PDV

Projekt Prozeßlenkung mit DV-Anlagen

Entwicklungsnotizen

PDV-E 63

Beschreibung des Macroübersetzers STAGE 2 als Hilfsmittel zur Realisierung der Prozeßrechnersprache PEARL auf dem Prozeßrechner Dietz 'mincal 621'

Rainer Kluttwig, Manfred Alt
Institut für Verfahrenstechnik und Dampfkesselwesen der Abteilung Automatisierungstechnik und Stromerzeugung, Uni Stuttgart

GESELLSCHAFT FÜR KERNFORSCHUNG

INSTITUT FÜR VERFAHRENSTECHNIK
UND DAMPFKESSELWESEN
PROFESSOR DR.-ING. R. QUACK
7 STUTTGART 80, PFAFFENWALDRING 23

UNIVERSITÄT
STUTTGART

Beschreibung des Macroübersetzers STAGE 2 als Hilfsmittel zur Realisierung der Prozeßrechnersprache PEARL auf dem Prozeßrechner Dietz 'mincal 621'

von
Rainer Kluttig und Manfred Alt

Dipl.-Ing. R. Kluttig und Dipl.-Ing. M. Alt sind Mitarbeiter am Institut für Verfahrenstechnik und Dampfkesselwesen (Prof. Dr.-Ing. R. Quack) der Abteilung Automatisierungstechnik und Stromerzeugung (Priv.-Doz. Dr.-Ing. E. Welfonder)

Der vorliegende Bericht entstand auf der Grundlage einer Diplomarbeit, die Herr Kluttig unter der Betreuung von Dipl.-Ing. Alt, Prof. Lauber (Institut für Regelungstechnik und Prozeßautomatisierung) und Dr.-Ing. Welfonder angefertigt hat.

INHALTSVERZEICHNIS:

f) Ersetzen eines Parameters durch eine gebildete Code-Zeile

g) "Context-gesteuerte Iteration"

h) Umwandlung von Zeichen in ganze Zahlen

2.3.2.6 Übersetzerfunktionen

a) Beenden der Übersetzung

b) Sofortige Ausgabe einer Zeile

c) Wechsel der Ein/Ausgabekanäle und Übergabe von Eingabetext an bestimmte Ausgabekanäle

d) Ablegen von Information im STAGE 2-Speicher

e) Unbedingter Sprung

f) Bedingter Sprung bei Gleichheit zweier Strings

g) Bedingter Sprung nach Vergleich zweier arithmetischer Ausdrücke

h) "Zähler-gesteuerte Iteration"

i) Auslösung eines neuen Iterationsschritts

j) Beenden der Bearbeitung des laufenden Macros

k) Zurückverfolgen von Fehlern

2.4 Rechenzeituntersuchungen für den Macroübersetzer STAGE 2

2.4.1 Abhängigkeit der Baumerstellungszeit

2.4.2 Abhängigkeit der Übersetzungszeit

2.4.3 Macroschachtelung

2.4.4 Parameterlänge

2.4.5 Vergleiche verschiedener Übersetzerversionen

2.4.6 Ergebnisse der Rechenzeituntersuchungen

2.4.7 Zusammenfassung der Ergebnisse der Rechenzeituntersuchungen

2.5 Vergleich des Macroübersetzers STAGE 2 mit anderen Macroübersetzern

KURZFASSUNG:

In dieser Arbeit wird die Verwendbarkeit des Macroübersetzers STAGE 2 für einen vom Arbeitskreis PfK (PEARL für Kleinrechner) geplanten PEARL-Compiler für Kleinrechner untersucht, mit dem eine Zwischensprache in die Assemblersprache eines Zielrechners umgesetzt werden soll.

Hierzu wird zuerst eine Beschreibung der Eigenschaften dieses Übersetzers gegeben, die durch Ergebnisse von Rechenzeituntersuchungen vervollständigt werden.

Durch Vergleich seiner wesentlichen Merkmale mit denen von entsprechenden, in der Fachliteratur beschriebenen Macroübersetzern wird die grundsätzliche Verwendbarkeit des STAGE 2 zur Codeerzeugung verdeutlicht, was auch ein Beispiel für einen Codegenerator zur Übersetzung von CIMIC/1 in den Assemblercode des Kleinrechners DIETZ 621 zeigt. Allerdings wurde eine Übersetzung nur auf einem Großrechner (CDC 6600) durchgeführt, da der STAGE 2 auf dem Kleinrechner noch nicht lauffähig ist.

ABSTRACT:

This paper is a research on the usability of the macroprocessor STAGE 2 for a PEARL-compiler for small computers, projected by the research group PfK (PEARL für Kleinrechner). It shall transfer an intermediate code into the assembly language of the target-machine.

At first the characteristics of the macroprocessor are described. They are completed by the results of examination of the computing time.

By comparing its essential characteristics to those of corresponding macroprossessors, which are described in the technical literature, the basic usability of STAGE 2 for codegeneration is made clear. This is also shown by the example of a codegenerator for translating CIMIC/1 into the assembly language of the small computer DIETZ 621. Up to now a translation, however, has been carried out only on a large computer (CDC 6500), because STAGE 2 does not yet work on the small one.

1. Einleitung

Der Einsatz von Digitalrechnern zur Lösung von Prozeßautomationsaufgaben oder zur Experimentsteuerung verlangt bisher vom Anwender umfangreiche Erfahrung im Umgang mit dem jeweils eingesetzten Rechnertyp und hohen Programmieraufwand. Um die stark steigenden Kosten der Software gegenüber der Hardware zu verkleinern, wurde es notwendig dem Anwender, der mit der Funktionsweise von Digitalrechnern nicht unbedingt vertraut sein muß, ähnliche Hilfsmittel zu bieten, wie er sie zur Lösung von kommerziellen und technisch wissenschaftlichen Problemen mit den höheren Programmiersprachen ALGOL, FORTRAN, PL/1 und COBOL besitzt.

Die Erfahrung bei der Bereitstellung sogenannter Programmpakete und problemorientierter Programmiersprachen für Teilgebiete des Anwendungsfelds hat gezeigt, daß die Schwierigkeiten beim Einsatz von Digitalrechnern zur Prozeßautomatisierung auf diesem Weg nur unzureichend behoben werden können, da die Unterschiede in den Aufgabenstellungen eines Anwendungsgebietes normalerweise so groß sind, daß nur ein kleiner Teil des Gebiets, für das ein Paket oder eine Sprache entwickelt wurde überdeckt werden kann. Zudem müssen Pakete und problemorientierte Programmiersprachen wegen der rasch fortschreitenden Technologie häufig an den aktuellen Stand der Technik angepaßt werden.

Als Möglichkeit zur Überwindung der Schwierigkeiten in der

Prozeßprogrammierung wurde die Entwicklung einer Prozeßrechnersprache angesehen, die folgende Kriterien erfüllen soll:

- Überdeckung eines möglichst weiten Aufgabenbereichs (d.h. Anwendbarkeit auf möglichst viele Probleme der Prozeßautomation und Experimentsteuerung)
- Implementierbarkeit auf verschiedene Rechner
- leichte Erlern- und Anwendbarkeit
- leichte Anpassungsfähigkeit der Programme an sich weiterentwickelnde Technologie
- Portabilität von Programmen (d.h. Übertragbarkeit auf andere Rechnertypen)
- Standardisierung der Dokumentation von Programmen

Hierzu wurde in einem vom Bundesministerium für Bildung und Wissenschaft geförderten Projekt innerhalb PDV die Realzeitsprache PEARL (Process and Experiment Automation Realtime Language) entwickelt. Die im Juni 1973 vorgestellte Projektgruppe "PEARL für Kleinrechner" hat es sich zur Aufgabe gemacht, die Implementierbarkeit dieser Sprache auf die Kleinrechner (16 K-Worte à 16 bit mit externem Massenspeicher) Mincal 621 (Dietz), Mulby 3 (Krantz) und EPR 1100 (Krupp-Atlas) zu untersuchen.

PEARL ist eine Programmiersprache für Anwendungen in der industriellen Prozeßsteuerung und der Experimentiertechnik. Sie verknüpft zur Formulierung algorithmischer Zusammenhänge erforderliche Sprachmittel mit einer hinreichenden Menge grundlegender Sprachelemente für die Realzeitprogrammierung und erlaubt so die problemgerechte Aufteilung von Anwendungsproblemen

in schwach gekoppelte Prozesse und die Beschreibung der Wechselwirkung von Prozessen mit ihrer Umwelt. Da es unmöglich erscheint, alle möglichen Hardwarekonfigurationen zu standardisieren und auf der Stufe einer Programmiersprache einheitlich zu beschreiben, wird ein PEARL-Programm also aufgeteilt in einen maschinenabhängigen, sogenannten Systemteil, der das Programm mit den aktuellen Peripherieelementen verknüpft, und einen maschinenunabhängigen sogenannten Problemteil, in dem auf die Hardware nur über Namen Bezug genommen wird. Auf diese Weise muß bei der Übertragung eines Programms auf einen anderen Rechnertyp lediglich der Systemteil modifiziert werden. PEARL erlaubt außerdem die Regelung von Automationsprogrammen in Teilprogramme und somit den systematischen Aufbau von Prozeßprogrammen durch die Möglichkeit ein Objektprogramm aus einer Menge unabhängig compilierbarer Einheiten, sog. PEARL-Programmoduln aufzubauen. Die zur Lösung eines Automationsproblems erforderlichen Programmoduln werden nach ihrer Übersetzung in einem Bindelauf mit den Funktionen des Betriebssystems und des jeweils erforderlichen Bibliotheksfunktionen verknüpft und ergeben so die lauffähige Version eines sog. PEARL-Programmsystems.

Die Übersetzung von PEARL in die Assemblersprache eines Zielrechners kann dabei wie in Bild 1, nach der von der ASME (Arbeitsgemeinschaft Stuttgart, München, Erlangen) gewählten Methode, ablaufen.

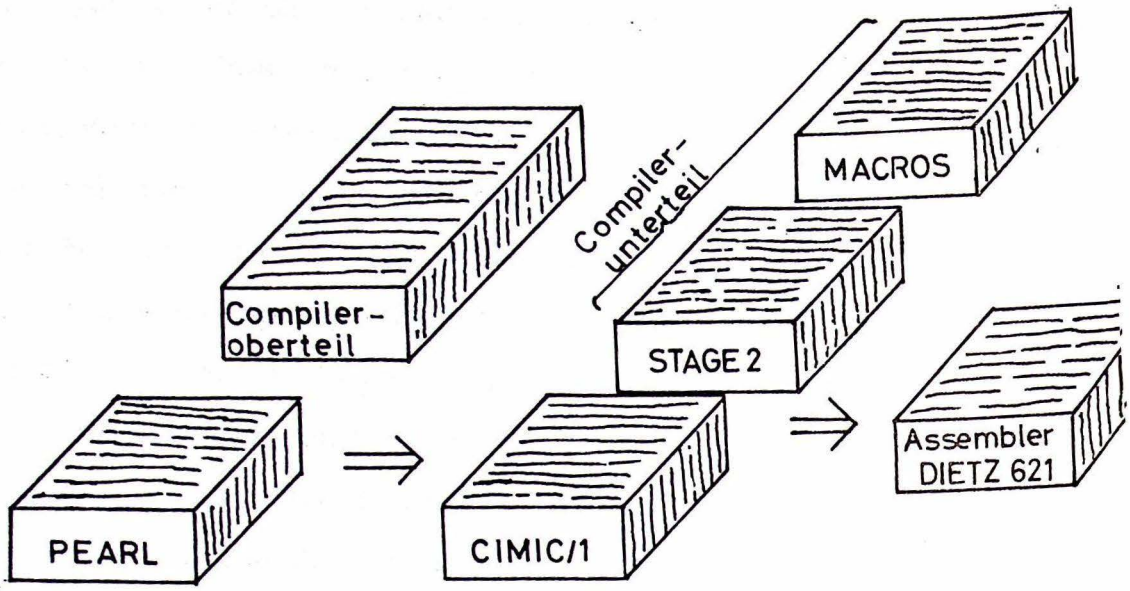


BILD 1 Compileraufbau

Wie man sieht besteht der ASME-Compiler aus 2 Teilen, einem zielmaschinenunabhängigen Oberteil und einem zielmaschinenabhängigen Unterteil, dem Codegenerator. Im oberen Compilerteil werden bei der Syntaxanalyse alle für die Übersetzung notwendigen Informationen im Zwischenstring und im Namensbuch abgelegt. Im nachfolgenden Semantiklauf werden diese Informationen in die compilerinterne Zwischensprache CIMIC/1 umsetzt. (siehe Bild 2)

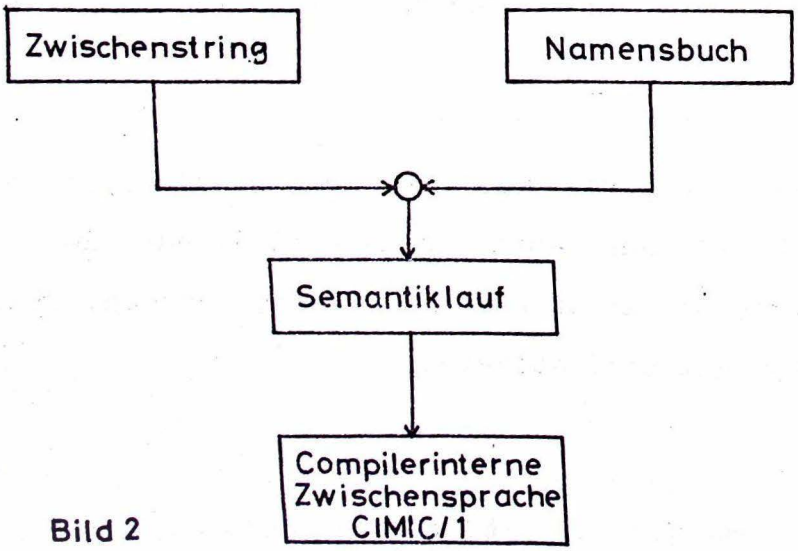


Bild 2
Übersetzungsablauf

Die Zwischensprache CIMIC/1 ist dabei so definiert, daß sie bereits mit den Assemblersprachen von Zielrechnern möglichst gut übereinstimmt und somit der zielmaschinenabhängige Compileraufwand möglichst klein wird.

Der Compilerunterteil übersetzt CIMIC/1 weiter in den Assemblercode der Zielrechner. Er besteht aus einem Steuerteil, dem Macrogenerator (Macroübersetzer) und dem Macroteil. Der Macroübersetzer, von der ASME wurde hierzu der Macroübersetzer STAGE 2 gewählt, hat die Aufgabe zu einem eingelesenen CIMIC-Befehl durch Aufruf eines entsprechenden Macrobefehls den zugehörigen Assemblercode zu generieren. Diese Arbeit soll sich mit dem Codegenerator beschäftigen. In Abschnitt 2 werden zur Codegenerierung verwendbare Macroübersetzer und hierbei insbesondere der Macroübersetzer STAGE 2 beschrieben.

Ein Codegenerator, der den algorithmischen Teil der compilerinternen Zwischensprache CIMIC/1 in den Assemblercode des DIETZ 621 übersetzt, wird in Abschnitt 3 beschrieben. Der Codegenerator ist in der Lage, Programme mit arithmetischen Befehlen, Bitoperationen, Vergleichs-, Sprungbefehle und Deklarationen zu übersetzen.

Als Beispiel zur Prüfung des Codegenerators wurden die zu einem PEARL-Programm zur Matrizenmultiplikation gehörenden CIMIC-Befehle in ein Programm im Assemblercode des DIETZ 621 umgewandelt.

2. Macroübersetzer

2.1 Was ist ein Macroübersetzer?

Ein Macrobefehl ist in seiner Grundbedeutung ein Mittel, eine bestehende Grundsprache durch Einführen neuer Einheiten zu erweitern, und zwar sollen diese Einheiten ermöglichen, eine bestimmte Befehlsfolge in einem Programm durch eine einzige Anweisung zu ersetzen. Diese Befehlsgenerierung übernehmen sogenannte Macrogeneratoren, die meist eng mit dem zugehörigen Assemblerübersetzer verknüpft und deshalb mit diesem als Einheit betrachtet werden können und dann Macroassembler genannt werden.

Für derartige Macrogeneratoren gelten strenge Syntaxregeln, die durch den Sprachumfang der Grundsprache (z. B. Assemblersprache) vorgegeben sind. Sie können nur Anweisungen der Grundsprache in andere Anweisungen der Grundsprache umsetzen. Es wurde nun eine Reihe von Macroübersetzern entwickelt, die einen allgemeingültigen Formalismus zum Umsetzen und Verarbeiten von Zeichenfolgen beinhalten. Sie analysieren eine vom Benutzer erstellte Zeichenfolge (Quellsprache) nach bestimmten Regeln und setzen diese in eine Ersatzzeichenfolge (Zielsprache) um. Auf diese Weise kann der Benutzer Macros in einer für ihn geeigneten Sprache schreiben und ist nicht an Assemblersprachen und deren Übersetzer gebunden. Diese Macrogeneratoren sind also nicht nur ein Hilfsmittel für das Programmieren in einer bestimmten Sprache (z.B. Assemblersprache), sondern können auch für Übersetzungsarbeiten von höheren Programmiersprachen z.B. zur Übersetzung

von CIMIC/1 in Assemblercode verwendet werden.

Im allgemeinen haben Macrodefinitionen folgende Form

Macrodefinition: MACRO NAME (P_1, ..., P_n)

MACROKÖRPER

END

Die Strings MACRO und END begrenzen die Macrodefinition. NAME stellt den Namen des Macrobefehls dar, und P_1, ..., P_n sind die formalen Parameter des Macros. Ein Macroaufruf besteht aus dem Macronamen und einer Liste aktueller Parameter, deren Zahl kleiner oder gleich der Zahl der formalen Parameter sein muß.

Macroaufruf: NAME (A_1, ..., A_m) , $m \leq n$

Wird dem Macrogenerator ein solcher Aufruf eingegeben, so ersetzt er diesen durch den zugehörigen Macrokörper, wobei in diesem die formalen durch die aktuellen Parameter ersetzt werden. Ist $m < n$, so werden für die übrigen formalen Parameter vom Übersetzer nach bestimmten Regeln Werte erzeugt.

2.2 Forderungen an einen Macroübersetzer.

Um die Codeerzeugung mittels Macrogeneratoren in einem möglichst weiten Bereich zu ermöglichen, sollten diese möglichst portabel, d.h. auf beliebigen Rechnern lauffähig und auf beliebige Zielsprachen anwendbar sein und außerdem zuverlässig und schnell arbeiten. Hierzu müssen an einen solchen Übersetzer folgende Forderungen gestellt werden:

1. Der Übersetzer sollte in einer maschinenunabhängigen Sprache (z.B. FORTRAN) geschrieben sein.

2. Die Zuordnung der Eingabezeichenfolge zu den entsprechenden Macros und somit zur gewünschten Ausgabezeichenfolge muß eindeutig sein, d.h. es muß ein zuverlässiger Suchalgorithmus existieren.

3. Die Definition der einzelnen Macroköpfe (auch Macrokörper) soll möglichst formatfrei sein.

4. Der Macroübersetzer soll möglichst wenig Kernspeicher benötigen und schnell arbeiten.

5. Macroteil und Steuerteil eines Macrogenerators müssen möglichst unabhängig von einander sein.

6. Die Macros müssen möglichst einfache und leicht ausführbare Ausgabeanweisungen erlauben.

Nach /2/ muß man, um einen leistungsfähigen Übersetzer zu er-

halten, der nicht nur Eingabetext durch Text ersetzt, sondern z.B. auch Codeoptimierung erlaubt, an diesen weitere Forderungen stellen.

a) Macroschachtelung, d.h. Aufruf anderer definierter Macros innerhalb einer Macrodefinition. So kann der Compileraufwand stufenweise durch Einführung von Macros, die von bereits eingeführten Macros aufgerufen werden, reduziert werden.

Beispiel: Macro 1 :
```
MACRO ADD, A, B, C
FETCH, A
ADD, B
STORE, C
END
```

Macro 2 :
```
MACRO COMPLEXADD, A, B, C
ADD, A, B, C
ADD, A+1, B+1, C+1
END
```

b) Abfragen von Bedingungen in den Macros (bedingte Macros)
Um z.B. P+Q+R mit Macro 1 addieren und nach S abspeichern zu können, müßte der Aufruf lauten:

```
ADD, P, Q, ACC
ADD, ACC, R, S
```

wobei ACC für das Arbeitsregister steht. So würde ein un-

aktueller Parameter zugeordnet werden. In Macro 4 erzeugt der Macroübersetzer bei jedem Aufruf von sich aus ein anderes Zeichen.

d) Zusammenfassung von Blöcken.
Steht z.B. in einem Macroaufruf als aktueller Parameter ein weiterer Macroaufruf mit eigenen aktuellen Parametern, so muß, um Uneindeutigkeit zu vermeiden, dieser Block durch besondere Steuerzeichen, wie z.B. Klammern, als Einheit zusammengefaßt werden können.

e) Neue Macrodefinitionen innerhalb eines Macros

```
Macro 5: MACRO VARIABLE, A.
                A: 1 Speicherplatz reserviert
                MACRO A
                FETCH, A.
                END
         END
```

f) Iteration über eine Parameterliste.

```
Macro 6: MACRO VARIABLES, X
                repeat over X
                VARIABLE, X
                END
```

Hierbei bedeutet "repeat over X", daß X als Parameterliste betrachtet wird, und daß die Auswertung des Macro-

nötiger Speicher- und Ladebefehl generiert. Ein bedingtes Macros, welches den hier zu erzeugenden Code optimiert und überflüssige Befehle wegläßt hat die Form:

```
Macro 3: MACRO ADD, A,B,C,
                 {if A ≠ ACC
                 {FETCH, A
                  ADD, B
                 {if C ≠ ACC
                 {STORE, C
                  END
```

c) Eindeutige Symbolgenerierung.

