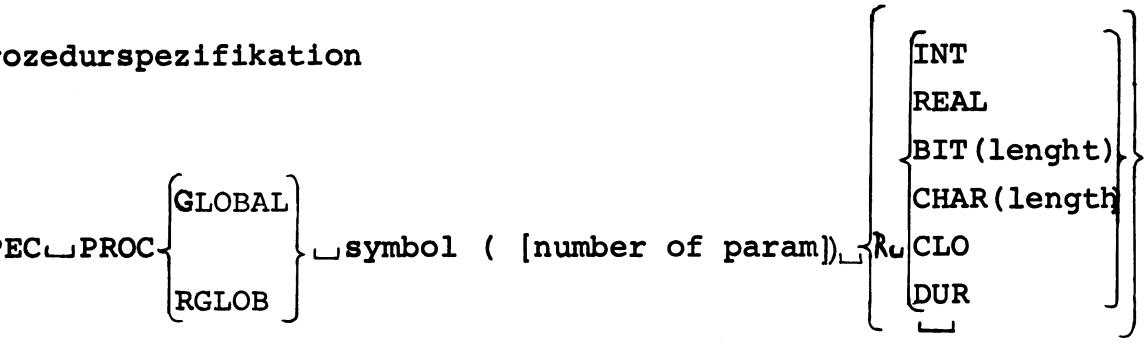
Die Spezifikation von Werten besteht aus dem Operationscode SPEC, dem Mode, dem Schlüsselwort VLGLB, dem Symbol und bei VAL-Felder der Feldlänge.

2.7.2 Taskspezifikationen

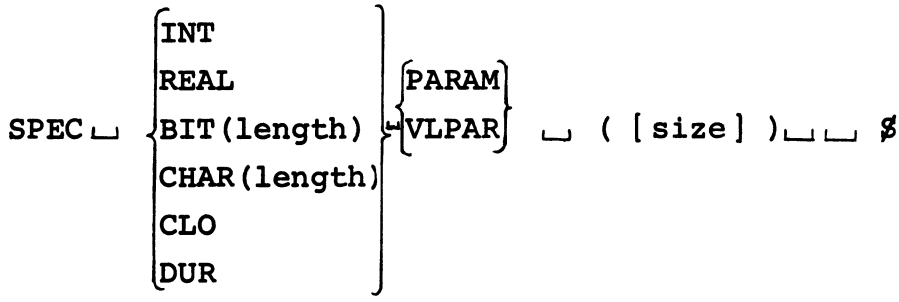
SPEC ␣ TASK ␣ { GLOBAL / RSGLB } ␣ symbol ( ) ␣ ␣ ¢

Eine Taskspezifikation enthält den Operationscode SPEC, den Mode TASK, das Schlüsselwort GLOBAL bzw. RSGLB für globale bzw. residente globale Tasks und den Tasknamen.

2.7.3 Prozedurspezifikation



Eine Prozedurspezifikation enthält in der ersten Anweisung neben den Schlüsselwörtern SPEC, PROC, GLOBAL bzw. RGLOB (für globale bzw. globale reentrant Prozeduren) den Prozedurnamen, die Anzahl der Parameter und ggfs den Resultat-Mode. Für jeden formalen Parameter folgen Spezifikationsanweisungen der Form



wobei "size" bei Feldern deren Länge angibt.

2.8 Realtime - Features und Ein- und Ausgabe

Die meisten PEARL-Anweisungen für das Tasking, die Interruptbehandlung sowie für die Ein- und Ausgabe enthalten eine Anzahl von Einzelparametern wie Schedule, Priorität, Format, Positionierung usw. Infolgedessen besteht die Darstellung dieser PEARL-Anweisungen in CIMIC in vielen Fällen aus einer Folge von mehreren CIMIC-Anweisungen, d.h. jeder Einzelparameter wird i.a. in eine CIMIC-Anweisung übersetzt. Der Operationscode dieser CIMIC-Anweisung wurde in Anlehnung an die PEARL-Schlüsselwörter gebildet.