Die Notwendigkeit dieser Forderung wird ersichtlich, wenn man an Marken denkt, die nur innerhalb eines bestimmten Macros zugänglich sein sollten, bei mehrmaligem Aufruf dieses Macros aber nicht zu Mehrfachdefinitionen führen dürfen.

```
Macro 4: MACRO OVERDRAW, X,Y, ACTION
                 creat 2
                 FETCH, X
                 SUB, Y
                 PLUSJUMP, Z
                 ACTION
                 Z: Leerzeile
                 END
```

Wäre in Macro 4 keine Symbolgenerierung möglich, so müßte die Marke Z in der Parameterliste dieses Macros enthalten sein und ihr müßte bei jedem Aufruf des Macros ein anderer

körpers für jedes Element dieser Liste vorgenommen wird.

Beispiel: VARIABLES (X,Y,Z)

```
VARIABLE, X
VARIABLE, Y
VARIABLE, Z
```

g) Zeitweilige Belegung von Speicherplätzen während eines Übersetzungslaufes und Ausführung arithmetischer Operationen, was die Möglichkeiten zur Codeoptimierung vergrößert.

2.3 Der Macroübersetzer STAGE 2

2.3.1 Wirkungsweise

Mit dem Macroübersetzer STAGE 2, auf den im folgenden näher Bezug genommen wird, wurde 1968 an der Universität Colorado von W. M. Waite ein leicht auf verschiedene Rechenanlagen übertragbares Programm entwickelt, welches den in 2.2 gestellten Forderungen in hohem Maße entspricht. Es analysiert eine vom Benutzer eingegebene Zeichenfolge (Quellsprache) nach bestimmten Regeln und setzt sie in eine neue Zeichenfolge (Zielsprache) um, die zur weiteren Bearbeitung (z. B. durch einen Compiler oder Assembler) bereitgestellt werden kann. Der STAGE 2 ist im Gegensatz zu herkömmlichen Macroübersetzern von den Eigenheiten der Sprache der verwendeten Rechenanlage unabhängig. Er besitzt ein verallgemeinertes Ein/Ausgabesystem, und die hier vorliegende Version (STAGE 2-2) wurde schon auf verschiedenen Rechnern implementiert. Sie steht z. B. als Lochkartendeck in FORTRAN - Version zur Verfügung.

Bei den herkömmlichen Macroübersetzern, bei denen ein Macro, wie in Abschnitt 2.1 beschrieben, definiert wird, ist die Macrodefinition formatgebunden. Da der STAGE 2 aber für möglichst viele zu übersetzende Sprachen verwendbar sein soll, wird bei ihm ein sehr allgemein gehaltenes Verfahren der Mustererkennung benutzt. Hier kann die Anfangszeile einer Macrodefinition, die den Namen des entsprechenden Macros darstellt, aus einer beliebigen Zeichenfolge bestehen, die nur durch ein spezielles Zeichen, die "Source-end-of-line-flag" (SEF), abgeschlossen sein muß +). Parameter können an beliebiger Stelle dieser Zeile durch Parameterplatzhalter, die "Source-parameter--flags" (SPF) vorgesehen werden.

Jeder Satz von Macrodefinitionen muß beim STAGE 2 mit einer besonderen Zeile, der "flag-line" (s. 2.3.2.2) beginnen, in der dem Übersetzer Sonderzeichen (wie z.B. SEF oder SPF) angegeben werden, wobei deren Definition dem Benutzer überlassen bleibt. Jede einzelne

+) s. Seite 21

Macrodefinition besteht aus der Anfangszeile, die der STAGE 2 al Zeilenmuster verwendet, und einem zugehörigen Macrorumpf, dem Code-body. Jede Zeile des Code-body muß durch ein weiteres Sonde zeichen, die "MCT-end-of-line-flag" abgeschlossen+). Dieses Zeichen in einer eigenen Zeile beschließt auch eine Macrodefiri- tion. Beim letzten Code-body eines Satzes von Macrodefinitionen müssen in dieser Zeile zwei solche Zeichen stehen.

Der letzten Macrodefinition folgen sofort die Macroaufrufe. Ein Macro wird auch hier durch seinen Namen aufgerufen. Jedoch gehör hier zum Namen alle Zeichen des Zeilenmusters, mit Ausnahme der Parameterplatzhalter. Der Aufruf ist hier also nicht formatgebur den und kann direkt als Anweisung in der Quellsprache erfolgen, wobei diese Zeichen für Zeichen mit den vom Benutzer eingegebene Zeilenmustern verglichen wird. Kann keine Übereinstimmung ge- funden werden, so wird die eingelesene Zeile ohne Änderung wiede ausgegeben. Bei Übereinstimmung der eingelesenen Zeile mit einem Zeilenmuster werden in den zugehörigen Macrorumpf die aktuellen Parameter übergeben, und der Macroübersetzer steuert dann die Au gabe des entsprechenden Codes.

Im einfachsten Fall besteht der "Code-body" eines Macros aus Tex in der Zielsprache, in die die Werte der aktuellen Parameter ein gesetzt werden. Um aber mehr als bloßes Ersetzen eines Textes durch einen anderen Text mit einem Übersetzer zu ermöglichen, wurden von McIlroy /2/ die in 2.2 beschriebenen Forderungen an einen Macroübersetzer gestellt.

Zur Erfüllung dieser Forderungen besitzt der Übersetzer STAGE 2 9 mögliche Arten der Parameterumformung und 10 Übersetzerfunktio nen. Durch Parameterumformung werden die aktuellen Parameter bearbeitet und mit Hilfe der Information aus dem dem STAGE 2 zur Verfügung stehenden Speicher ergänzt und in die auszugebende Code-Zeile übergeben. Die Übersetzerfunktionen steuern das Verh ten des Übersetzers (z.b. Sprünge oder Beenden des Übersetzungs- vorgangs).

Im Code -body eines Macros können weitere Macroaufrufe stehen (Macroschachtelung). Dieser Schachtelung sind nur durch die dem STAGE 2 zur Verfügung stehende Speicherplatzmenge Grenzen gesetzt.

+) SEF und MCT-end-of-line-flag können fehlen, dann sind aber alle Leerzeichen bis zum Zeilenende von Bedeutung (z. B. Lochkartenende) und werden gespeichert.

2.3.1.1 Mustererkennung

Wie bereits erwähnt, wird die Anfangszeile einer Macrodefinition beim STAGE 2 als Zeilenmuster verwendet. Dieses Zeilenmuster besteht aus sogenannten festen Zeichenfolgen und Parameterplatzhaltern. Beim Vergleichsvorgang werden nun die Zeichen der Eingabezeile mit den festen Zeichenfolgen der Zeilenmuster verglichen un bei Übereinstimmung diesen zugeordnet. Den Parameterplatzhaltern werden dann die restlichen Zeichenfolgen der Eingabezeile, und wenn diese nicht vorhanden sind, "leere Zeichenfolgen" zugewiesen. Diese Zeichenfolgen müssen aber bezüglich Klammern ausgeglichen sein, d.h. gleich viel linke wie rechte Klammern enthalten. Die Parameterplatzhalter der Zeilenmuster werden von links nach rechts mit 1 beginnend bis höchstens 9 numeriert. (In einem Zeilenmuster dürfen höchstens neun Parameterplatzhalter auftreten.)

Die Mustererkennung, d.h. die Zuordnung von Zeichen der Eingabezeile zu Zeichen des Zeilenmusters oder Parameterplatzhaltern muß nach bestimmten Regeln ablauffen, um Mehrdeutigkeit, d.h. die Übereinstimmung einer Eingabezeile mit mehreren Zeilenmustern, zu vermeiden. Hierzu wird die Menge aller definierten Zeilenmuster zeichenweise baumförmig aufgegliedert. Jedes Zeichen entspricht einem Zweig des Baumes, die durch Verzweigungspunkte (Knoten) getrennt werden. Von jedem dieser Knoten können beliebig viele Zweige ausgehen, aber nur einer kann auf ihn zulaufen. Der Ursprung des Baums ist die Wurzel (Root).

Das nachfolgende Beispiel (siehe Bild 3) zeigt, wie dieser Baum aus den möglichen Zeilenmustern aufgebaut wird. Dabei stellt im Baum c eine Zeilenendemarke und p einen Parameterplatzhalter dar, um die Unabhängigkeit von den vom Benutzer verwendeten Zeichen zu verdeutlichen.

Entspricht nun eine eingegebene Zeile dem Baum von der Wurzel bis zu einer Zeilenendemarke (c), so wird sie als Aufruf des entsprechenden Zeilenmusters erkannt. Die eingegebene Zeile wird nach folgenden Regeln mit dem aus den Zeilenmustern entstehenden Baum verglichen. Diese Regeln sind für den Benutzer nur bei der Deutung unerwartete Ergebnisse des Mustervergleichs von Bedeutung.

EINGEGEBENE ZEILENMUSTER:

(1) SAM=A$

(2) SAM=!$

(3) !=!$

(4) !=A$

(5) SAM=!=!$

(6) SAM=JOE$

(7) !=!=!$

AUFGEBAUTER BAUM:

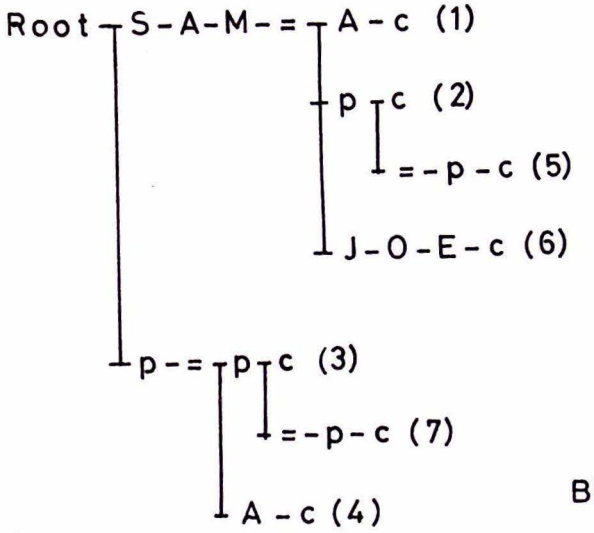


Bild 3 Baumaufbau aus Zeilenmustern

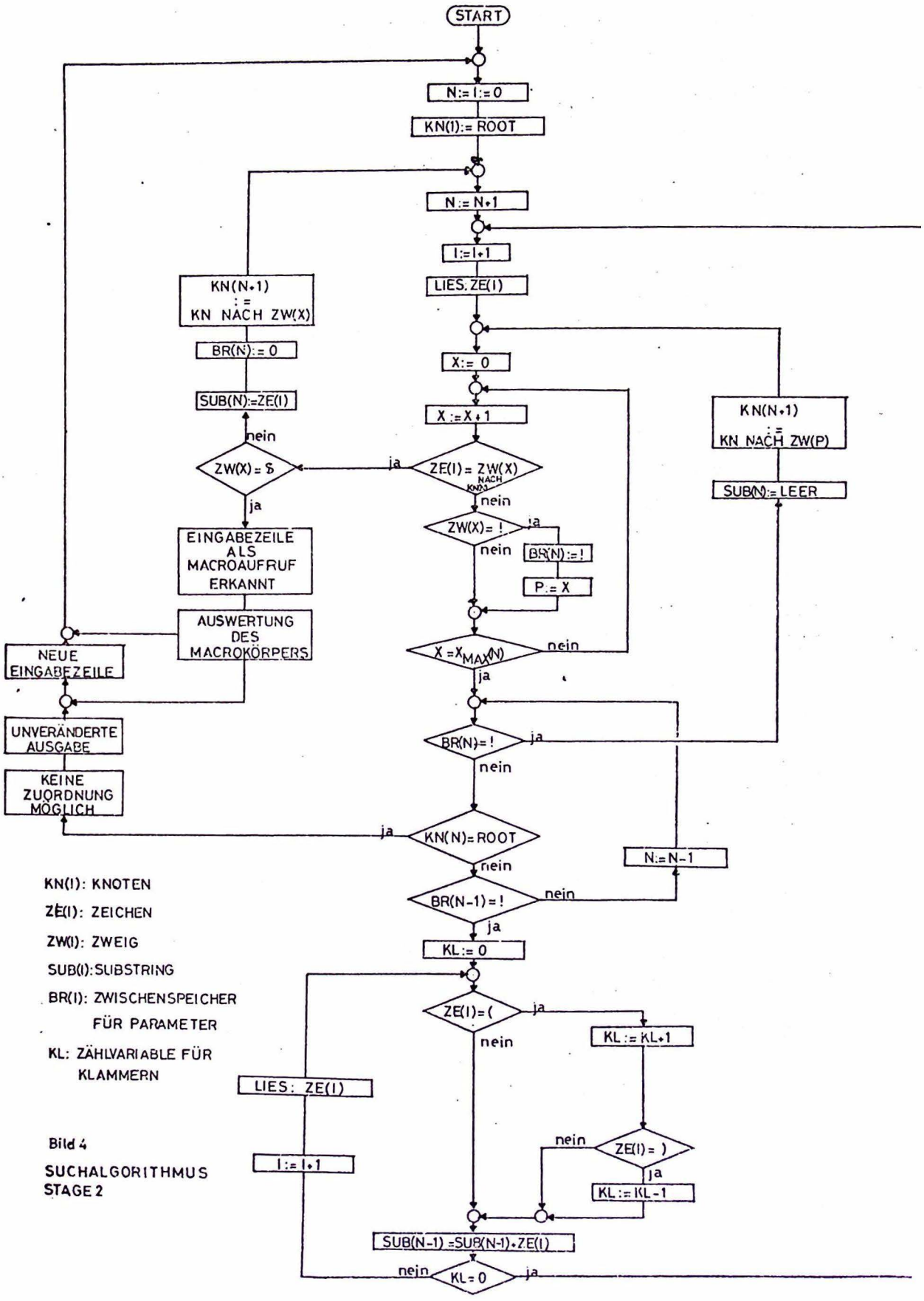
Regel 1: Regel 2 wird auf die Wurzel des Baumes und das erste eingelesene Zeichen angewandt.

Regel 2: Alle vom momentan betrachteten Knoten ausgehenden Zweige werden mit dem eingelesenen Zeichen verglichen. Gibt es eine Übereinstimmung, so wird über diesen Zweig zum nächsten Knoten fortgeschritten, für den dann mit dem nächsten eingelesenen Zeichen der gleiche Vorgang abläuft. Wird hierbei eine Zeilenendmarke erreicht, so ist eine Zeile erkannt und wird dem entsprechenden Macro zugeordnet. Gibt es keine Übereinstimmung, so wird Regel 3 aufgerufen.

Regel 3: Konnte nach Regel 2 keine Übereinstimmung gefunden werden, und stellt einer der vom betrachteten Knoten ausgehenden Zweige einen Parameterplatzhalter dar, so wird Regel 2 auf den über diesen Zweig zu erreichenden Knoten und das eingelesene Zeichen angewandt, während diesem Zweig ein "Leerstring" zugeordnet wird. Ist dies nicht der Fall, so wird auf den betrachteten Knoten Regel 4 angewandt.

Regel 4: Ist dieser betrachtete Knoten die Wurzel, und brachten Regel 2 und 3 kein Vorrücken auf dem Baum, so wird der Vergleich als gescheitert betrachtet. Die eingelesene Zeile wird nicht erkannt und unverändert ausgegeben. Sonst wird, wenn der auf den betrachteten Knoten zulaufende Zweig keinen Parameterplatzhalter darstellt, Regel 3 auf den vorigen Knoten und das eingelesene Zeichen angewandt. Ist dieser Zweig aber ein Parameterplatzhalter, so wird Regel 5 auf den momentan betrachteten Knoten und das gerade eingegebene Zeichen angewandt.

Regel 5: Dem Substring, der dem Zweig entspricht, der auf den betrachteten Knoten zuläuft, wird der kleinste bezüglich Klammern ausgeglichene und beim gerade eingelesenen Zeichen beginnende Substring angehängt. Hiernach wird Regel 2 auf diesen Knoten und das dem neuen Substring folgende Zeichen angewandt. Gibt es keinen solchen Substring, so wird Regel 4 auf den vorangehenden Knoten und das erste eingelesene Zeichen angewandt, das einem Parameterplatzhalter zugeordnet wurde. Hatte der Platzhalter zu einem Leerstring gehört, so wird das gerade eingelesene Zeichen genommen. Der Substring, der dem am weitesten links stehenden Parameterplatzhalter zugeordne wurde, wird zu Parameter 1 gemacht usw. Nach der Festlegung der Parameter beginnt die Auswertung des Macrorumpfes.



KN(I): KNOTEN

ZE(I): ZEICHEN

ZW(I): ZWEIG

SUB(I):SUBSTRING

BR(I): ZWISCHENSPEICHER FÜR PARAMETER

KL: ZÄHLVARIABLE FÜR KLAMMERN

Bild 4

SUCHALGORITHMUS STAGE 2

In Bild 4 ist der nach diesen Regeln ablaufende Suchalgorithmus des STAGE 2 in leicht vereinfachter Form in einem Flußdiagramm dargestellt. Die Suche beginnt beim Knoten KN(1)(Wurzel).
Alle von einem Knoten KN(N) ausgehenden Zweige ZW(X) werden mit dem betrachteten eingelesenen Zeichen ZE(I) verglichen. Bei Übereinstimmung wird der Suchvorgang abgebrochen und eine Zeile gilt als erkannt, wenn das eingelesene Zeichen eine Zeilenendemarke war, sonst wird das einem Zweig entsprechende Zeichen einem Substring SUB(N) zugeordnet, und der diesem Zweig folgende Knoten wird KN(N+1), für den wiederum der gleiche Vorgang abläuft.

Kann keine Übereinstimmung zwischen einem Zweig und dem betrachteten eingelesenen Zeichen gefunden werden, so wird abgefragt, ob einer der Zweige ZW(X) ein Parameterplatzhalter ist, der dann eine Kennvariable BR(N) setzt, die im nächsten Schritt abgefragt wird. ist BR(N) gleich !, so wird über den zu diesem Parameterplatzhalter gehörenden Zweig zum Knoten KN(N+1) fortgeschritten und dem Substring SUB(N) wird ein Leerstring zugeordnet. War BR(N) ungleich !, und ist KN(N) die Wurzel, so ist keine Zuordnung möglich, und die eingelesene Zeile wird unverändert ausgegeben.

Ist KN(N) jedoch nicht die Wurzel, so wird so lange zum vorigen Knoten KN(N-1) zurückgesprungen bis dieser entweder der Wurzel entspricht, oder von ihm ein Parameterzweig ausgeht. Der diesem entsprechende Substring wird dann mittels der Klammerzählvariablen KL um den kleinsten bezüglich Klammern symmetrischen Eingabestring erweitert. Danach wird der Vergleichsvorgang bei dem diesem Parameterzweig folgenden Knoten mit dem nächsten Eingabezeichen fortgesetzt.

Als Beispiel soll das zu der Eingabezeile SAM = (B = C) = A gehörende Zeilenmuster im Baum nach Bild 1 gesucht werden. Regel 1 bewirkt, daß der Vergleichsvorgang beginnt, indem sie Regel 2 aufruft, die dann auf die Wurzel und das erste eingegebene Zeichen angewandt wird. Das erste Zeichen der Eingabezeile ist hier "S", und es gibt einen von der Wurzel ausgehenden Zweig, der "S" darstellt. "S" wird also diesem Zweig zugeordnet. Da dieser Zweig keiner Zeilenendemarke entspricht, wird nun Regel 2 auf den über diesen Zweig erreichten Knoten und das nächste Zeichen der Eingabezeile angewandt. Dies ist "A", und es gibt einen "A"-Zweig, der von diesem Knoten ausgeht. Somit kann Regel 2 weiter benutzt werden.