In den CIMIC-Anweisungsfolgen treten Referenzen auf, bei denen ein variabler Offset zulässig ist (reference 1, reference 2, reference 7). Die erforderliche Berechnung der Indizes wird

- bei Device-Referenzen (reference 7) unmittelbar vor der Device-Referenz angeführt. Da Device-Referenzen immer am Anfang einer CIMIC-Anweisungsfolge steht (Ausnahme: bei CREATE und OPEN, dort handelt es sich jedoch um ein Standardgerät, das nicht als Element eines Device-Array vorkommt), steht die "index-calculation-sequence" unmittelbar vor der CIMIC-Folge.

- bei Referenz 1 und Referenz 2 in Lese- und Schreibanweisungen wie CREAD, CWRITE, UREAD usw. direkt vor dieser CIMIC-Anweisung ausgeführt.

- bei den übrigen Referenzen 1 und 2 (z.B. AT-Anweisung bei ACTIVATE) vorgezogen, d.h. der Operand wird in eine Hilfszelle abgespeichert und diese Hilfszelle in der betreffenden CIMIC-Anweisung angegeben.

2.8.1 Tasking-Operationen

- Aktivieren einer Task (ACTIVATE):

```
ACTV ␣ TASK ␣ reference 4  $
SCHED ␣ ␣ ␣ ␣ ␣ INTEGER $

[ { AT ␣ CLO                                          }
  { AFTER ␣ DUR } ␣ {reference 1/ reference 2}        }  $  ]
  { WHEN ␣ interrupt-type ␣ reference 7               }

[ ALL ␣ DUR ␣ {reference 1/ reference 2}  $ ]

[ { UNTIL ␣ CLO } ␣ {reference 1/ reference 2}   $ ]
  { DRNG ␣ DUR  }

AEND ␣ ␣ ␣ ␣ ␣  $
```

- Unterbrechen einer Task (SUSPEND)

```
SUSP ␣ TASK ␣ reference 4  $
```

- Beenden einer Task (TERMINATE):

```
TERM ␣ TASK ␣ reference 4  $
```

- Aufheben des Task-Schedules (PREVENT):

```
PREV ␣ TASK ␣ reference 4  $
```

- Fortsetzen einer Task (CONTINUE):

```
CONT ␣ TASK ␣ reference 4  $
SCHED␣␣␣␣␣integer $
[[{AFTER ␣ DUR} ␣ {reference 1/ reference 2}]
 {AT  ␣  CLO}                                 } $ ]
 [WHEN   interrupt-type ␣ reference 7         ]

CTEND ␣␣␣␣␣ $
```

- Unterbrechen und Fortsetzen einer Task entsp. Schedule (RESUME):

```
RESM ␣ TASK ␣ reference 4 $
SCHED ␣␣␣␣␣integer $
[[{AFTER ␣ DUR} ␣ {reference 1/ reference 2}]
 {AT  ␣  CLO}                                 }   $ ]
 [WHEN ␣ interrupt-type ␣ reference 7         ]

REND ␣␣␣␣␣ $
```

Die bei einigen Tasking-Anweisungsfolgen auftretende Pseudo-Anweisung SCHED enthält den Schedule-Typ.
Der Schedule-Typ ist folgendermaßen verschlüsselt:

Schedule:	Typ:
Kein Schedule	Leerstring
AT	1
AT ALL	2
AT ALL UNTIL	3
AT ALL DURING	4
ALL	5
ALL UNTIL	6
ALL DURING	7

AFTER	8
AFTER ALL	9
AFTER ALL UNTIL	10
AFTER ALL DURING	11
WHEN	12
WHEN ALL	13
WHEN ALL UNTIL	14
AFTER ALL DURING	15

2.8.2 Anweisungen für die Interrupt- und Signalbehandlung

- Sperren eines Interrupts (DISABLE):

  DSAB ␣ interrupt-type ␣ reference 7 $

- Zulassen eines gesperrten Interrupts (ENABLE):