Dieser Vorgang wiederholt sich für die folgenden Zeichen, bis man zum Knoten nach dem Zeichen "=" gelangt. Für das jetzt eingelesene Zeichen "C" kann kein entsprechender Zweig gefunden werden, und wir müssen Regel 3 benutzen. Von diesem Knoten geht ein Parameterzweig aus. Diesem wird ein "Leerstring" zugeordnet. Nun wird Regel 2 auf den über diesen Zweig erreichten Knoten und das eingegebene Zeichen "C" angewandt, aber mit Regel 2 läßt sich auch hier keine Übereinstimmung finden, so daß man Regel 3 versuchen muß. Da aber von diesem Knoten kein Parameterzweig ausgeht, müssen wir Regel 4 anwenden. Der betrachtete Knoten ist nicht die Wurzel, aber der auf ihn zulaufende Ast ist ein Parameterzweig, so daß Regel 5 dazu benutzt wird, den "Leerstring", der ursprünglich diesem Parameterzweig zugeordnet wurde, zu erweitern. Der kleinste, bezüglich Klammern ausgeglichene Substring ist "(B = C)". Dieser wird Parameter 1, und Regel 2 wird auf den nächsten Knoten und das der Klammer ")" folgende Zeichen "=" angewandt. Hier kann wieder eine Übereinstimmung gefunden werden, aber beim nächsten eingelesenen Zeichen "A" versagt Regel 2 erneut, so daß wiederum Regel 3 dazu benutzt wird, um auf dem Baum über einen Parameterzweig vorzurücken. Diesem wird wieder ein "Leerstring" zugeordnet. Auch hier rreicht man mit Regel 2 und 3 keinen Fortschritt. Über Regel 4 wird dieser "Leerstring" nach Regel 5 um den ausgeglichenen Substring "A" erweitert. "A" wird Parameter 2. Nun wird Regel 2 auf die "A" folgende Zeilenendemarke angewandt. Es gibt eine Übereinstimmung und die Eingabezeile wird als Aufruf des Zeilenmusters (5) erkannt.

Der hier beschriebene Mustervergleich durch den STAGE 2 erlaubt formatlosen Aufruf von Macros, verlangt aber, daß in jeder Zeile nur eine Anweisung steht. Leerzeichen werden berücksichtigt.

Die Richtigkeit des beschriebenen Beispiels soll mit dem Zeilenmuster nach Bild 3 gezeigt werden. Die Code-bodys der einzelnen Macros werden durch Namen gebildet. Der zum aufgerufenen Macro gehörende Name wird auf Lochkarte ausgegeben. Mit einer später beschriebenen Übersetzerfunktion kann jedoch ein anderer Ausgabekanal gewählt werden. In der "flag-line" wird vereinbart:

Ø als SEF und als MCT-end-of-line-flag
! als SPF

Macrodefinitionen:

```
SAM=AØ
ANTON Ø
Ø
SAM=!Ø
OTTO Ø.
Ø
!=!Ø
KARL Ø
Ø
!=AØ
EGON Ø
Ø
SAM=!=!Ø
GISELA Ø
Ø
SAM=JOEØ
INGE Ø
Ø
!=!=!Ø
MONIKA Ø
ØØ
```

Eingabezeile:

```
SAM=(B = C)=AØ
```

Ausgabe:

```
GISELA
```

2.3.1.2 Code-body

Der Code-body ist der Rumpf der Macros. Mit seiner Hilfe setzt der STAGE 2 die Eingabezeile in Zeilen der Zielsprache um. Hierbei werden in diese Zeilen entweder direkt Zeichen der Code-body-Zeilen, Parameterplatzhaltern zugeordnete Eingabezeichen oder Zeichen aus einem vom STAGE 2 gebildeten Speicher übertragen. Jede Zeilenendemarke im Code-body bewirkt auch die Beendigung der gerade gebildeten Neuzeile. Die weitere Untersuchung des momentan betrachteten Code-bodys wird daraufhin zunächst ausgesetzt, und der STAGE 2 sucht den Baum daraufhin zunächst nach einem zu dieser Neuzeile passenden Zeilenmuster ab. Existiert ein solches, so wird nun zuerst dessen Code-body vollständig untersucht. Wenn dies geschehen ist oder wenn kein Zeilenmuster gefunden werden kann, wird der alte Code-body weiteruntersucht.

Da der Code-body also eigentlich ein Programm zur Bildung neuer Zeilen ist, müssen in ihm Wertzuweisungen, Berechnung arithmetischer Ausdrücke bzw. bedingte und unbedingte Sprungbefehle und Wiederholungsschleifen möglich sein. Hierzu dienen dem STAGE 2 Parameterumformungen bzw. Übersetzerfunktionen, deren Möglichkeiten in Abschnitt 2.3.2 dargestellt werden. Die Anweisungen dieses Programms zur Bildung der Neuzeilen werden durch den Typ der Elemente des Code-bodys bestimmt, und zwar gibt es hierbei vier verschiedene Typen, zu deren Unterscheidung in der "flag-line" die MCT-end-of-line-flag und das Fluchtsymbol definiert werden müssen.

Typ 0 - einzelne Zeichen (z.B. Ziffern, Buchstaben);

Typ 1 - Zeilenende oder Zeilenendemarke;

Typ 2 - Elemente, die Parameterumwandlung bewirken;

Typ 3 - Elemente, die Übersetzerfunktionen aufrufen.

Beim Lesen des Code-body wird nun jedes einzelne Zeichen untersucht. Die Wirkung eines Elements vom Typ 1 wurde bereits erwähnt. Es bewirkt die Beendigung einer gebildeten Neuzeile und der Rest der Code-body-Zeile wird überlesen. Beim Auftreten eines Fluchtsymbols wird sofort das nächste Zeichen untersucht, dem dann eine spezielle Bedeutung zukommt, wenn es sich entweder um

eine Ziffer oder einen Buchstaben handelt. Dann werden nämlich das Fluchtsymbol und die beiden folgenden Zeichen, wobei das zweite Zeichen nach dem Fluchtsymbol stets eine Ziffer sein muß, zu einem Element vom Typ 2 bzw. Typ 3 zusammengefaßt, das dann entsprechendes Verhalten des Übersetzers bewirkt.

Alle übrigen Zeichen im Code-body sind Elemente vom Typ 0 und haben keine besondere Funktion. Sie werden wie ein dem Fluchtsymbol folgendes Fluchtsymbol oder eine diesem folgende Zeilenendmarke direkt in die Neuzeile übergeben.

2.3.1.3 Code-Erzeugung (Beispiel)

In diesem Abschnitt wird die Code-Erzeugung, d.h. die Bildung von neuen Ausgabezeilen in der Zielsprache an einem Beispiel verdeutlicht. Doch zuerst soll das Verhalten des Übersetzers hierbei noch einmal zusammengefaßt werden.

Nach der Erkennung eines Zeilenmusters sucht der STAGE 2 den zugehörigen Rumpf nach den verschiedenen Elemente ab. Elemente vom Typ 0 werden vom Übersetzer ohne weitere Bearbeitung in die Code-Zeilen übertragen.
Elemente vom Typ 2 signalisieren dem Übersetzer, daß ein aktueller Parameter auf eine bestimmte Art umgewandelt und das Ergebnis dieser Umwandlung an Stelle des Typ 2-Elements in die Code-Zeile übertragen werden soll. Dabei bestimmt die erste Ziffer in diesem Element die Nummer des zu bearbeitenden aktuellen Parameters und die zweite die Art der Umwandlung (z.B. einfaches Kopieren). Als Ergebnis des Mustervergleichs können den aktuellen Parametern nur Nummern von 1 bis 9 zugeordnet werden, trotzdem kann die erste Ziffer in einem Typ 2-Element auch Null sein. Auf diese Weise können in einem Code-body z.B. vom Mustervergleich unabhängige Marken gebildet werden, denn der Übersetzer setzt an Stelle des Elements von sich aus Ziffern in die Neuzeile ein. So sind 10 verschiedene Ziffern in einem Code-body möglich, zu deren Unterscheidung dem Übersetzer die zweite Kennziffer des Typ 2-Elements dient.

Parameterumformung betrifft immer nur einzelne Elemente des Code-body und ihr Ergebnis wird in die Ausgabezeilen übertragen, hat aber keinen Einfluß auf den Ablauf des Übersetzungsvorganges. Dieser wird durch Übersetzerfunktionen, die durch Typ-3-Elemente aufgerufen werden, gesteuert. Durch sie ändert sich das Verhalten des Übersetzers in Bezug auf einzelne Code-body-Zeilen, einen ganzen Code-body oder die restlichen Eingabezeilen, indem z.B. der Übersetzungsvorgang abgebrochen wird. Sie bewirken aber keinen Informationsfluß in die Ausgabezeilen hinein.

Elemente vom Typ 3 haben das Format eFc, wobei e das Fluchtsymbol darstellt, und c ($0 \leq c \leq 9$) die auszuführende Funktion bestimmt. Mit Ausnahme der Funktionen 0 und 9 (s. 2.3.2.6 a und j), bei denen der laufende "Code-body" endgültig verlassen wird, wird nach einem Typ-3-Element erst das übernächste Element untersucht. Typ-3-Elemente stehen im allgemeinen am Ende einer Code-body-Zeile und nach der Ausführung der zugehörigen Übersetzerfunktion liegt eine noch leere Code-Zeile vor, welche wegen der folgenden Zeilenendemarke auch leer bliebe. Da diese überlesen wird, kann in die Neuzeile sofort das Ergebnis der Auswertung der nächsten Code-body-Zeile übertragen werden, und somit wird die Ausgabe überflüssiger Leerzeilen vermieden.

Trifft der Übersetzer auf ein Element vom Typ 1, so erkennt und markiert er ein Zeilenende und unterbricht das weitere Absuchen des Code-body. Alle aktuellen Parameter werden gespeichert und die abgeschlossene Code-Zeile wird vom STAGE 2 wie eine Eingabezeile dem Mustervergleich unterzogen (Macroschachtelung).

Bei Übereinstimmung mit einem Zeilenmuster wird nun zuerst dessen Code-body untersucht.
Für das folgende Beispiel gilt:

```
ß SEF
ß MCT-end-of-line-flag   (Typ-1-Element)
! Fluchtsymbol
! SPF (Parameterplatzhalter)
!d0  (siehe 2.3.2.5,a) Typ-2-Element (mit 1 ≤ d ≤ 9)
     bewirkt, daß der aktuelle Parameter "d" in die Code-Zeile
     kopiert wird.
```

!F1 (s. 2.3.2.6,b) Typ-3-Element bewirkt, daß die gebildete Code-Zeile sofort ausgegeben (auf Lochkarte), und der Rest der Code-body-Zeile überlesen wird.

```
Macro:  ! = ! + ! ø    ADDITION
        FETCH  !20 ø
        ADD    !30 !F1ø
        STORE  !10 !F1ø
        ø
```

Eingabezeile: A = B + C ø

Ausgabe (auf Lochkarte):
```
FETCH  B
ADD    C
STORE  A
```

Nachdem die Eingabezeile als Aufruf des obigen Macros erkannt wurde und die Substrings "A", "B", "C" zu Parameter 1,2,3 gemacht wurden, beginnt der STAGE 2 den Code-body abzusuchen. Zuerst werden 6 Zeichen vom Typ 0 (einschließlich Leerzeichen) erkannt und unverändert in die Code-Zeile übergeben. An Stelle des folgenden Typ-2-Elements wird der aktuelle Parameter 2, nämlich "B", übergeben. Es folgt ein Typ-1-Element, d.h. die Code-Zeile ist abgeschlossen und lautet FETCH B. Nun wird FETCH B wie eine Eingabezeile dem Mustervergleich unterzogen, jedoch kann kein entsprechendes Zeilenmuster gefunden werden, so daß diese Zeile auf Lochkarte ausgegeben wird und der STAGE 2 die nächste Code-body-Zeile untersucht. In die neue Code-Zeile wird an Stelle des Typ-2-Elements der aktuelle Parameter C übergeben. Das folgende Typ-3-Element wirkt auf den Übersetzer ein und die bestehende Code-Zeile wird ohne nochmaliges Vergleichen sofort auf Lochkarte ausgegeben, und der Übersetzer beginnt eine neue Code-Zeile. Die dem Typ-3-Element folgende Zeilenendemarke wird überlesen, und in die Code-Zeile wird das Ergebnis der nächsten Code-body-Zeile übertragen. Das Ende des zu bearbeitenden Macros wird dem Übersetzer durch eine Zeile mit einer MCT-end-of-line-flag angegeben.

2.3.2 Parameterumformungen und Übersetzerfunktionen

2.3.2.1 Allgemeines

Für die Formatangaben der Steuerfunktionen des STAGE 2 gelten folgende Vereinbarungen:

Einzelne Zeichen
(z.B. Großbuchstaben und Ziffern): stehen für sich selbst;

c: steht für beliebige Zeichen, außer Zeilenendemarken;

d: steht für Ziffern zwischen 1 und 9, diese eingeschlossen;

e: steht für das Fluchtsymbol;

m,n: stehen für Ziffern zwischen 0 und 9, diese eingeschlossen.

2.3.2.2 Flag-line

Wie bereits erwähnt, muß jeder Satz von Macrodefinitionen mit der "flag-line" beginnen, deren Zeichen vom Benutzer willkürlich gewählt werden können. Die Bedeutung der einzelnen Zeichen, deren Reihenfolge einzuhalten ist, soll am folgenden Beispiel gezeigt werden.

Z.B. flag-line: Ø!Ø!O (+-*/)

Position	Zeichen im Beispiel	Bedeutung
1	Ø	Zeilenende in der Macrodefinition und im Macroaufruf (SEF)
2	!	Parameterplatzhalter in der Macrodefinition (SPF)
3	Ø	Zeilenende im Macrorumpf (MCT end-of-line flag)
4	!	Fluchtsymbol
5	O	Null
6		Leerzeichen
7	(	linke Klammer
8	+	Additionszeichen
9	-	Subtraktionszeichen
10	*	Multiplikationszeichen
11	/	Divisionszeichen
12	)	rechte Klammer

2.3.2.3 Fehlererkennung

Um die Sprachabhängigkeit des STAGE 2 zu bewahren, wurden in ihr nur zwei Möglichkeiten der Fehlererkennung eingebaut:

1. Falsche arithmetische Ausdrücke
2. Unerlaubte Parameterumformungen oder unerlaubte Kennziffern von Übersetzerfunktionen

Es werden dann die Fehlermeldungen:

xxxxxxxx ERROR IN ARITHMETIC EXPRESSION
xxxxxxxx ERROR IN CONVERSION DIGIT

auf dem Zeilendrucker ausgegeben. Der Fehlermeldung folgt die neugebildete Zeile, wie sie zum Zeitpunkt der Fehlererkennung be stand. Danach wird der Aufruf des laufenden Macros zurückverfolg und ausgegeben.

Eine weitere Fehlermeldung wird ausgegeben, wenn der dem STAGE 2 zugewiesene Speicherplatz zu gering ist.

xxxxxxxx MEMORY OVERFLOW

Dies ist ein Fehler, der zur Beendigung der Übersetzung führt, während bei den anderen Fehlern die Übersetzung mit dem nächsten Zeichen des "Code body" weitergeführt wird.

2.3.2.4 Ein/Ausgabe-Kanäle

Die Nummer der Ein/Ausgabe-Kanäle hängt von der Implementierung des STAGE 2 auf eine bestimmte Rechenanlage ab. Bei der hier vorliegenden Version (CD 6600) und in den nachfolgenden Beispielen gelten die Vereinbarungen.

Kanal 1 - Eingabe von Macrodefinitionen und zu übersetzendem Text (Kartenleser)

Kanal 3 - Ausgabe in der Zielsprache, für Maschinen lesbar (Lochkarten)

Kanal 4 - Ausgabe von Fehlermeldungen (Zeilendrucker)

2.3.2.5 Parameterumformung

a) Übertragen eines Parameters der Eingabezeile in die Code-Zeile

Format: edO

Wirkung: Bei der Bildung der Neuzeile aus einer Code-Body-Zeile werden die Zeichen edO durch eine genaue Kopie von Parameter d ersetzt

Beispiel 1:

Macro 1:

```
!=! + !Ø
   FETCH !20 !F14Ø
   ADD   !30 !F14Ø
   STORE !10 !F14Ø
 Ø
```

Hier wird noch die Übersetzerfunktion eF14 verwendet, welche bewirkt, daß die Neuzeilen nicht noch einmal dem Mustervergleich unterzogen und sofort auf dem Zeilendrucker ausgegeben werden (siehe 2.3.2.6,b).

Eingabe: A=B + CØ

Ausgabe:

```
FETCH    B
ADD      C
STORE    A
```

Beispiel 2: Komplexe Addition (bei gleichzeitiger Vereinbarung von Macro 1)

Macro 2:

```
Z!=! + !Ø
!10=!20 + !30Ø
!10+1=!20+1 + !30+1Ø
Ø
```

Eingabe: ZA=B + C Ø

Ausgabe:

```
FETCH B
ADD   C
STORE A
FETCH B+1
ADD   C+1
STORE A+1
```

Beispiel 2 zeigt, wie jede gebildete Neuzeile, wenn dies nicht ausdrücklich anders vereinbart wird (z.B. mit eF1), erneut dem Mustervergleichsprozeß unterzogen wird. Wurde die Eingabezeile als Aufruf des Macros nach Beispiel 2 erkannt, so beginnt der STAGE 2, eine Code-Zeile zu erzeugen, die aus der ersten Code-body-Zeile dieses Macros mit den entsprechenden Parametern besteht. Die Bearbeitung des Code-Body von Macro 2 wird nun unterbrochen und der STAGE 2 sucht ein der Neuzeile entsprechendes Zeilenmuster, das mit Macro 1 gegeben ist. Nun wird der Code-body von Macro 1 bearbeitet. Dort wird durch die später beschriebene Steuerfunktion eF1 verhindert, daß die gebildeten Neuzeilen wieder als Macroaufruf angesehen werden. Nach Beendigung dieses Vorgangs kehrt der STAGE 2 wieder in den Code-body von Macro 2 zurück und verfährt mit Zeile 2 genauso. Hierbei muß noch auf die Bedeutung der Leerzeichen hingewiesen werden, die dazu benutzt werden, um zwischen dem + Zeichen im Aufruf des Additionsmacros und dem in den Addressen der Imaginärteile der Variablen zu unterscheiden. Sonst entstünde Zweideutigkeit, und man erhielte nicht die erwünschten Ausgabezeilen.

b) Übertragen eines Strings aus dem Speicher des STAGE 2 in die Code-Zeile

b.1) Format: edl

Wirkung: Die Zeichen edl werden bei der Bildung einer Neuzeile durch den Inhalt des Speicherplatzes ersetzt, dessen Adresse durch Parameter d angegeben wird. Es erfolgt keine Änderung des Speicherinhalts. Ist der durch d bestimmte Speicherplatz leer oder nicht vorhanden, so wird nichts in die Neuzeile übertragen.

Beispiel:

Macro: DRUCKE DEN INHALT VON SPEICHERPLATZ ! AUS ß
IM SPEICHERPLATZ !1O STEHT !11!F14 ß
ß

Eingabe: DRUCKE DEN INHALT VON SPEICHERPLATZ SAM AUS ß

Ausgabe: (Nach Belegung von Speicherplatz Sam mit (Anton) s. später)

IM SPEICHERPLATZ SAM STEHT ANTON

Eingabe: DRUCKE DEN INHALT VON SPEICHERPLATZ JOE AUS ß

Ausgabe: (ohne vorherige Definition von JOE).

IM SPEICHERPLATZ JOE STEHT

b.2) Format: ed2

Wirkung: Wie bei ed1 wird Parameter d als Symbol für eine Adresse des STAGE 2-Speichers genommen, deren Inhalt an Stelle der Zeichen ed2 in die Neuzeile übergeben wird. Ist der betreffende Speicherplatz leer, so wird auch hier nichts in die Neuzeile eingesetzt. Wenn Parameter d jedoch nicht als Symbol einer Speicheradresse definiert ist, so wird diesem hier von STAGE 2 ein Speicherplatz zugewiesen, der mit dem laufenden Inhalt eines "Symbolgenerators" belegt wird. Dieser wird dann an Stelle von ed2 in die Neuzeile übergeben und der Inhalt des "Symbolgenerators" wird um 1 erhöht.

Dies ist eine Möglichkeit Information in den Speicher einzutragen. Eine andere Möglichkeit bietet die Übersetzerfunktion eF3.

(Weitere Möglichkeit der "Symbolerzeugung" eOm)

Beispiel: (relative Adressierung)

```
Macro:      REL ADR !=!+!ß
                    FETCH VARS + !22!F14ß
                    ADD   VARS + !32!F14ß
                    STORE VARS + !12!F14ß
```

Eingabe: REL ADR A=B+Cß

Speicherinhalt (A): 3 (Belegung mit später beschriebener Funtion
B,C undefiniert

Symbolgenerator = 0

```
Ausgabe:    FETCH VARS + 0
            ADD   VARS + 1
            STORE VARS + 3
```

Nach Behandlung des Macros:

Speicheradresse (A) : 3
Speicheradresse (B) : 0
Speicheradresse (C) : 1
Symbolgenerator = 2

c) Übertragen eines Trennzeichens in die Code-Zeile

Format: ed3

Wirkung: An Stelle der Zeichen ed3 wird das Zeichen der Eingabezeile in die Neuzeile eingesetzt, das dort Parameter d folgt. (Steht Parameter d am Ende der Eingabezeile, so soll nach /1/ nichts in die Neuzeile übertragen werden. Dies kann aber hier nach dem folgenden Beispiel nicht bestätigt werden).

Beispiel:

Macro:
```
!B!E!I!S!P!I!E!L!$
+!13!23!33!43!53!63!73!83!93+!F14$
$
```

Eingabe: /B/E/I/S/P/I/E/L/$

Ausgabe: +BEISPIEL$+

d) Behandlung eines Parameters als arithmetischer Ausdruck und Übertragen des Ergebnisses in die Code-Zeile

Format: ed4

Wirkung: An Stelle der Zeichen ed4 wird eine ganze Zahl, der ein Minuszeichen vorangehen kann, in die Neuzeile eingesetzt. Diese Zahl ist das ganzzahlige Ergebnis der Berechnung des durch Parameter d dargestellten arithmetischen Ausdrucks. Sind in diesem Ausdruck aufgerufene Speicherplätze leer oder nicht definiert, so wird an Stelle des Aufrufs bei der Berechnung des Aufrufs Null verwendet. Das Ergebnis von d wird stets abgeschnitten, so daß eine ganze Zahl ausgegeben wird. Stellt Parameter d einen unerlaubten arithmetischen Ausdruck dar, so erfolgt eine Fehlermedlung nach Abschnitt 3.3.

Beispeile:

Macro:
```
BERECHNE !$
ERGEBNIS !14!F14$
$
```

Eingabe: BERECHNE (3+15)/6$

Ausgabe: ERGEBNIS 3

Eingabe: BERECHNE 15/6$

Ausgabe: ERGEBNIS 2

Eingabe: BERECHNE 12-15$

Ausgabe: ERGEBNIS -3

Eingabe: BERECHNE APFEL+BIRNExORANGE+4$

Speicheradresse (APFEL): -21

Speicheradresse (BIRNE): 1234567890
ORANGE nicht definiert

Ausgabe: ERGEBNIS -17

Eingabe: BERECHNE UNSINN$

UNSINN nicht definiert

Ausgabe: ERGEBNIS 0

Eingabe: BERECHNE SAM$

Speicheradresse (SAM): ANTON

Ausgabe: xxxxxxxx ERROR IN ARITHMETIC EXPRESSION
ERGEBNIS
BERECHNE SAM$
ERGEBNIS SAM 0 (steht auf Lochkarte)

e) Übertragen der Länge eines Parameterstrings in die Code-Zeile

Format: ed5

Wirkung: Auch an Stelle dieser drei Zeichen wird eine ganze Zahl eingesetzt, die angibt, aus wieviel einzelnen Zeichen der Parameterstring d besteht. Besteht Parameter d aus überhaupt keinem Zeichen, so wird die Ziffer 0 eingesetzt.

Beispiele:

Macro: WELCHE LAENGE HAT DER PARAMETERSTRING !$
DER STRING BESTEHT AUS !15 ZEICHEN!F14$
$

Eingabe: WELCHE LAENGE HAT DER PARAMETERSTRING JOE$

Ausgabe: DER STRING BESTEHT AUS 3 ZEICHEN

Eingabe: WELCHE LAENGE HAT DER PARAMETERSTRING $

Ausgabe: DER STRING BESTEHT AUS 0 ZEICHEN

f) Ersetzen eines Parameters durch eine gebildete Code-Zeile

Format: ed6

Wirkung: Der Aufruf von ed6 bewirkt, daß die Neuzeile, die bis zum Aufruf von ed6 entstanden ist als aktueller Wert von Parameter d genommen wird. Dabei geht der alte Wert von Parameter d verloren. Das ed6 nachfolgende Zeichen wird überlesen. Da ed6 meistens am Ende einer Zeile steht, wird hiermit verhindert, daß unnötige Leerzeilen ausgegeben werden.

Bei Verwendung von ed6 innerhalb einer Wiederholungsschleife muß beachtet werden, daß bei Änderung von Parameter d der alte Wert jedesmal verlorengeht. (Siehe ed7 und eF1). Der jeweilige Wert von Parameter d steht also weder in der nächsten Schleife, noch nach Beendigung der ganzen Iteration zur Verfügung. Es empfiehlt sich also ed6 möglichst nicht in einer Wiederholungsschleife anzuwenden.

Beispiel:

Macro:
```
ERSETZE PARAMETER ! DURCH PARAMETER !$
!20!16$
PARAMETER 1 !10!F14$
PARAMETER 2 !20!F14$
$
```

Eingabe: ERSETZE PARAMETER ABC DURCH PARAMETER XYZ$

Ausgabe:
```
PARAMETER 1 XYZ
PARAMETER 2 XYZ
```

Erklärung: Nachdem der STAGE 2 die Eingabezeile als Aufruf des obigen Macros erkannt und ABC Parameter 1 und XYZ Parameter 2 zugeordnet hat, wird die erste Zeile des Code-body untersucht. e20 wird durch XYZ er-

setzt. Wird nun ed6 aufgerufen, so steht in der Neuzeile nur XYZ. Dies wird nun zum neuen Wert von Parameter 1 gemacht und die Neuzeile ist wieder leer. Da nun das ed6 folgende Zeichen, die Zeichenendemarke überlesen wird, kann sofort die nächste Code-body-Zeile in die Neuzeile übertragen werden, und es wird keine überflüssige Leerzeile ausgegeben.

g) "Context-gesteuerte Iteration"

Format: ed7

Wirkung: Durch den Aufruf von ed7 wird eine Wiederholungsschleife begonnen und einmal durchlaufen. Zuerst wird dabei der beim Mustervergleich Parameter d zugewiesene String gespeichert. Die Zeichen, die nach ed7 in der gleichen Code-body-Zeile stehen, werden als Trennzeichen vereinbart. Parameter d wird nun ein bezüglich Klammern ausgeglichener String zugewiesen, der am Anfang der bis zum Aufruf von ed7 entstandenen Neuzeile beginnt und bis zum ersten Trennzeichen geht, das nicht von einer zu diesem String gehörenden Klammer eingeschlossen wird. Dieser String und das folgende Trennzeichen werden nun aus der Neuzeile gestrichen, und diese wird für den nächsten Wiederholungsschritt gespeichert. Steht in dieser Neuzeile beim Aufruf eines neuen Wiederholungsschrittes nichts mehr, so wird die Iteration beendet. Nach jedem Schritt wird die Untersuchung des Code-body mit der dem Aufruf von ed7 folgenden Zeile fortgesetzt.

Steht ed7 am Ende einer Code-body-Zeile, dann sind keine besonderen Trennzeichen vereinbart. Nun wird Parameter d bei jedem Iterationsschritt nur ein Zeichen zugeordnet, das dann anschließend aus der gespeicherten Neuzeile gestrichen wird. Hier haben Klammern keine besondere Bedeutung.

Beispiel 1:

Macro:

```
KETTENADDITION !=!+!ß
      FETCH    !20!F14ß
!30!37+ß
      ADD      !30!F14ß
!F8ß
      STORE    !10!F14ß
ß
```

Die Übersetzerfunktion eF8 gibt das Ende des zu wiederholenden Bereichs an und bewirkt einen neuen Wiederholungsschritt.

Eingabe:

```
KETTENADDITION A=B+(C+D)+E+Fß
```

Ausgabe:

```
FETCH B
ADD   (C+D)
ADD   E
ADD   F
STORE A
```

Erklärung:

Das Macro wird von jeder Eingabezeile aufgerufen, bei der nach dem Gleichheitszeichen ein oder mehrere + Zeichen stehen.
Als Ergebnis des Mustervergleichs gilt die Zuordnung:

Parameter 1: A
Parameter 2: B
Parameter 3: (C+D)+E+F

Nachdem zuerst die FETCH B-Anweisung ausgegeben wurde, beginnt der Übersetzer aus der zweiten Code-body-Zeile eine Neuzeile zu bilden. Beim Aufruf von ed7 besteht diese aus

(C+D)+E+F

Durch ed7 wird diese Zeile gespeichert und das + als Trennzeichen vereinbart. Der Übersetzer beginnt nun die Neuzeile nach einem + abzusuchen. Hierbei wird ein String gebildet, der bezüglich Klammern ausgeglichen ist und vom Anfang der Neuzeile bis zum ersten nicht zu diesem String gehörenden + Zeichen geht. Dieser wird nun Parameter 3 zugeord-

net und einschließlich des folgenden Trennzeichens aus der Neuzeile gelöscht, die dann für den nächsten Wiederholungsschritts gespeichert wird.
Nun ist: Parameter 3: (C+d)
für den nächsten Schritt gespeicherte Zeile: E+F.

Der neue Wert von Parameter 3 gilt nun bis zum Aufruf von eF8 und somit wird die Zeile

ADD(C+D)

ausgegeben.
Die Funktion eF8 untersucht den Rest der gespeicherten Zeile. Ist sie leer, so wird die Wiederholung beendet. Sonst beginnt ein neuer Wiederholungsschritt, indem der Rest der gespeicherten Zeile untersucht wird.

Ergebnis: Parameter 3 : E
gespeicherte Zeile: F.
Es wird nun ADD E ausgegeben.

Im dritten Wiederholungsschritt steht kein Pluszeichen mehr in der gespeicherten Zeile und das Ende des Strings gilt als Trennzeichen.

Ergebnis: Parameter 3 : F
gespeicherte Zeile: leer.
Es wird ADD F ausgegeben. Beim erneuten Aufruf von eF8 ist die gespeicherte Zeile leer und die Wiederholung wird beendet. Als Parameter 3 wird wieder sein ursprünglicher Wert (C+D)+E+F abgespeichert.

Beispiel 2:

Macro:

```
LISTE !$
!10!17,$
!10!F14$
!F8$
$
```

```
Eingabe:    LISTE A,B,C,D$

Ausgabe:    A
            B
            C
            D

Eingabe:    LISTE A,(B,C),D$

Ausgabe:    A
            (B,C)
            D
```

Beispiel 3:

```
Macro:      AUFSPALTEN !$
            !10!17$ KEIN TRENNZEICHEN
            !10!F14$
            !F8$
            $

Eingabe:    AUFSPALTEN A,(B,C),D$

Ausgabe:    A
            ,
            (
            B
            ,
            C
            )
            ,
            D
```

h) Umwandlung von Zeichen in ganze Zahlen

Format: ed8

Wirkung: Die Zeichen ed8 werden in der erzeugten Neuzeile durch eine ganze Zahl ersetzt. Diese Zahl resultiert aus der im STAGE 2 definierte Zahlendarstellung der entsprechenden Zeichen.

Parameter d darf also nur ein einzelnes Zeichen zugeordnet werden, sonst wird eine Fehlermeldung nach 2.3.2.3 ausgegeben.

Beispiel 1:

```
Macro:      WELCHE ZAHL ENTSPRICHT !$
            !1O ENTSPRICHT !18!F14$
            $

Eingabe:    WELCHE ZAHL ENTSPRICHT A$

Ausgabe:    A ENTSPRICHT 1

            (gilt für CD 6600)

Eingabe:    WELCHE ZAHL ENTSPRICHT Q$

Ausgabe:    Q ENTSPRICHT 17

Eingabe:    WELCHE ZAHL ENTSPRICHT / $

Ausgabe:    / ENTSPRICHT 40

Eingabe:    WELCHE ZAHL ENTSPRICHT AB$

Ausgabe:    xxxxxxxx ERROR IN CONVERSION DIGIT
            AB ENTSPRICHT
            WELCHE ZAHL ENTSPRICHT AB$
            AB ENTSPRICHT
```

2.3.2.6 Übersetzerfunktionen

a) Beenden der Übersetzung

```
Format:     eFO

Wirkung:    Beim Aufruf von eFO wird die Übersetzung sofort be-
            endet. Die bis dahin gebildete Neuzeile wird nicht
            ausgegeben oder erneut untersucht.

Beispiel:
Macro:      END$
            END!F14$
            !FO$
            $
Eingabe:    END$

Ausgabe:    END     (und die Übersetzung wird beendet)
```

b) Sofortige Ausgabe einer Zeile (ohne nochmalige Untersuchung

b.1) Format: eFlm

Wirkung: Wenn die bis zum Aufruf von eFlm gebildete Neuzeile nicht leer ist, wird diese auf Kanal m ausgegeben, ohne daß sie vom Übersetzer nochmals untersucht wird. Hierbei muß eFlm am Ende der Code-body Zeilen stehen, da deren Rest überlesen wird.
Ist die gebildete Neuzeile beim Aufruf von eFlm noch leer, so wird dieser Aufruf als Forderung nach formatgebundener Ausgabe angesehen. Auch hier wird der Rest der Zeile überlesen. Die nächste Zeile wird dann als Muster für das geforderte Ausgabeformat genommen, wobei durch jede Zifferkette ein Feld definiert wird, in das die den Ziffern entsprechenden Parameter übertragen werden. Ist ein aktueller Parameter hierbei länger als das für ihn definierte Feld, so wird er auf der rechten Seite abgeschnitten; ist er kürzer, so wird das Feld rechts mit Leerzeichen aufgefüllt. Sind die in dieser Musterzeile stehenden Zeichen keine Ziffern, so werden diese unverändert ausgegeben. Stehen zwischen zwei Ketten aus lauter gleichen Ziffern andere Zeichen, so heißt dies, daß für den entsprechenden Parameter zwei Felder definiert werden, wobei der Parameter in jedes Feld von vorn eingetragen wird.
Gibt es bei der Verwendung von eFlm als Aufruf für formatgebundene Ausgabe nach diesem Aufruf keine weiteren Code-body-Zeilen, so wird "Conversion error" ausgegeben und die Übersetzung des laufenden Code-body beendet. Hiernach wird das aufrufende Macro (oder der Eingabetext, falls kein solches vorhanden ist) weiterübersetzt.

Die Angabe des Kanals m kann weggelassen werden. Dann wird Kanal 3 als Ausgabekanal verwendet.

Beispiele:

Macro: DIREKTE AUSGABE$
DIESE ZEILE WURDE NICHT WEITER UNTERSUCHT!F14$
$

Eingabe: DIREKTE AUSGABE$

Ausgabe: DIESE ZEILE WURDE NICHT WEITER UNTERSUCHT

Macro: GIB AUS !$
!10!F14$
$

Eingabe: GIB AUS A$

Ausgabe: A

Macro: AUSGABE$
DIREKTE AUSGABE AUF LOCHKARTE!F1$
$

Eingabe: AUSGABE$

Ausgabe: DIREKTE AUSGABE AUF LOCHKARTE
(Wird nur auf Lochkarte ausgegeben)

Macro: FORMAT !,!,!$
!F14$
111111 222222 333333$

Eingabe: FORMAT OTTO,ANTON, VIEL ZU LANG$

Ausgabe: OTTO ANTON VIEL Z

Macro: ZWEI MAL !,!$
!F14$
111111 111111 22222 22222$
$

Eingabe: ZWEI MAL BEIDE PARAMETER, SIND ZU LANG$

Ausgabe: BEIDE BEIDE SIND SIND

Macro: MARKE!$
!F14$
111111 MARKE×$
$

Eingabe: MARKE ZIEL$

Ausgabe: ZIEL MARKE×

b.2) Format: eFlmc

Wirkung: Wie bei eFlm, aber Kanal m steht bereits zur Verfügung, ehe begonnen wird, die Zeile zu schreiben.
Das eFlmc direkt folgende Zeichen wird überlese

Beispiel:

Macro: STARTE KANAL 4$
REWOUND!F14F$
$

Eingabe: STARTE KANAL 4$

Ausgabe: REWOUND
(dies ist die erste Zeile auf Kanal 4)

c) Wechsel der Ein/Ausgabekanäle und Übergabe von Eingabetext an bestimmte Ausgabekanäle

c.1). Format: meF2n

Wirkung: Kanal m wird zum momentanen Eingabekanal gemacht, und wenn Parameter 1 (resultiert aus dem Mustervergleich) nicht Null ist, wird der eingelesene Text an Kanal n übergeben. Dies geschieht dann solange bis der Übersetzer auf eine Eingabezeile trifft, die mit Parameter 1 beginnt. Diese Zeile wird überlesen, und über den momentanen Eingabekanal wird nun die nächste Zeile eingelesen. Die Textübergabe kann auch durch ein "end file" auf dem Eingabekanal beendet werden.

Hat Parameter 1 den Wert Null, dann wird kein Text übergeben und Kanal m wird zum neuen Eingabekanal. Die Zeichen m oder n können auch weggelassen werden. Wenn eF2 dann am Beginn einer Zeile steht, bleibt der momentane Eingabekanal erhalten, und wenn es am Ende steht wird Text auf Kanal 3 ausgegeben.

Die Zeichen meF2n müssen alleine in einer Codebody-Zeile stehen.

Beispiele:

Macro:
```
UEBERSPRINGE BIS !ß
!F20ß KANAL O IST NUR ATTRAPPE
ß
```

Eingabe:
```
UEBERSPRINGE BIS KOMMENTARENDEß
DIESER TEXT WUERDE
NACH KANAL O UEBERGEBEN
KANAL O IST ABER NUR ATTRAPPE
KOMMENTARENDE
DIESE ZEILE WIRD WIEDER AUSGEGEBEN
```

Ausgabe:
```
DIESE ZEILE WIRD WIEDER AUSGEGEBEN
```

Macro:
```
UEBERGEBE ! VON KANAL !ß
!20!F24ß
ß
```

Eingabe:
```
UEBERGEBE     EOF VON KANAL 1ß
```

Kanal 1:
```
UNSINN
EOF
```

Ausgabe:
```
UNSINN
```

c.2) Format mceF2nc

Wirkung: Die gleiche wie bei meF2n, nur daß die beiden Kanäle bereits vor Beginn der Textübergabe zur Verfügung stehen. Das mceF2nc folgende Element wird überlesen, und erst das nächste Element des Code-body wird wieder untersucht. Auch hier können die Zeichen m,c und n weggelassen werden.

Beispiel:

Macro:
```
COPY TO 4 REWOUND UNTIL!ß
!F24Rß
ß
```

Eingabe:
```
COPY TO 4 REWOUND UNTIL END DATAß
A
NUR WENN END DATA AM
ZEILENANFANG STEHT WIRD
DIE UEBERGABE BEENDET. ENDFILE
END DATA STEHT AM ZEILENANFANG
```

```
Ausgabe:        A
(Kanal 4)       NUR WENN END DATA AM
                ZEILENANFANG STEHT WIRD
                DIE UEBERGABE BEENDET.

Macro:          DRUCKE BIS ! VON 4 REWOUND$
                4R!F24$
                $

Eingabe:        DRUCKE BIS ENDFILE VON 4 REWOUND
```

(Kanal 4 enthält die Ausgabe des vorigen Beispiels)

```
Ausgabe:        A
                NUR WENN END DATA AM
                ZEILENANFANG STEHT WIRD
                DIE UEBERGABE BEENDET
```

(Ausgabe auf Kanal 4 Zeilendrucker)

d) Ablegen von Information im STAGE 2-Speicher

Format: eF3

Wirkung: Mit Parameter 1 der Eingabezeile wird ein Speicherplatz adressiert und Parameter 2 wird zu dessen neuen Inhalt. Der alte Inhalt wird gelöscht.
Das eF3 direkt folgende Element wird bei der weiteren Untersuchung des Code-body überlesen

Beispiel:

```
Macro:          ! EQU !$
                !F3$
                $

Eingabe:        OBST EQU ORANGE$

Ausgabe:        Diese Anweisung bewirkt keine Ausgabe.
```

Speicheradresse (OBST): ORANGE

```
Eingabe:        SAM EQU ANTON$
                A EQU 3$
                APFEL EQU -21$
                BIRNE EQU 1234567890$
                N EQU 1$
                D EQU 7$
                C EQU 5$
```

e) Unbedinger Sprung

Format: eF4

Wirkung: Parameter 1 wird als arithmetischer Ausdruck betrachtet und sein Ergebnis wird in einem Sprungzähler abgelegt. Ist Parameter 1 kein zulässiger arithmetischer Ausdruck, erfolgt eine Fehlermeldung.
Das eF4 direkt folgende Element wird bei der weiteren Untersuchung des Code-body überlesen.