  ENAB ␣ interrupt-type ␣ reference 7 $

- Simulation eines Interrupts (TRIGGER):

  TRIG ␣ interrupt-type ␣ reference 7 $

- Anmelden einer Fortsetzungsadresse beim Auftreten eines spezifizierten Ereignisses (ON-Anweisung):

Die CIMIC-Befehlsfolge besteht aus den Anweisungen ONC und ONL, von denen die erste eine Bedingung spezifiziert (Signal), die zweite die Fortsetzungsadresse ergibt. Sollten beim Eintreffen des Ereignisses die Systemmaßnahmen ausgeführt werden, folgt statt dessen die Anweisung SYSTEM.

```
[index - calculation - sequence]          /*Indexrechnung
 ONC ␣ signal-type  reference 7 $           für reference 5*/
{ONL,I␣INSTR␣reference 5  $ }
{ONL ␣ ADDR ␣ reference 5  $ }
{SYSTEM ␣␣␣␣␣␣ $ }
```

Die vorangehende Indexrechnung bezieht sich auf die Referenz 5.

2.8.3 Standard - E/A

- Standard - Eingabe

```
GET ␣ device-type ␣ reference 7 $
FORMAT ␣ FORM ␣ reference 6  $
[{[index-calculation-sequence]

                 {INT                  }
                 {REAL                 }
                 {BIT(length)          }
CREAD␣           {CHAR(length)         } ␣{reference 2/ array-reference 2 } $
                 {DUR                  }
                 {CLO                  }

...}]

GEND␣␣␣␣␣␣ $
```

- Standard - Ausgabe

```
PUT ␣ device-type ␣ reference 7 $
FORMAT ␣ FORM ␣ reference 6   $
[{[index-calculation-sequence]

                 {INT                  }
                 {REAL                 }
CWRITE␣          {BIT(length)          } ␣ {reference 1/ array-reference 1/
                 {CHAR(length)         }
                 {DUR                  }     REFERENCE 2/ array-reference 2 } $
                 {CLO                  }
...}]
PEND␣␣␣␣␣␣ $
```

2.8.4 Prozeß - E/A

- Eingabe:

```
[index-calculation-sequence]
 IMOVE␣device-type␣{reference 7 /array reference 7}$
 OPT ␣␣␣␣␣␣       ( option-list  ) $
[index-calculation-sequence]
 UREAD␣ ⎧BIT(length)⎫ ␣{reference 2/ array-reference 2} $
        ⎩    INT   ⎭
```

- Ausgabe:

```
[index-calculation-sequence]
 OMOVE ␣ device-type ␣{reference 7 /array reference 7}$
 OPT␣␣␣␣␣␣ (  option-list  )  $
[index-calculation-sequence]
 UWRITE ␣⎧BIT(length)⎫ ␣{reference 1/ array-reference 1/
         ⎩    INT   ⎭    reference 2/ array-reference 2} $
```

- MOVE ohne Datentransfer

```
[index-calculation-sequence]
 SPMOV ␣ device-type␣{reference 7 /array reference 7}$
 OPT ␣␣␣␣␣␣ (  option-list  )  $
```

2.8.5 File-Handlung

- CREATE,OPEN

```
{CREATE}␣FILE␣reference 8 ß
{OPEN  }

TITLE␣CHAR(4)␣{reference 1/ reference 2/␣( )␣␣} ß

{INPUT  }
{OUTPUT } ␣device-type␣reference 7 ß
{UPDATE }

{SEQ} ␣␣␣␣␣[ integer] ß
{DIR}

{ALPHA}
{INT  }␣␣␣␣␣[integer] ß
{REAL }
```