Beispiele:

Macro 1:

```
SKIP !$
!F4$
$
```

Macro 2:

```
TEST SKIP !,!,!$
SKIP !10$
 AB !F14$
 CD !F14$
 EF !F14$
 SKIP !20$
 GH !F14$
 IJ !F14$
 KL !F14$
 SKIP !30$
 MN !F14$
 OP !F14$
 QR !F14$
 ST !F14$
 UV !F14$
 $
```

Eingabe:

```
TEST SKIP 2,N,N+2$
```

Speicheradresse (N): 1

Ausgabe:

```
EF
IJ
KL
ST
UV
```

Eingabe: TESTSKIP 6,2,3Ø

Ausgabe: KL
ST
UV

(Der zweite Aufruf des Skip-Macro wird durch den ersten übersprungen)

f) Bedingter Sprung bei Gleichheit zweier Strings

Format: eF5K

Wirkung: Parameter 1 und 2 werden auf Gleichheit untersucht. Wenn K = 0 und die Parameter 1 und 2 gleich sind, wird der Sprungzähler mit Parameter 3 geladen. Ist K = 1, so wird der Sprungzähler mit Parameter 3 geladen, wenn die Parameter 1 und 2 ungleich sind.
Das eF5K direkt folgende Element wird bei der weiteren Untersuchung des Code-body überlesen. Die Parameter 1 und 2 können beliebige Strings sein, aber Parameter 3 muß ein zulässiger arithmetischer Ausdruck sein, sonst erfolgt eine Fehlermeldung.

Beispiele:

Macro 1:
```
IF ! = ! SKIP !Ø
!F5ØØ
Ø
```

Macro 2:
```
IF ! NE ! SKIP !Ø
!F51Ø
Ø
```

Macro 3:
```
TEST EF5 !,!,!Ø
IF !10 = !20 SKIP !30Ø
AB !F14Ø
CD !F14Ø
EF !F14Ø
GH !F14Ø
IF !10 NE !20 SKIP !30Ø
IJ !F14Ø
KL !F14Ø
MN !F14Ø
OP !F14Ø
Ø
```

Eingabe: TEST EF5 ANTON,OTTO,(1+2)$

Ausgabe:
AB
CD
EF
GH
OP

Eingabe: TEST EF5 ANTON,ANTON,2+1$

Ausgabe:
GH
IJ
KL
MN
OP

g) Bedingter Sprung nach Vergleich zweier arithmetischer Ausdrücke

Format: eF6K

Wirkung: Parameter 1 und 2 werden als arithmetische Ausdrücke betrachtet und ihre Ergebnisse miteinander verglichen. Abhängig vom Zeichen K und der relativen Beziehung der Werte von Parameter 1 und 2 wird der Sprungzähler mit Parameter 3 geladen. Bedingungen für das Setzen des Sprungzählers sind:

K = - Parameter 1 < Parameter 2
K = 0 Parameter 1 = Parameter 2
K = 1 Parameter 1 ≠ Parameter 2
K = + Parameter 1 > Parameter 2

Das eF6K direkt folgende Element wird bei der weiteren Untersuchung des Code-body überlesen. Alle Parameter müssen gültige arithmetische Ausdrücke sein, sonst erfolgt eine Fehlermeldung

Beispiele:

Macro 1:

```
IF ! LT ! SKIP !$
!F6-$
$
```

Macro 2:

```
IF ! LE ! SKIP !$
IF !10 LT !20 SKIP !30+1$
!F60$
$
```

Macro 3:

```
IF ! GT ! SKIP !$
!F6+$
$
```

Macro 4:

```
IF ! GE ! SKIP ! $
IF !10 GT !20 SKIP !30+1$
!F60$
$
```

Macro 5:

```
TEST EF6K !,!,!$
IF !10 LT !20 SKIP !30$
AB !F14$
CD !F14$
EF !F14$
IF !10 LE !20 SKIP !30$
GH !F14$
IJ !F14$
KL !F14$
IF !10 GT !20 SKIP !30$
MN !F14$
OP !F14$
QR !F14$
IF !10 GE !20 SKIP !30$
ST !F14$
UV !F14$
XY !F14$
$
```

Eingabe: TEST EF6K 12,12,2Ø

Ausgabe:
AB
CD
EF
KL
MN
OP
QR
XY

Eingabe: TEST EF6K 2,28,D-CØ

Speicheradresse (D): 7

Speicheradresse (C): 5

Ausgabe:
EF
KL
MN
OP
QR
ST
UV
XY

h) "Zähler-gesteuerte Iteration"

Format: eF7

Wirkung: Die bis zum Aufruf von eF7 gebildete Neuzeile wird als arithemtischer Ausdruck berechnet, und sein Wert wird in einen Iterations-Zähler geladen. Ist dieser Wert ungleich Null, so wird eine Iteration ausgelöst und einmal durchgeführt. Ist der Wert Null, so geschieht nichts. Auf jeden Fall wird bei der weiteren Untersuchung des Code-body das eF7 folgende Element überlesen.

Eine Zähler-gesteuerte Iteration beginnt damit, daß der Wert des Iterationszählers gespeichert wird und wird fortgesetzt, indem der Wert des Iterationszählers um 1 erniedrigt wird. Wenn

der Wert des Zählers nicht negativ ist, wird die dem Aufruf von eF7 folgende Code-body-Zeile untersucht.
Ist die bis zum Aufruf von eF7 gebildete Zeile kein gültiger arithmetischer Ausdruck, erfolgt eine Fehlermeldung.

Beispiele:

Macro: AUSGABE VON ! ZEILEN, BEGINNEND MIT x$
!10!F7$
x!F14$
!F8$
$

Eingabe: AUSGABE VON 5 ZEILEN, BEGINNEND MIT x$

Ausgabe: x
x
x
x
x

Eingabe: AUSGABE VON N-2 ZEILEN, BEGINNEND MIT x$

Speicheradresse (N): 3

Ausgabe: x

i) Auslösung eines neuen Iterationsschritts

Format: eF8

Wirkung: Für die laufende Iteration (Context- oder Zählergesteuert) wird ein neuer Schritt ausgelöst. Auch hier wird bei der weiteren Untersuchung des Code-body das eF8 folgende Zeichen überlesen.
Wird eF8 am Anfang einer Zeile aufgerufen, während Zeilen übersprungen werden, so wird die laufende Iteration beendet. Soll eine Iterationsschleife nicht verlassen, sondern nur um sie herumgesprungen werden, so müssen besondere Maßnahmen getroffen werden. Während des Springens soll die Zahl der Wiederholungsschritte

beibehalten werden. Diese Zahl wird durch jedes eF7 am Anfang einer Zeile um eins erhöht und durch jedes eF8 am Anfang einer Zeile erniedrigt.
Die laufende Iteration wird nur beendet, wenn eF8 am Anfang einer Zeile steht und die Zahl der Wiederholungsschritte 0 ist. Dabei ist zu beachten, daß wenn eF7 am Anfang einer Zeile steht, die neu gebildete Zeile noch leer ist und daher den Wert 0 hat. D.h. der Aufruf einer Zähler-gesteuerten Iteration hat keine Wirkung.

Beispiel:

Macro:

```
SPALTE AB !ß
!10!17 ,ß
IF !13 = , SKIP 5ß
!F7ß
!10!27ß
!20!F14ß
!F8ß
SKIP 1ß
!10!F14ß
!F8ß
ß
```

Eingabe:

```
SPALTE AB ABC,XYZ JONES,14ß
```

Ausgabe:

```
ABC
X
Y
Z
JONES
1
4
```

Erklärung:

Die erste Code-body-Zeile ist ein Aufruf für Context-gesteuerte Iteration. Beim Aufruf von ed7 enthält die gebildete Zeile ABC, XYZ JONES, 14 und das Leerzeichen und das Komma werden als Trennzeichen vereinbart. Der Übersetzer sucht nun diese Zeile nach dem ersten Trennzeichen ab und spaltet ABC als Parameter 1 von ihr ab.

Die nächste Zeile vergleicht das dem neuen Parameter 1 in der Eingabezeile folgende Zeichen mit , . Hier besteht Gleichheit und die nächsten 5 Code-body-Zeilen werden übersprungen. Die Iteration wird wegen des Überspringens von eF8 nicht beendet, da zuvor durch eF7 die Zahl der Wiederholungsschritte erhöht wurde, d.h. insgesamt bleibt sie erhalten. Es wird der laufende Parameter 1 ABC ausgegeben. Das folgende eF8 markiert das Ende der äußeren Schleife und bewirkt einen neuen Iterationsschritt.

In diesem Schritt wird nun bis zum nächsten Trennzeichen XYZ von der gespeicherten Zeile als Parameter 1 abgespalten. XYZ folgt in der Eingabezeile ein Leerzeichen, und der Sprungbefehl der zweiten Code-body-Zeile wird nicht ausgeführt. Der Aufruf von eF7 hat keine Wirkung, da die gebildete Zeile in diesem Augenblick noch leer ist. Die nächste Zeile ist wieder ein Aufruf für Context-gesteuerte Iteration. Als Neuzeile, über die die Iteration laufen soll, wird XYZ gespeichert. Da für die innere Iterationsschleife kein besonderes Trennzeichen vereinbart ist, wird durch diese in jedem Schritt ein Zeichen als Parameter 2 abgespalten und ausgegeben. Nach der Ausgabe von Z ist die gespeicherte Zeile leer, und durch eF8 wird die innere Iteration beendet. Mit der nächsten Code-body-Zeile wird die Ausgabe von Parameter 1 übersprungen, und das letzte eF8 bewirkt wieder einen neuen äußeren Iterationsschritt. Dies wiederholt sich solange, bis auch die für die äußere Iteration gespeicherte Zeile leer ist.

j) Beenden der Bearbeitung des laufenden Macro

Format: eF9

Wirkung: Die Untersuchung des laufenden Macros wird sofort beendet und der Übersetzer kehrt zum aufrufenden Macro zurück. Die bis dahin gebildete Zeile geht verloren. eF9 kann bei bedingten Sprüngen verwendet werden, wenn nur aus dem laufenden Macro herausgesprungen, nicht aber Anweisungen des aufrufenden Macros übersprungen werden sollen.

Beispiel:

Macro:
```
TEST EF9ø
HILFSZEILE!F14ø
DIESE ZEILE GEHT VERLOREN!F9ø
DIESE ZEILE WIRD NICHT UNTERSUCHT!F14ø
ø
```

Eingabe: TEST EF9ø

Ausgabe: HILFSZEILE

k) Zurückverfolgen von Fehlern

Format: eFE

Wirkung: Da dies eigentlich eine ungültige Übersetzerfunktion ist, wird auf Kanal 4 eine Fehlermeldung gegeben, der die bis dahin gebildete Neuzeile folgt. Dann wird der Fehler weiterzurückverfolgt, und der Aufruf des laufenden Macros, der Aufruf des diesen Aufruf bewirkenden Macros usw. werden ausgegeben. Die letzte Ausgabezeile ist die Eingabezeile. Auch hier wird das eFE direkt folgende Element bei der weiteren Untersuchung des Codebody überlesen.

Beispiel:

Macro:
```
!ø
DIE FOLGENDE ZEILE WURDE NICHT ERKANNT:!FE!F9ø
ø
```

Eingabe: UNSINN$

Ausgabe: xxx ERROR IN CONVERSION DIGIT
DIE FOLGENDE ZEILE WURDE NICHT ERKANNT.
UNSINN

2.4 Rechenzeituntersuchungen für den Macroübersetzer STAGE 2

Um die Rechenzeit für eine Sprachübersetzung mit dem Macroübersetzer STAGE 2 optimieren und um Aufschlüsse über die Schnelligkeit dieses Übersetzers bekommen zu können, wurde am Rechenzentrum der Universität Stuttgart auf einer CDC 6600 das Zeitverhalten des Macroübersetzers STAGE 2 bei der Baumerstellung und der Übersetzung von Macroaufrufen untersucht. Die Baumerstellung ist deshalb von Bedeutung, weil nicht bei jedem Übersetzerlauf alle definierten Macros aufgerufen werden, diese aber in die Baumerstellung einbezogen werden.

Die folgenden Aussagen wurden aus den Rechenzeiten im Kernspeicher abgeleitet, die der STAGE 2 zur Bearbeitung einfach gestalteter Macrodefinitionen und -aufrufe benötigte. Ähnliche und weitergehende Hinweise zum Aufbau des Macroteils hätte man auch durch Untersuchung des STAGE 2 - Programms erhalten können. Bei der Deutung der im folgenden dargestellten Diagramme muß man berücksichtigen, daß sie aus Durchschnittsrechenzeiten aufgestellt wurden. Dies wurde nötig, da in die von der CDC 6600 ausgedruckte "Central-processor"(CP)-Zeit für ein Programm stets auch ein unterschiedlicher Rechenzeitanteil eingeht, der durch Programmunterbrechungen durch andere Programme mit höherer Priorität verursacht wird.

Im Einzelnen wurden dabei folgende Punkte untersucht:

2.4.1 Abhängigkeit der Baumerstellungszeit

1) Von der Zahl der Macros (siehe Bild 5).
   Die Rechenzeit steigt proportional zur Erhöhung der Zahl der Macros an. Der Ordinatenabschnitt entsteht aus Baumerstellungszeit und der Übersetzungszeit für ein benötigtes END-Macro, ohne das der Übersetzer einen Fehler meldet.

2) Von der Länge der Macroköpfe.

Eine Verlängerung der Macroköpfe auf das Zehnfache zeigte keinen erkennbaren Einfluß auf die Baumerstellungszeit.

3) Von der Zahl der Codezeilen im Macrokörper.
(Die Codezeilen sollen nicht als erneute Macroaufrufe zugelassen sein) (siehe Bild 5).
Nach Erhöhung der Zahl der Codezeilen ergab sich ein steilerer Anstieg der Rechenzeit mit der Zahl der Macros als bei der Baumerstellung für Macros ohne Codezeilen, und das Ergebnis läßt vermuten, daß die Rechenzeit ausgehend von der Baumerstellungszeit für Macros ohne Codezeilen linear mit der Zahl der Codezeilen ansteigt.

4) Von der Länge der Codezeilen.
Hier konnte kein Einfluß auf die Rechenzeit festgestellt werden.

5) Von der Zahl der Parameterplatzhalter im Macrokopf.
Wie das Ergebnis zeigt, werden Parameterplatzhalter im Macrokopf offensichtlich wie andere Zeichen behandelt. So konnte auch keine Beeinflussung der Rechenzeit durch die Veränderung ihrer Anzahl festgestellt werden.

6) Vom Vorhandensein von Elementen vom Typ 2 (Parameterumformung) im Macrokörper.
Grundsätzlich läßt das Ergebnis erkennen, daß das Vorhandensein solcher Elemente einen Anstieg der Baumerstellungszeit gegenüber Macros mit einfachen Textzeichen bewirkt. Die Anzahl der Typ-2-Elemente in einer Codezeile ging in die Rechenzeit nicht ein.

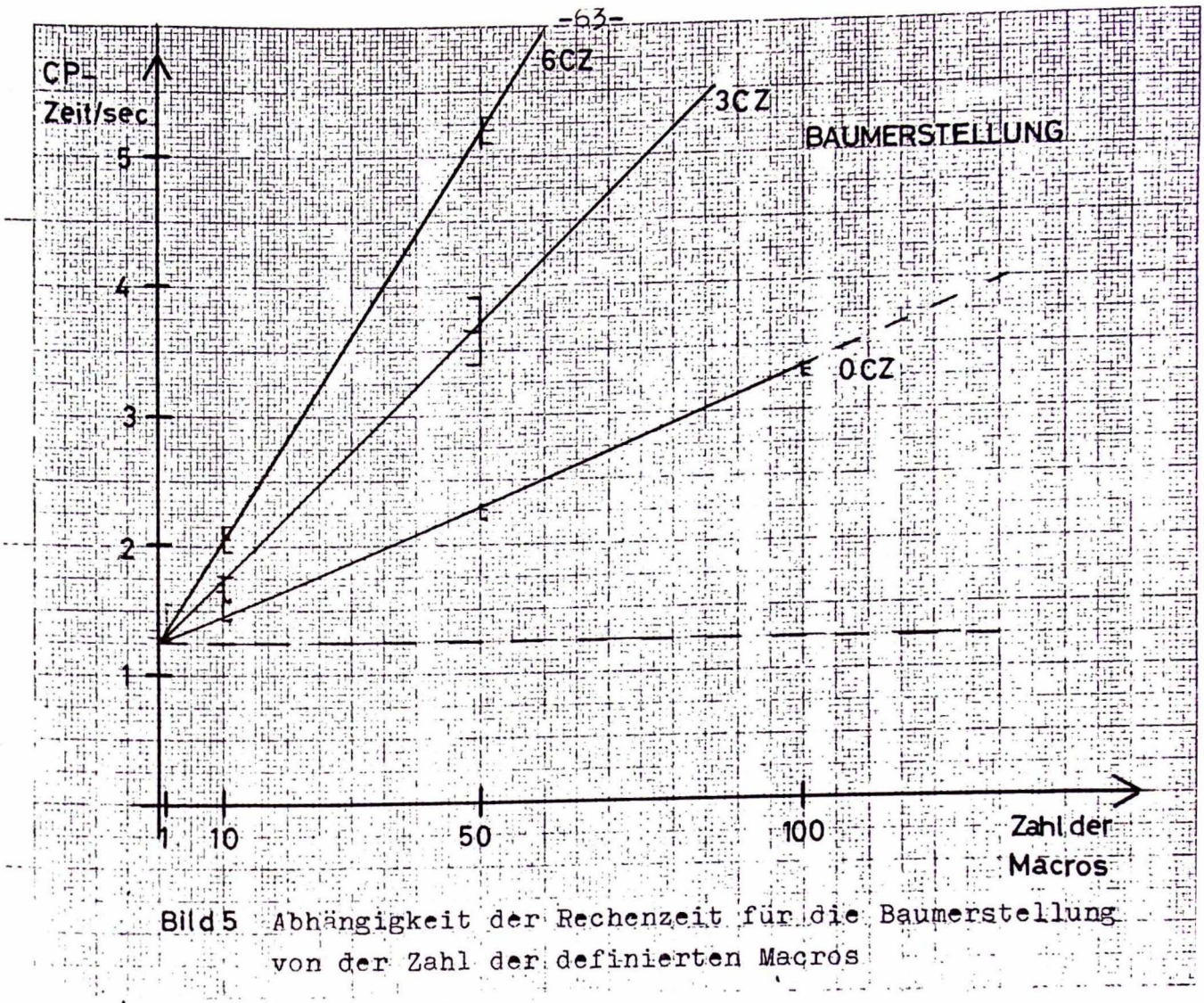


Bild 5 Abhängigkeit der Rechenzeit für die Baumerstellung von der Zahl der definierten Macros

CZ: Codezeilen

[: Bereich der Schwankungen der Rechenzeit

2.4.2 Abhängigkeit der Übersetzungszeit

1) Von der Zahl der Macros (siehe Bild 6).
Man erhält hier, wenn stets das gleiche Macro (bezüglich der Reihenfolge) aufgerufen wird, eine lineare Abhängigkeit der Übersetzungszeit von der Zahl der Macroaufrufe. Der Verlauf der Rechenzeit entspricht also einer Geraden parallel zu der von 2.4.1,1. (Bild 6 Gerade für 0 Aufrufe).

2) Von der Länge der Macroköpfe (Bild 6).
Wie Bild 6 zeigt wird für lange Macroköpfe eine größere Übersetzungszeit als für kurze Macroköpfe nach 2.4.2,1 (Verhältnis 5:1) benötigt. Dieses Ergebnis läßt sich auch aus dem Suchalgorithmus des STAGE 2 ableiten. Es muß eine größere Zahl von Eingabezeichen verglichen werden. Die Zahl der definierten Macros hat wiederum keinen Einfluß auf die Übersetzungszeit.

3) Von der Zahl der Codezeilen (Bild 6).
Diese beeinflußt die Übersetzungszeit offenbar nicht. Man erhält Geraden, die zu den in 2.4.1,3 erhaltenen (Bild 6 - 0 Aufrufe) etwa im gleichen Abstand parallel laufen wie die Gerade von 2.4.2,1 zu der von 2.4.1,1.

4) Von der Stellung des aufgerufenen Macros.
Der Versuch ergab für den Aufruf des letzten definierten Macros eine erkennbar höhere Rechenzeit als für den Aufruf des ersten. Diese Abhängigkeit läßt sich auch aus dem Ablauf des Mustervergleichs ableiten.

5) Von der Zahl der Macroaufrufe (Bild 7).
Hier konnte ein linearer Zusammenhang zwischen Rechenzeit und der Zahl der Macroaufrufe (bei gleichbleibender Zahl definierter Macros) festgestellt werden.

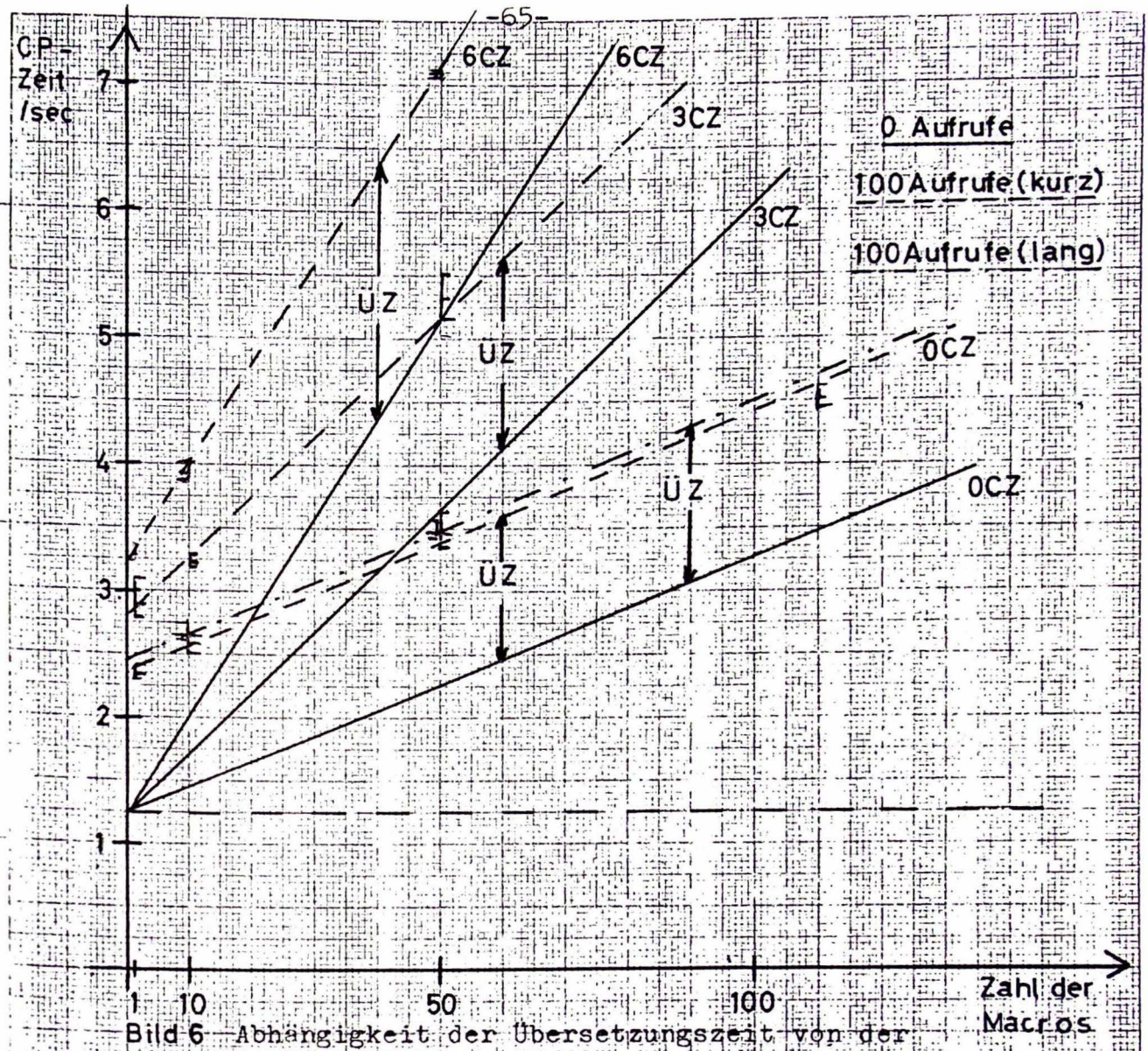


Bild 6 Abhängigkeit der Übersetzungszeit von der Zahl der Macros.

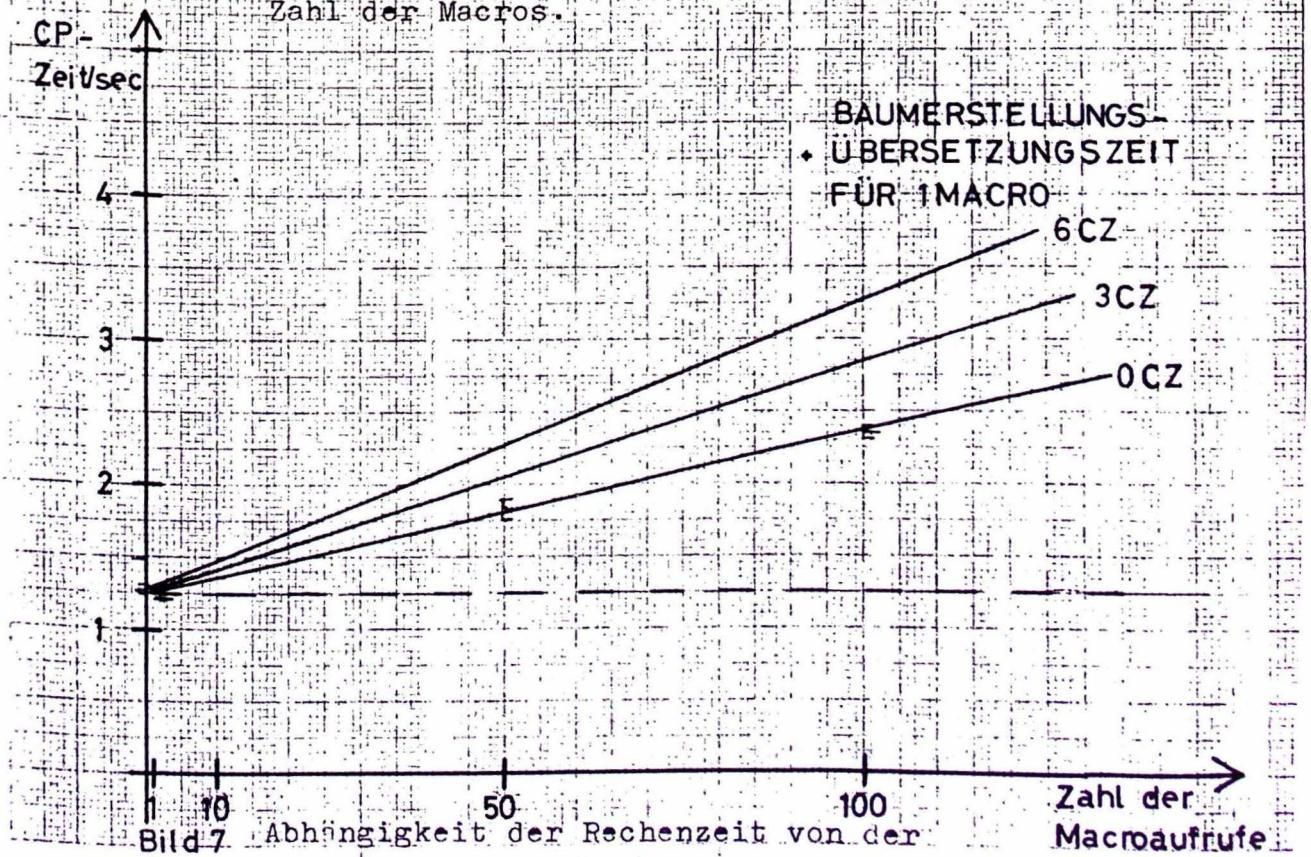


Bild 7 Abhängigkeit der Rechenzeit von der Zahl der Macroaufrufe

2.4.3 Macroschachtelung (Bild 8).

Das Ergebnis zeigt hier, daß sich die Rechenzeit bei mehreren Macros mit gleichem Macrokörper durch Verwendung eines Aufrufmacros für diesen Körper deutlich verringern läßt. Der Anstieg der Rechenzeit mit der Zahl der Macros, der wesentlich durch die Baumerstellung bestimmt wird, verläuft flacher und für unterschiedliche Codezeilenzahl nahezu parallel zu einer Geraden, die dem Rechenzeitverlauf für Macros mit 1 Codezeile entspricht.

2.4.4 Parameterlänge (Bild 9).

Hier wurde untersucht, ob es günstiger ist, einen Block von Zeichen eines Macroaufrufs mit einem langen oder mehreren kurzen Parametern in ein Aufrufmacro im Macrokörper übergeben. Man erhielt für einen langen Parameter einen etwas flacheren Verlauf der Rechenzeitgeraden als bei 9 kurzen Parametern.

2.4.5 Vergleiche verschiedener Übersetzerversionen.

1) Es wurden zwei STAGE 2-Versionen an Hand eines von Waite/1/ erstellten Testprogramms (liegt am IVD vor) zur Verdeutlichung der Funktionen des STAGE 2 miteinander verglichen. Die auf dem FTN-Compiler ablauffähige Version benötigte hierzu 2.0 sec, während für die auf dem RUN-Compiler ablauffähige Version 2.8 sec benötigt wurden (CDC 6600).

2) Als Möglichkeit zur Abschätzung der Schnelligkeit des Macroübersetzers STAGE 2 existiert der Vergleich zu einem von Herrn Mühlhan Firma ESG, München in FORTRAN erstellten Programms zur Übersetzung von 650 CIMIC-Befehlen (liegt am IVD vor). Hierzu wurden auf der verwendeten Rechenanlage (SIEMENS 404) 60 sec benötigt, während dieser Vorgang mit einer in Assembler geschriebenen Version des STAGE 2 280 sec dauerte. Die FORTRAN-Version des STAGE 2 wäre noch um den Faktor 3 langsamer und würde auch den dreifachen Speicherplatz benötigen.

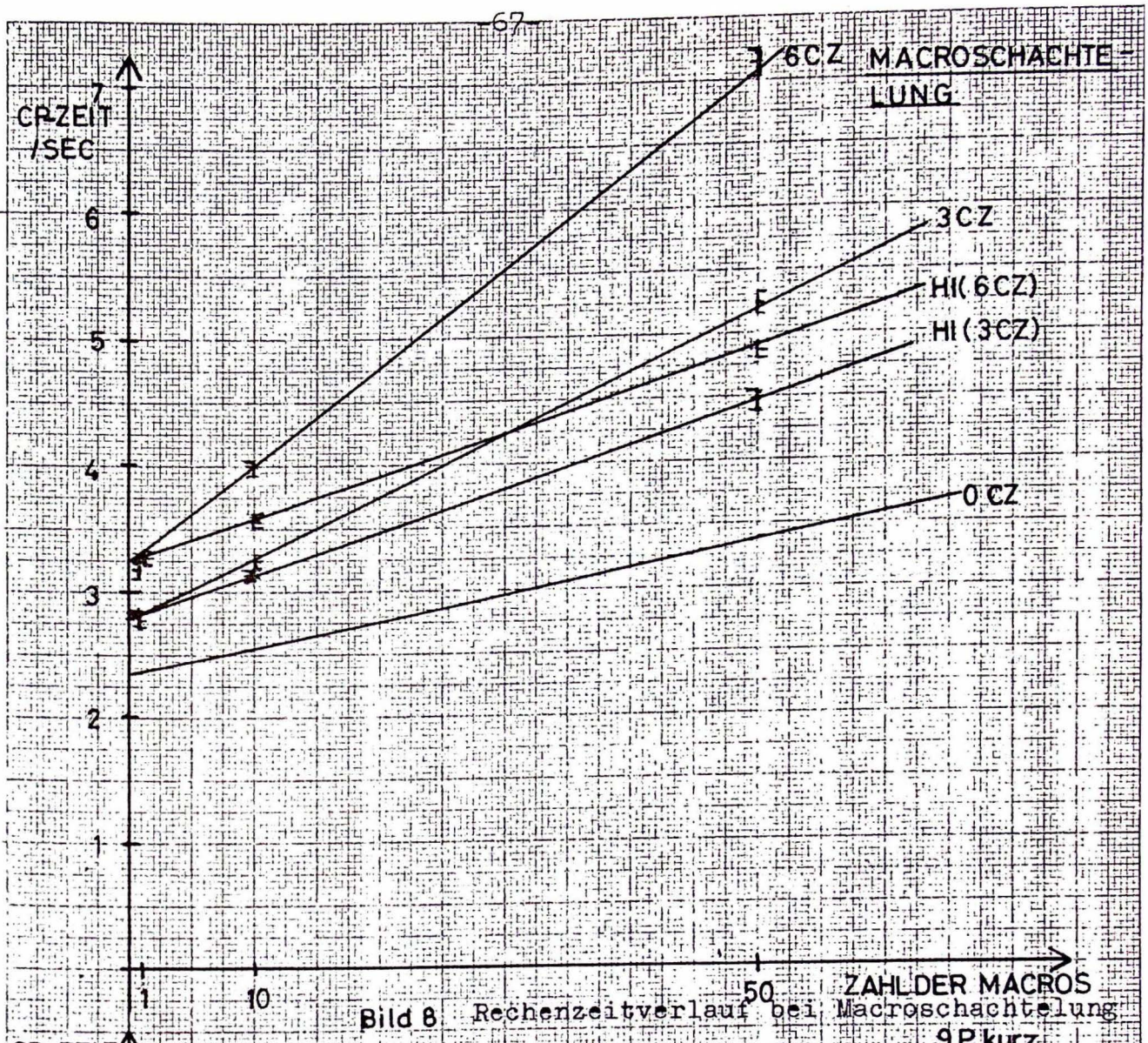


Bild 8 Rechenzeitverlauf bei Macroschachtelung

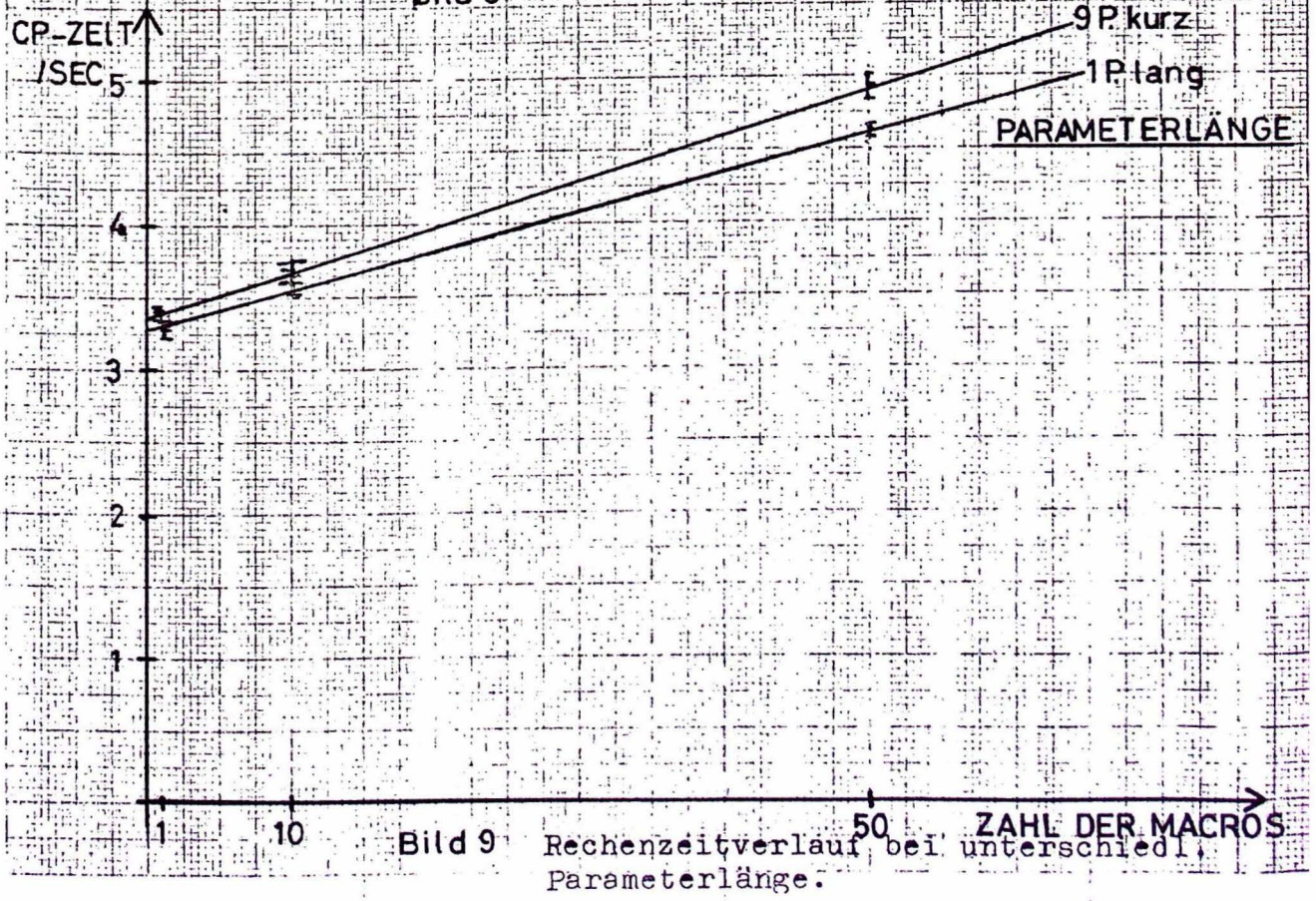


Bild 9 Rechenzeitverlauf bei unterschiedl. Parameterlänge.

Bei diesem Vergleich muß man jedoch berücksichtigen, daß der Aufwand zum Schreiben des FORTRAN-Übersetzerprogramms etwa 3 Wochen betrug, und hierbei der zu erzeugende Code bereits durch einen STAGE 2 - Lauf bekannt war.

2.4.6 Ergebnisse der Rechenzeituntersuchungen

a) Baumerstellung:

kurze Köpfe (0-Codezeilen):

	END	1 Macro	10 Macros	50 Macros	100 Macros
CP-Zeit/ sec	1.312	1.253	1.408	2.244	3.284
	1.318	1.417	1.747	2.276	3.351
	1.411	1.539	1.604	2.244	3.324
Ø / sec	1.347	1.403	1.586	2.255	3.320

lange Köpfe (mit Blanks)

CP-Zeit/ sec	1.447	2.340
	(1.408)	(2.278)
	1.493	2.275
	(1.492)	(2.402)
	1.435	2.322
	(1.487)	(2.275)
Ø / sec	1.458	2.312
	(1.462)	(2.318)

mit 3 Codezeilen / kurzer Kopf

CP-Zeit/ sec	1.688	3.722
	1.716	3.875
	1.677	3.374
Ø / sec	1.694	3.657

mit 6 Codezeilen / kurzer Kopf

CP-Zeit/ sec	2.043	5.141
	1.993	5.264
	2.151	5.200
Ø / sec	2.062	5.206

b) Übersetzungszeit:

kurzer Kopf / 0 Codezeilen / 100 Aufrufe:

	1 Macro	10 Macros	50 Macros	100 Macros	
CP-Zeit/ sec	2.311	2.613	3.660	4.356	/ 4.467
	2.333	2.543	3.442	4.511	/ 4.568
	2.385	2.655	3.373	4.466	/ 4.446
Ø / sec	2.343	2.604	3.495	4.444	/ 4.527 Aufruf des letzten Macros

kurzer Kopf / 3 Codez. / 100 Aufrufe

CP-Zeit/ sec	2.773	3.195	5.502
	2.782	3.230	5.234
	3.134	3.279	5.275
Ø / sec	2.869	3.235	5.337

kurzer Kopf / 6 Codez. / 100 Aufrufe

CP-Zeit/ sec		3.904	7.081
		3.884	7.072
		3.947	7.065
Ø / sec		3.906	7.073

langer Kopf / 10 Codez. / 100 Aufrufe

CP-Zeit/ sec		2.630	3.424
		2.636	3.497
		2.647	3.564
Ø / sec		2.638	3.495

1 Macro / kurzer Kopf

	1 Aufruf	50 Aufrufe	100 Aufrufe
CP-Zeit/ sec	1.225	1.922	2.311
	1.253	1.750	2.333
	1.207	1.844	2.385
Ø / sec	1.228	1.839	2.343

c) Macroschachtelung / 100 Aufrufe

3 Codez. / Hilfsmacro

	1 Macro	10 Macros	50 Macros
CP-Zeit/ sec	2.842	3.118	4.390
	2.841	3.165	4.346
	2.848	3.128	4.345
Ø / sec	2.844	3.137	4.360

3 Codez. / ohne Hilfsmacro

CP-Zeit/ sec	2.749	3.195	5.502
	2.733	3.230	5.234
	2.767	3.279	5.275
Ø / sec	2.761	3.235	5.337

6 Codez. / Hilfsmacro

CP-Zeit/ sec	3.319	3.579	4.757
	3.264	3.545	4.779
	3.244	3.558	4.769
Ø / sec	3.279	3.561	4.768

6 Codez. / ohne Hilfsmacro

CP-Zeit/ sec	3.197	3.884	7.081
	3.135	3.947	7.072
	3.200	3.888	7.065
Ø / sec	3.186	3.906	7.073

d) Parameterlänge

1 langer Parameter

CP-Zeit/ sec	3.281	3.525	4.740
	3.271	3.761	4.813
	3.236	3.651	4.805
Ø / sec	3.263	3.648	4.786

9 kurze Parameter

CP-Zeit/ sec	3.346	3.657	4.971
	3.375	3.647	5.143
	3.404	3.674	5.049
Ø / sec	3.375	3.659	5.054

2.4.7 Zusammenfassung der Ergebnisse der Rechenzeituntersuchungen.

Aus den Ergebnissen der Rechenzeituntersuchungen ergeben sich für die optimale Gestaltung eines Codegenerators folgende Hinweise:

- Ab einer bestimmten Macrozahl (Bild 8) ist es günstiger, eine größere Anzahl Macros mit wenig Codezeilen als als eine kleinere mit vielen Codezeilen zu definieren.