- GET, PUT

```
GET␣FILE␣reference 8 ß
{POS}␣INT␣ {reference 1/ reference 2} ß
{ADV}
FORMAT␣FORM␣reference 6 ß
[{[index-calculation-sequence]

                  {     INT     }
                  {     REAL    }
CREAD␣            {BIT(length)  }␣{reference 2/ array-reference 2} ß
                  {CHAR(length) }
                  {     DUR     }
                  {     CLO     }

...}]
GEND␣␣␣␣␣ß
```

```
PUT␣FILE␣reference 8 ∅
{ADV} ␣ INT␣ {reference 1/ reference 2}  ∅
{POS}
FORMAT␣FORM␣reference 6 ∅

 [{[index-calculation-sequence]

                 {INT                }
                 {REAL               }
CWRITE␣{BIT(length)}␣{reference 1/reference 2/array-reference 1/
                 {CHAR(length)       }
                 {CLO                                array-reference 2}  ∅
                 {DUR                }

...}]
PEND␣␣␣␣␣ ∅
```

- LOCK, UNLOCK, CLOSE, DELETE:

```
        {LOCK  }
        {UNLOCK} ␣ FILE␣ reference 8 ∅
        {CLOSE }
        {DELETE}
```

- MOVE auf Files

```
  FIMOV␣FILE␣reference 8 ∅
 {POS} ␣INT␣{reference 1/ reference 2}  ∅
 {ADV}

 {[index-calculation-sequence]
```

```
                  [INT                ]
                  [REAL               ]
FREAD ␣           [BIT(length)        ] ␣{reference 2/array-reference 2} $
                  [CHAR(length)       ]
                  [CLO                ]
                  [DUR                ]

...}
FIEND ␣␣␣␣␣ $

FOMOV␣FILE ␣ reference 8 $
[POS] ␣INT␣ {reference 1/reference 2} $
[ADV]

 {[index-calculation-sequence]

                  [INT                ]
                  [REAL               ]
 FWRITE ␣         [BIT(length)        ]␣{reference 1/reference 2/
                  [CHAR(length)       ]   array-reference 1/array-reference 2}
                  [CLO                ]
                  [DUR                ]

...}
FOEND␣␣␣␣␣ $
```

2.8.6 Graphische E/A

- Eingabe

```
    SEE␣device-type␣reference 7 $
    FORMAT␣FORM␣reference 6 $
    [{[index-calculation-sequence]
    GREAD ␣ [INT ] ␣{reference 2/array-reference 2 }  $
            [REAL]

   ...}]
    SEND␣␣␣␣␣ $
```

- Ausgabe

```
DRAW␣device-type␣reference 7 ∅
FORMAT␣FORM␣reference 6 ∅
[{[index-calculation-sequence]
GWRITE␣ {INT }␣{reference 1/array-reference 1/
        {REAL}  reference 2/array-reference 2} ∅
...}]
DEND␣␣␣␣␣ ∅
```

2.8.7 Semaphore-Operationen

Die Semaphore-Operationen in CIMIC/1 bestehen aus den Anweisungen REQU und RELE, die dieselbe Bedeutung wie die PEARL-Anweisungen REQUEST und RELEASE haben.

```
{REQU}␣SEMA␣ {GLOBAL}␣symbol(  )␣␣ ∅
{RELE}       {MODUL }
```

2.9 Sonstige Anweisungen

2.9.1 LINE-Anweisungen

Die LINE-Anweisung enthält die Zeilennummer des PEARL-Programms, um eine bessere Fehlerlokalisierung zu ermöglichen für Fehler, die vom Codegenerator oder zur Laufzeit erkannt werden.

```
LINE␣␣␣␣␣ integer ∅
```

2.9.2 START- und FINISH-Anweisung

Die START- und FINISH-Anweisung sind abgesehen von LINE-Anweisungen die erste und letzte Anweisung eines CIMIC-Moduls. Sie enthalten den Modulnamen.

```
{START }
{FINISH}␣␣␣module-identifier( )␣␣ ∅
```

LEVEL-Anweisung

Um Speicherplatzoptimierungen beim Codegenerieren zu ermöglichen, wird bei den SPACE-Anweisungen am Anfang einer Prozedur oder Task die Blockstruktur in Form von zwischengeschobenen LEVEL-Anweisungen mitgeliefert. Die LEVEL-Anweisung enthält die Blocktiefe.

```
LEVEL ␣␣␣␣␣integer  ¢
```