- Bei häufig in verschiedenen Macros wiederkehrenden Gruppen von Codezeilen empfiehlt es sich, für diese ein besonderes Aufrufmacro zu definieren, was die Zahl der Codezeilen in den einzelnen Macros und somit auch die Baumerstellungszeit verringert.

- Macros, die bei Übersetzerläufen wahrscheinlich häufig aufgerufen werden, so z.B. Aufrufmacros für Macrokörper, müssen möglichst bald definiert werden.

- Müssen größere Gruppen von Zeichen einer Aufrufzeile in eine Macrokörperzeile übergeben werden, um dort z.B. einen neuen Macroaufruf zu bestimmen, so empfiehlt es sich hierzu möglichst wenige, dafür aber längere Parameter zu verwenden.

In den Bildern 10 und 11 ist der aus den Ergebnissen nach Bild 6 extrapolierte Verlauf der Rechenzeit für bis zu 500 Macros mit einer mittleren Codezeilenzahl von 3 Codezeilen und bis zu 600 Aufrufen dargestellt. Hieraus läßt sich die Rechenzeit für die Übersetzung eines Testprogramms mit 150 CIMIC-Befehlen mit dem in Abschnitt 3 beschriebenen Codegenerator bestimmen, der aus etwa 150 Macros besteht. Berücksichtigt man die Tatsache, daß die durch CIMIC-Befehle aufgerufenen Macros compilerintern im Mittel 3 weitere Macros aufrufen, so erhält man für nun 450 Macroaufrufe bei 150 definierten Macros aus Bild 10 die Rechenzeit 15.5 sec, was mit den tatsächlich erhaltenen Ergebnissen übereinstimmt. Bild 12 enthält den Verlauf der Rechenzeit für bis zu 2oo definierte Macros mit 0, 1, 3 und 6 Codezeilen und maximal 200 Aufrufen.

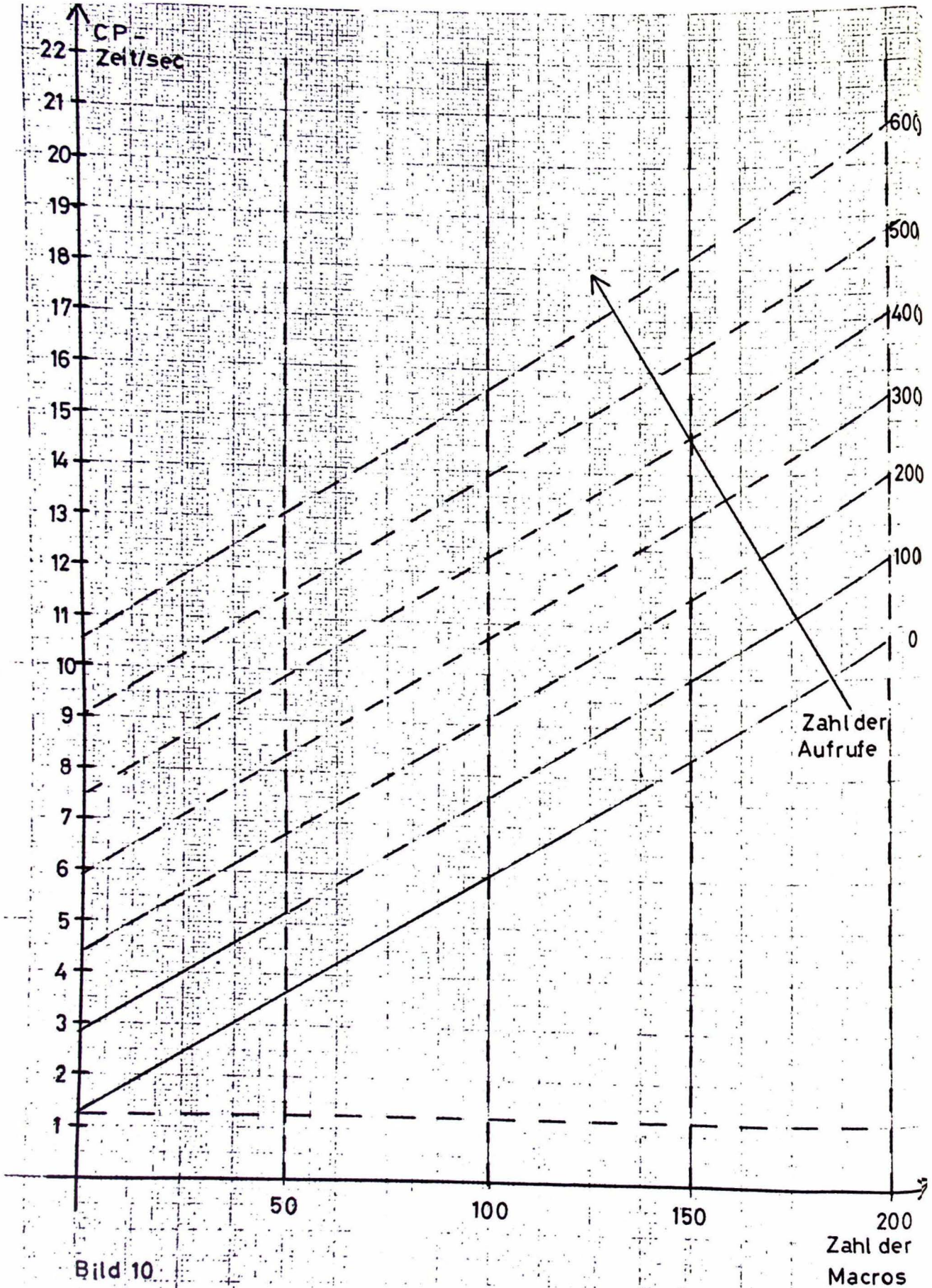


Bild 10

BAUMERSTELLUNGS-
+ ÜBERSETZUNGSZEIT
FÜR MACROS MIT 3 CODEZEILEN

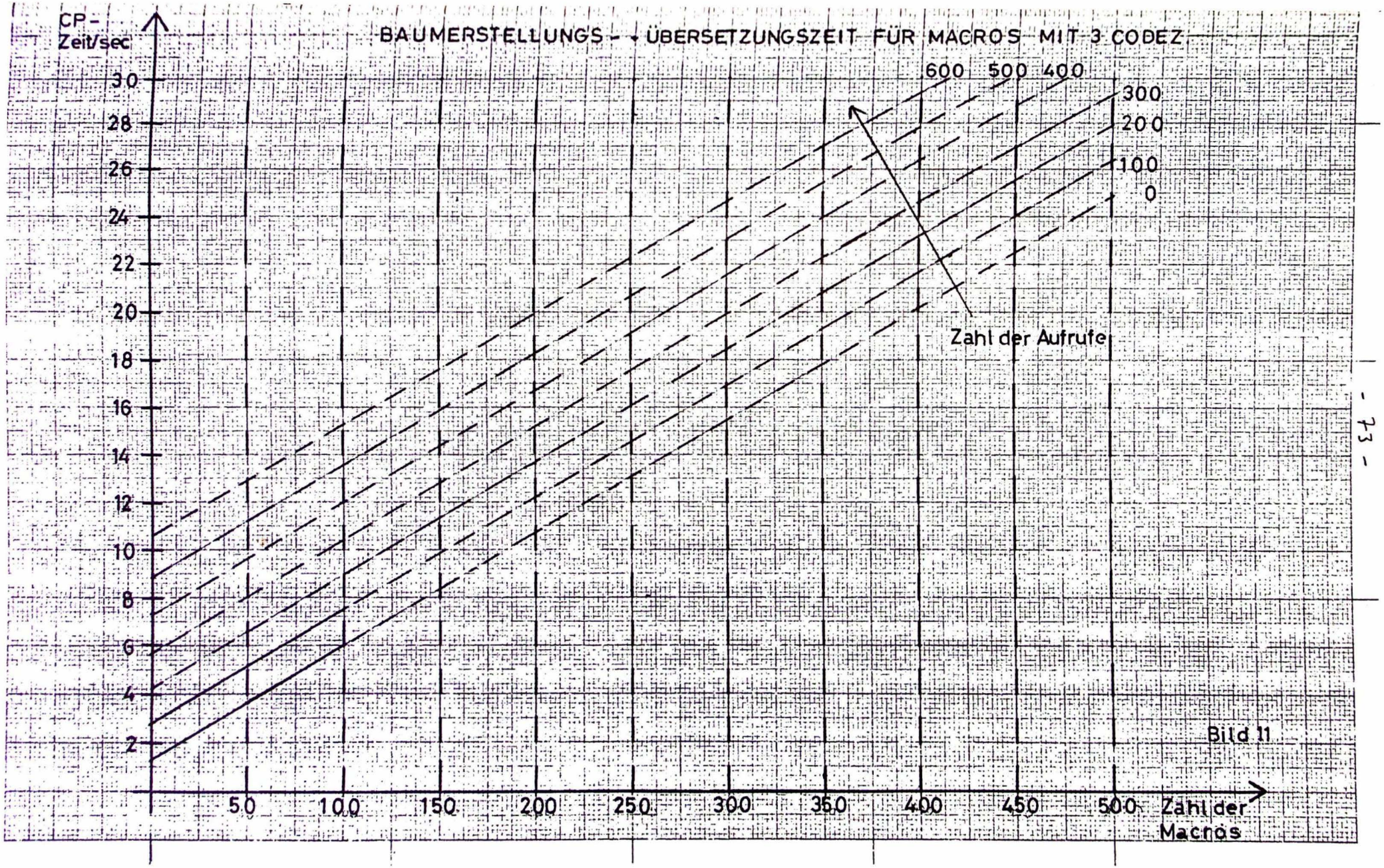

CP-Zeit/sec
BAUMERSTELLUNGS- + ÜBERSETZUNGSZEIT FÜR MACROS MIT 3 CODEZ
600 500 400
300
200
100
0
Zahl der Aufrufe
30
28
26
24
22
20
18
16
14
12
10
8
6
4
2
5.0
10.0
15.0
20.0
25.0
30.0
35.0
40.0
45.0
50.0
Zahl der Macros

Bild 11

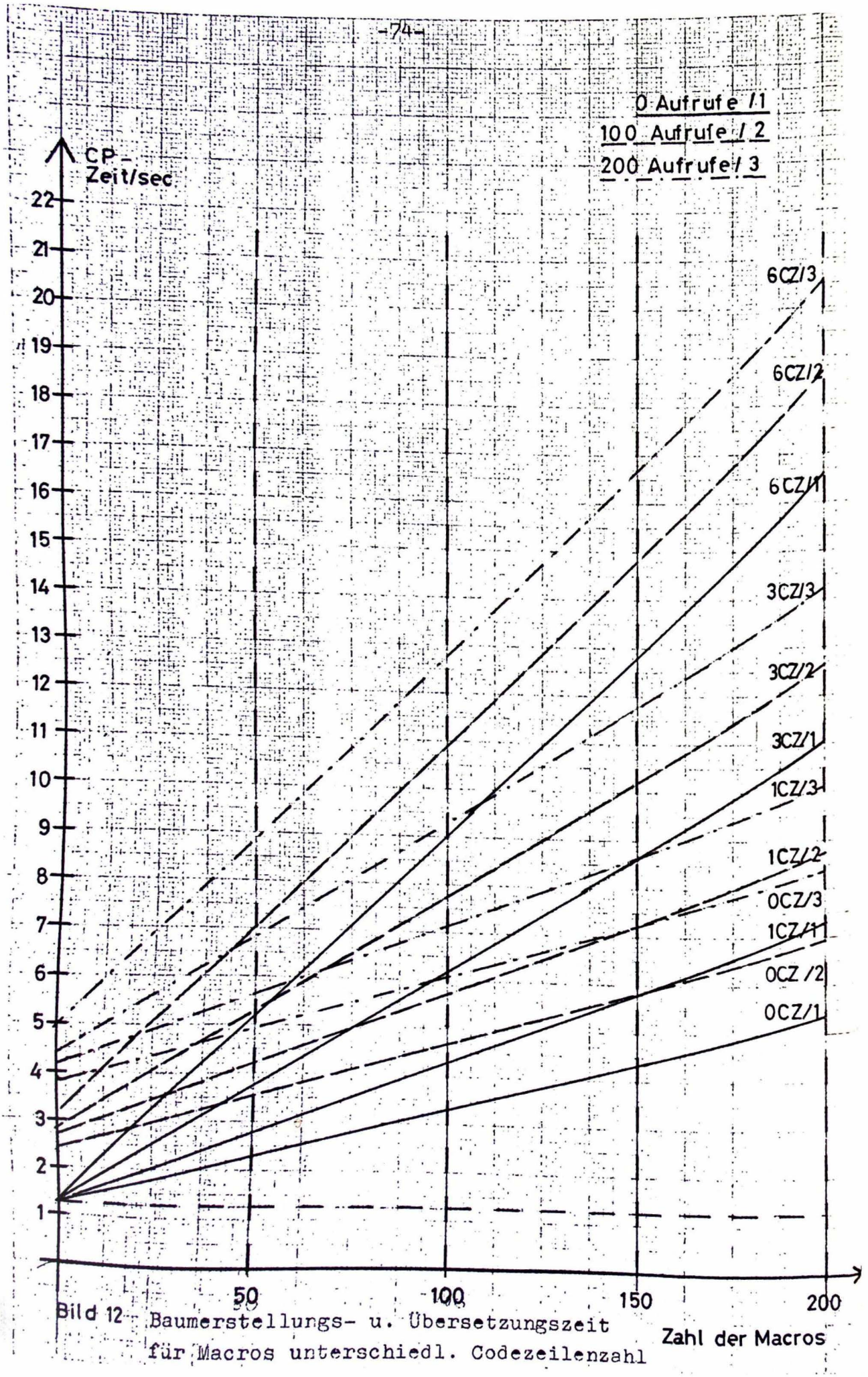


Bild 12 Baumerstellungs- u. Übersetzungszeit für Macros unterschiedl. Codezeilenzahl

2.5 Vergleich des Macroübersetzers STAGE 2 mit anderen Macroübersetzern

Die dem Macroübersetzer STAGE 2 vergleichbaren Macroübersetzer, die wie dieser einen allgemeingültigen Formalismus zum Umsetzen von Zeichenfolgen einer Quellsprache in Zeichenfolgen in einer Zielsprache beinhalten, unterscheiden sich vor allem in der Art der Macrodefinitionen und Macroaufrufe und können danach in drei Gruppen eingeteilt werden.

2.5.1 Streng formatgebundene Macrodefinitionen

In diesem Abschnitt wird im Wesentlichen auf Beschreibungen des GPM (general purpose macrogenerator) von Strachey /4/ und der Macrosprache 320/330 von Siemens /10/ Bezug genommen. Macroaufrufe bestehen hier aus einem Macronamen und einer Liste aktueller Parameter, die durch Kommata von einander getrennt sind (vgl. 2.1), während beim STAGE 2 auf diese scharfe Trennung in Namen und Parameterliste verzichtet wird. Der STAGE 2 erlaubt sogenannte verteilte Namen, d.h. eine ganze Eingabezeile wird als Macronamen angesehen, der an beliebigen Stellen von Rarametern durchsetzt sein kann.

Im Gegensatz zum STAGE 2 wird bei den hier betrachteten Übersetzern nicht jede beliebige Eingabezeile als möglicher Macroaufruf ausgewertet. Die Eingabezeichenfolge wird von links nach rechts untersucht und solange unbearbeitet ausgegeben, bis der Übersetzer ein sogenanntes Macroelement erkennt. Dies ist eine Zeichenfolge, die zwischen zwei bestimmten Steuerzeichen (z.B. § und ;) steht. Diese besondere Kennzeichnung der Macroelemente bringt den Vorteil, daß solche Macroaufrufe an beliebiger Stelle eines Programms in Verbindung mit oder an Stelle eines Symbolstrings stehen können, während der STAGE 2 immer eine gesamte Eingabezeile bearbeitet, die er entweder nicht als Macroaufruf erkennt und deshalb unverändert ausgibt, oder die er vom ersten Zeichen bis zu einer Zeilenendemarke als Macroaufruf auswertet. Text nach der Zeilenendemarke geht für die Ausgabe verloren.

Auch bei diesen Übersetzern wird der gesamte Aufruf (einschließlich Steuerzeichen) durch die zuvor definierte Ersatzzeichenfolge ausgetauscht, wobei die formalen Parameter der Macrodefinition durch die aktuellen Parameter des Aufrufs ersetzt werden. Als aktuelle Parameter können wie beim STAGE 2 auch erneute Macroaufrufe stehen, die z.B. durch Klammern als solche zusammengefaßt werden und gegebenenfalls mit den nötigen Steuerzeichen versehen sein müssen. Die Verarbeitung solcher Macroaufrufe ist beim STAGE 2 umständlicher, da hier der ein Macro aufrufende Parameter erst in eine besondere Zeile des Macrokörpers kopiert werden muß. Diese kann dann wieder mit der Liste der definierten Macros verglichen werden. Beim STAGE 2 übergibt man jedoch üblicherweise Macros nicht als Parameter in ein anderes Macro, sondern sieht diese schon bei der Definition als Macrokörperzeile vor, wenn Macroschachtelung beabsichtigt wird.

Eine Entsprechung zu der Parameterumformung und den Übersetzerfunktionen des STAGE 2 besitzen diese Übersetzer mit den in Maschinencode fest einprogrammierten Basismacros. Diese Basismacros haben jedoch eine fundamentalere Bedeutung, denn sie müssen viele Funktionen des Übersetzers steuern, die beim STAGE 2 automatisch ablaufen, und erlauben so in gewisser Weise eine höhere Flexibilität.

Auch Macrodefinitionen müssen über ein solches Basismacro ablaufen. Eine Macrodefinition hat die Form :

§DEF,Name,Rumpf;

Hierbei ist DEF der Name des aufgerufenen Basismacros, das den Namen und den Rumpf des zu definierenden Macros als aktuelle Parameter 1 und 2 enthält und einander zuordnet. Da auf diese Weise eine Macrodefinition eigentlich nur ein Macroaufruf ist, kann diese bei diesen Übersetzern an beliebiger Stelle eines Programms erfolgen und sogar in einen aktuellen Parameter eines Aufrufs eingeschlossen sein. Solche Definitionen sind dann nur vorübergehend, da die Liste der aktuellen Parameter nach Auswertung des Aufrufs verloren geht. Diese flexible Macrodefinition ist

beim STAGE 2 nicht möglich, da hier zuerst alle Macros mit den zugehörigen Körpern als Block definiert werden müssen und dann erst die Aufrufe folgen können. Jedoch ist beim STAGE 2 die Zuordnung von Macronamen und Rumpf übersichtlicher und einfacher und der Rumpf kann leicht zu einem umfangreichen Programm zur Codeerzeugung ausgebaut werden.

Die Übernahme von aktuellen Parametern aus dem Macroaufruf bei der Auswertung eines Macrorumpfes erfolgt beim STAGE 2 über eine entsprechende Parameterumformung wie beim GPM durch Bezugnahme auf die Stellung des aktuellen Parameters im Macroaufruf. Bei der Macrosprache 320/330 besteht außerdem noch die Möglichkeit, einen Parameter durch ein Basismacro über ein Kennwort anzusprechen.

Entscheidender Vorteil des STAGE 2 gegenüber den beiden Vertretern dieser Gruppe ist seine hohe Portabilität. Er erlaubt nicht nur formatfreiere und übersichtlichere Textverarbeitung (Macroaufruf), indem zu verarbeitender Text und Steuerzeichen, die beim STAGE 2 frei definiert werden können, nicht in so hohem Maße vermischt werden müssen, sondern ist z.B. auch auf jeden Rechner übertragbar, der einen FORTRAN-Compiler besitzt. Dagegen ist die Macrosprache 320/330 an bestimmte Rechnertypen der Fa. Siemens gebunden (Übersetzerprogramm SM 30 für den Prozeßrechner 330, Programm SUMU für die Prozeßrecher 304, 305 und 306) und der GPM, der in CPL oder einer bestimmten Maschinensprache (z.B. Titan, Atlas) geschrieben ist, müßte erst auf die zu verwendente Rechenanlage implementiert werden. Nach Angaben des Autors /4/ würde hierzu bei einem geeigneten Rechner von einem erfahrenen Programmierer eine Woche benötigt.

2.5.2 Freie Definition der Trennzeichen zwischen den Parametern

Bei den Macroübersetzern von Abschnitt 2.5.1 hat ein Macroaufruf die Form :

§ Name, par 1, par 2, ... , par n;

wobei die Kommata als Trennzeichen zwischen den aktuellen Paramete und das Semikolon als Schlußzeichen des Macroaufrufs fest vorgegeb

sind, während diese beim Macroübersetzer ML/I von P.J.Brown /7/ vom Benutzer bei der Definition eines Macros bestimmt werden können. Auch hier wird Eingabetext, wenn er als Macroaufruf erkannt wird (vgl. 2.5.1), in Ausgabetext umgewandelt. Jedoch wird der Eingabetext nicht wie beim STAGE 2 als Folge von Einzelzeichen untersucht, sondern dieser wird von Trennzeichen zu Trennzeichen zu Einheiten zusammengefaßt. Als solche Trennzeichen sind beliebige Zeichenfolgen erlaubt, die der Benutzer zusammen mit dem Namen definieren muß. Dabei muß für jedes einzelne Trennzeichen ein oder mehrere Folgetrennzeichen angegeben werden.

Beispiel für den Aufruf eines Macros mit dem Namen IF :

IF Argument 1 = Argument 2 THEN Argument 3 END

oder :

IF Argument 1 = Argument 2 THEN Argument 3 ELSE Argument 4 END

Hierbei ist IF der Name und =, THEN, END bzw. ELSE, END sind die anzugebenden Trennzeichen.

Bezüglich Macroschachtelung und dem Übersetzer fest einprogrammierter Basismacros gelten ähnliche Aussagen wie für die in Abschnitt 2.5.1 beschriebenen Übersetzer. Der Übersetzer ML/I wurde bisher auf den Rechenanlagen PDP-7 und Atlas 2 implementiert und benötigte dort 3000 Speicherworte.

2.5.3 Verteilter Name

Zu dieser Gruppe gehört außer dem Macroübersetzer STAGE 2 der ebenfalls von Waite entwickelte Übersetzer LIMP (Language-Independent Macro Processor) /6/. Die Anlage dieses Übersetzers ähnelt der des STAGE 2 weitgehend und ist wohl eine seiner Vorstufen. Er kennt nur wenige Arten der Parameterumformung und keine Übersetzerfunktionen. Statt dessen müssen die Zeilen der Macrokörper als Anweisungen einer SNOBOL ähnlichen Sprache geschrieben werden. Dieser Übersetzer kann keine Alternative zum STAGE 2 darstellen, da er gegenüber diesem keine Vorteile bietet, sondern nur höheren

Programmieraufwand erfordert. Seine wesentlichen Merkmale wurden bereits mit dem STAGE 2 beschrieben.

Das Gleiche gilt auch für den von R.J.Orgass und W.M.Waite entwickelten Übersetzer SIMCMP (Simple Compiler). Es handelt sich hierbei um einen einfachen, nach dem Prinzip des STAGE 2 arbeitenden Übersetzer, der dazu verwendet werden kann, leistungsfähigere Übersetzer wie z.B. den STAGE 2 im Bootstrapverfahren auf eine Rechenanlage zu bringen. Er wurde schon auf mehreren Rechnern implementiert (IBM 7044, IBM 360/50 und CDC 6400) und in /8/ ist der SIMCMP-Algorithmus in FORTRAN angegeben.

2.5.4 Zusammenfassung des Vergleichs

Für die Verwendung des Macroübersetzers STAGE 2 spricht außer seiner Portabilität (FORTRAN-Version) auch seine Benutzerfreundlichkeit. Der Anwender kann z.B. zur Übersetzung von CIMIC/1 Macros in dieser Sprache definieren und direkt aufrufen, und den zugehörigen Macrokörper leicht und übersichtlich zu einem leistungsfähigen Programm zur Codeerzeugung ausbauen, ohne daß er eine Sprache zur Textverarbeitung, wie sie zum Beispiel im Macro-Generator System von Krupp-Atlas /12/ oder mit TRAC /5/ vorgestellt wird, erlernen muß , mit der er sich einen Macroübersetzer konstruieren könnte.

Wie vorteilhaft es ist, wenn man Macros formatfrei definieren und aufrufen, man also direkt mit der Quellsprache (z.B. CIMIC/1) arbeiten kann, soll am folgenden Beispiel eines Additionsmacros gezeigt werden, das die Anweisung A=B+C in die Befehle FETCH B, ADD C und STORE A umsetzen soll.

GPM

```
§DEF,ADDITION,
     [FETCH ∿2;
      ADD   ∿3;
      STORE ∿1;];
§ADDITION,A,B,C;
```

ML/I

```
MCDEF ADDITION=+;AS
     <FETCH :A2;
      ADD   :A3;
      STORE :A1;
      >;

ADDITION A=B+C;
```

STAGE 2

```
!=!+!§
FETCH!2§
ADD  !3§
STORE!1§
§

A=B+C §
```

Wie man sieht, kann mit dieser Anweisung nur mit dem Übersetzer STAGE 2 direkt ein Makro aufgerufen werden. Für ML/I muß diesem Befehl erst ein Name vorausgestellt werden, und beim GPM ist eine direkte Verarbeitung dieses Befehls überhaupt nicht möglich.

Ein Nachteil des STAGE 2 ist sein hoher Speicherplatz-und Rechenzeitbedarf. Jedoch wie hoch dieser gegenüber anderen Übersetzern ist, kann nicht beurteilt werden, da diese an bestimmte Rechner gebunden und nicht auf der CDC 660 lauffähig sind.

3. Codegenerator für den betriebssystemunabhängigen Teil von CIMIC/1

Nach der Beschreibung von CIMIC/1 in /13/ liegt den CIMIC-Anweisungen ein abstrakter Rechner zugrunde, dessen programmierbare Teile aus

- Akkumulator
- Indexregister
- Speicher

bestehen.

Der Akkumulator dient zur Aufnahme von Operanden aller in CIMIC zulässigen Datentypen und enthält das Ergebnis der mit disen 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 Offsets addiert werden können.

CIMIC-Anweisungen haben das Grundformat

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

Die Wirkung einer CIMIC-Anweisung wird durch den Operationscode bestimmt

operation:: = { opcode/opcode,I }

Wird dem Operationscode, "I" hinzugefügt, so wird die Operation nicht mit dem durch die Referenzen 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 führen.

Das Referenz-Feld enthält Hinweise auf die Operanden in Form von symbolischen Namen. Es treten verschiedene Kategorien von Referenzen auf. Für die im Folgenden ausgewählten CIMIC-Befehle sind jedoch nur die Referenzen 1 und 2 von Bedeutung.

Referenzen zu Programmkonstanten:

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

Mit "type" wird die Art der Konstante angegeben

$$\text{type} :: = \{ \text{A/E} \}$$

Der Wert einer Konstante vom Typ A ist unabhängig von der Zielmaschine, während eine Konstante vom Typ E zielrechnerabhängig und mit der Adreßstruktur des Zielrechner verknüpft ist. Die Konstante vom Typ E wird dargestellt in Form eines Produkts 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.

Referenzen zu VAL-Größen bestehen aus einem Schlüssenwert z.B. VLLOC für lokale VAL-Größen, aus dem symbolischen Namen, einem festen Offset und ggfs. dem Zeichen "+".

$$\text{reference 1 b} :: = \left\{ \begin{matrix} \text{VLLOC} \\ \text{VLPAR} \end{matrix} \right\} \sqcup \text{symbol} \; (\; [\text{fix offset}]) [+] \sqcup\sqcup$$

Das Symbol ergibt eine Basisadresse. Ein verhandener 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 zu Basisadresse addiert. Referenzen zu Variablen und Feldelementen enthalten ein Schlüsselwort z.B. LOCAL für lokale Variable oder Feldelemente, sowie ein Symbol, einen festen Offset und ggfs. das Pluszeichen:

$$\text{reference 2} :: = \left\{ \begin{matrix} \text{LOCAL} \\ \text{PARAM} \\ \text{ARG} \end{matrix} \right\} \sqcup \text{symbol} \; ([\text{fix offset}]) [+] \sqcup\sqcup$$

3.1 Voraussetzungen für die Übersetzung von CIMIC/1 in den Assemblercode des DIETZ 621.

Der DIETZ 621 ist ein Kleinrechner mit 4,8,16 oder 32 Kbyte Kernspeicher und einem Halbleiter-Speicher (Pool) mit extrem kurzer Zugriffszeit, in dem sich Register, Datenpuffer und schnellaufende Programme befinden. Er arbeitet mit variabler Wortlänge, z. B. bei der Prozeßrechnersprache BASEX 8 bit für Textverarbeitung, 16 bit für Prozeßdaten, 32 bit für Rechengrößen. Der Rechner hat bis zu 16 unabhängige Programmebenen mit eigenen Registern und Datenkanälen. Jede Programmebene verfügt über bis zu 256 Register. Als Arbeitsregister dient der Akkumulator (, Pool-Adresse '02) und eventuell die folgenden Register. Werden Befehle indiziert, so dient die in einem besonderen Byte angegebene Pool-Adresse und die folgende Adresse als Indexregister (2-Byte-Indexregister). Ist die angegebene Pool-Adresse ungerade, wird nur diese Adresse als Indexregister verwendet (1-Byte-Indexregister).

Für die Übersetzung von CIMIC/1 in den Assemblercode des DIETZ 621 wurden die folgenden Voraussetzungen festgelegt:

-1) Darstellung von INT, DUR, CIO und BIT-Größen durch 2 Byte (D-Typ).
Darstellung von REAL-Größen durch 4 Byte (G-Typ)
Darstellung von CHAR-Größen mit 8 Bit je Zeichen

-2) Auf dem DIETZ 621 sollen nicht mehrere Module gleichzeitig laufen können, d. h. GLOBAL-Größen sind nicht zulässig.

-3) Konstanten wird der in CIMIC mitgegebene Name zugewiesen. Operationen mit REAL-Konstanten können deshalb wie Variablenoperationen behandelt werden. Die übrigen Konstantenoperationen sind als solche auf dem DIETZ 621 direkt ausführbar.

-4) CLOCK- und DURATION-Größen werden intern als Festkommazahlen in Sekunden dargestellt.
-5) Als Indexregister für den abstrakten Rechner werden die Register mit den Pool-Adressen '40 und '41 verwendet (2-Byte-Indexregister), d.h. es ist positive und negative Indizierung möglich. Alle Integerzahlen (2-Byte-Darstellung) und alle Adressen (2 Byte) sind erfaßt. Zur Übergabe von Variablen für Bibliotheksprogramme werden 64 Bytes des jeweiligen Pools benötigt (Adresse '00 bis '3F).
-6) Marken der Form Zdd, wobei d für eine Ziffer steht, werden für die Übersetzung benötigt und sind reserviert.

Im folgenden verwendete Hilfsmacros sind in Abschnitt 2.3 beschrieben.

3.2 Implementierung der Referenzen 1 und 2 in CIMIC/1 für den DIETZ 621.

Bei der Implementierung der Referenzen zu Konstanten, Referenz 1, muß zwischen REAL-Größen (G-Typ) und Größen in 2-Byte-Darstellung (D-Typ) unterschieden werden. D-Typ-Größen können direkt als Konstante behandelt werden, während REAL-Größen wie Variable, Referenz 2, behandelt werden müssen.

Zur Übersetzung der Referenzen 1 und 2 werden 6 Macros und einige Hilfsmacros benötigt, mit denen der Wert von Konstanten vom Typ E schon bei der Übersetzung berechnet und in eine direkt ladbare Form gebracht wird. Zur Erklärung der Funktion der **Refe**renz-Macros müssen erst noch Macros zur Unterscheidung des MODE und Macros für spezielle Befehle definiert werden. In den Referenz-Macros übergibt Parameter 1 die Befehlsart, Parameter 2 das Arbeitsregister und Parameter 3 den MODE (entweder D für 2-Byte-Größen ode G für 4-Byte-Größen) in den zu erzeugenden Assemblerbefehl.

```
! ! REF 12 ! !() A !  $
IF !30 = D SKIP 2$
       !10A!30,!20,!40;!F14$
SKIP 3$
IF !50 NE HILF SKIP 1$
!51!56$
       !10C!30,!20,!50;!F14$
$
```

Soll die Operation mit einer REAL-Konstante ausgeführt werden, so wird der Operand über das ihm zugeordnete Symbol aufgerufen. 2-Byte-Größen, durch D gekennzeichnet, werden als Konstante behandelt.

```
! ! REF 12 ! !() E !  $
!50MEM$
HILF!56$
       !10C!30,!20,!51;!F14$
$
!E!MEM$
IF !21 NE  SKIP 1$
BIT!26$
!10*!21!10$
HILF ECU !14$
$
!ECHAR(!)MEM$
!10*!20!10$
HILF ECU !10$
$
!MEM$
HILF ECU !11$
$
CHAR(!)MEM$
HILF ECU,!10$
$
BI,(!)MEM$
HILF ECU 2$
$
```

Für Konstanten vom Typ E muß erst der Wert des Operanden berechnet werden. Dies geschieht mit Hilfe des STAGE 2 über eines der nebenstehenden Hilfsmacros. Sie rufen den Wert des im STAGE 2 - Speicher abgelegten MODE-Identifiers auf und legen den Wert der dann berechneten Konstante im STAGE 2 - Speicher unter HILF ab.

```
! ! REF 12 ! !()    $
       !10A!30,!20,!40;!F14$
$
```

Referenz 1b und 2. Der Operand wird als Variable behandelt und über das Symbol durch seine absolute Adresse angesprochen.

```
! ! REF 12 ! !()+    $
       !10A!30,!20,!40,'40;!F14$
$
```

Der durch dieses Referenzmacro aufgerufene Befehl wird mit dem Indexregister '40 indiziert, d.h. der Inhalt des Indexregisters wird zu der durch das Symbol angegebenen Basisadresse addiert.

```
! ! REF 12 ! !(!)    $
!50MEM$
LE '42 REF 12 D   () A HILF  $
       !10A!30,!20,!40,'42;!F14$
$
```

Der in dieser Referenz enthaltene Fix Offset wird zuerst vom STAGE 2 berechnet und im STAGE 2 - Speicher unter HILF abgelegt. Der Inhalt dieses Speiche wird dann mit einem Assemblerbefehl in das Hilfsindexregister '42

geladen und der erzeugende Befehl wird dann mit diesem Indexregister indiziert.

```
! ! REF 12 ! !(!)+     $
!50MEM$
LD '42 REF 12 D    () A HILF  $
       2=*ADR,'42,'40:!F14$
       !10A!30,!20,!40,'42:!F14$
$
```

Auch hier wird der Fix Offset berechnet und in das Hilfsindexregister '42 geladen, zu dem dann der Inhalt des Indexregisters '40 addiert wird. Der zu erzeugende Assemblerbefehl wird dann mit Register '42 indiziert.

3.3 Erkennung der MODE des Operanden.

Da Operanden mit unterschiedlichem MODE in unterschiedlicher Wortlänge dargestellt werden und zum Teil eine Vorbehandlung benötigen, die durch den STAGE 2 durchgeführt werden kann, sind folgende Macros nötig:

```
! MODE ! !  $
!10 @ REF 12 D !30  $
$
```

Dieses Macro erkennt einen Befehl mit einem Operanden vom MODE INT und übergibt in das den entsprechenden Befehl erzeugende Referenzmacro den Kennbuchstaben D (2-Byte-Darstellung). Das Gleiche bewirkt dieses Macro fur Variable vom MODE CLO oder DUR als Operanden.

```
! MODE REAL !  $
!10 @ REF 12 G !20  $
$
```

Operanden vom MODE REAL muß der Kennbuchstabe G zugeordnet werden (4-Byte-Darstellung).

```
! MODE CLO !() A !:!:!  $
EVALDUR !30*3600+!40*60+!50$
!10 @ REF 12 D !20() A STACKDUR  $
$

EVALDUR !$
STACKDUR EQU !14$
$
```

CLOCK-Größen werden als Festkommazahlen in Sekunden dargestellt (2 Byte - D). Der Wert der Uhrzeit in Sekunden für Konstante wird durch den STAGE 2 über das Macro EVALDUR berechnet und unter STACKDUR im STAGE 2 - Speicher abgelegt.

```
! MODE DUR !() A !  $
DURATION !50$
!10 @ PEF 12 D !20() A STACKDUR  $
$
DURATION !HRS$
EVALDUR !10*3600$
$
DURATION !HRS !MINS$
EVALDUR !10*3600+!20*60$
$
DURATION !HRS !SEC$
EVALDUR !10*3600+!20$
$
DURATION !HRS !MIN !SEC$
EVALDUR !10*3600+!20*60+!30$
$
DURATION !MIN$
EVALDUR !10*60$
$
DURATION !MIN !SEC$
EVALDUR !10*60+!20$
$
DURATION !SEC$
EVALDUR !10$
$
```

Auch DURATION-Konstanten müssen erst vom STAGE 2 in Sekunden umgewandelt werden. Dies geschieht über ein entsprechendes DURATION-Macro.

```
! MODE BIT(!) !  $
BITL EQU !20$
!10 @ PEF 12 D !30  $
$
```

Variable vom MODE BIT werden in 2 Byte dargestellt. Für evertuell nachfolgende BIT-Operationen wird die Länge der Bitkette im STAGE 2 - Speicher unter BITL abgelegt und ist somit abfragbar.

```
! MODE BIT(!) !() A *!*B1  $
BITL EQU !20$
SPILLSWITZEN 16$
BREAK BITLIST !40 17-!20$
BREAK HEXA B1 B2 B3 B4*B5 B6 B7 B8*B9 B10 B11 B12*B13 B14 B15 B16$
STORE HEXA B1 B2 B3 B4$
'10 @ PEF 12 D !30() A BIL2  $
$
SPILLSWITZEN !$
! EQU 0$
'10!171
B1+1 EQA 0$
!FF$
$
B! EQA !$
B!14 EQU !20$
! EQU !14$
$
BREAK BITLIST ! !$
1 EQU !24$
'10!17$
B1+1 EQA !10$
!FF$
$
```

Da der DIETZ 621 nur Dezimal- und Hexadezimalzahlen verarbeiten kann, muß der Operand des CIMIC-Befehls erst durch den STAGE 2 in eine entsprechende Hexadezimalzahl umgewandelt werden. Hierzu wird der Operand in die einzelnen Bits aufgespalten, die vom STAGE 2 in der Speicherplätzen B1 bis B16 abgelegt und dann zu 4-er-Bitmustern zusammengefaßt werden. Die Zuordnung die Bitmuster zu Hexazahlen wird über

```
BREAK HEXA !$
I EQU 0$
!10!17*$
HEXAMEM1 !10$
!F8$
$
HEXAMEM1 ! ! ! !$
HEXAMEM2 !11!21!31!41$
$
HEXAMEM2 !$
BI+1 EQA !11$
$
STORE HEXA ! ! ! !$
HILF2 EQU '!11!21!31!41$
$
```

den STAGE 2 - Speicher vorgenommen. Hierfür wird zu Anfang einer jeden Übersetzung zu jedem möglichen Bitmuster die entsprechende Hexadezimalzahl abgespeichert.

3.4 Transferbefehle.

3.4.1 Laden in den Akkumulator.

Für 2-Byte-Größen werden die Register '02 und '03 und bei 4-Byte-Größen die Register '02 bis '05 als Arbeitsregister belegt. Somit ist z.B. der Befehl "Akkumulator laden" vom MODE abhängig.

```
LOAD ! ! !$
LD MODE !10 !30  $
$
```

LOAD { INT / REAL / CLO / DUR / BIT(length) } {reference 1/2}ø

Beispiel:

Eingabezeile: LOAD INT LOCAL IOO1(3*INT)ø

Hilfsmacros:

```
! MODE ! !  $
!10 @ BYF 12 D !30  $
$
```

```
! ! BYF 12 ! !(!)    $
!SOMEMS
LD '42 BYF 12 D    (! A HILF  $
        !10A!30,!20,!30,'42;!F14$
$
```

```
!*(MEM
IF !21 NE  SKIP 1$
BIT!20$
!10*!21!10$
HILF EQU !14?
$
```

Ausgabezeile für den DIETZ 621:

LDA,@,IOO1,'42;

Beim Laden einer CHAR-Größe werden lediglich die Länge und der Name der Zeichenfolge im STAGE 2 - Speicher bei Adresse CHARL1 und CHAR1 eingetragen. Diese sind dann bei nachfolgenden Verkettungs- oder STORE-Befehlen abfragbar.

```
LOAD CHAR(!) ! !()  $
NULL EQU 0$
CHARL1 EQU !20$
CHAR1 EQU !30$
$
```

LOAD CHAR(length) {reference 1/2}ø

3.4.2 Abspeichern aus dem Akkumulator.

Dieser Befehl wird, wenn keine Pypkonversion nötig ist, wie folgt übersetzt:

```
STORE ! ! !$
ST MODE !10 !30  $
$
```

STORE { INT / REAL / BIT(length) / CIO / DUR } {reference 1/2} Ø

Der Inhalt des Akkumulators wird nicht verändert, deshalb kann der folgende Befehl dieses Macro ebenfalls verwenden.

```
STN !$
STORE !10$
$
```

STN { INT / REAL / BIT(length) / CLO / DUR } {reference 1/2} Ø

Bei Typkonversion müssen folgende Fälle unterschieden werden:

1) Die im Akkumulator stehende Gleitkommazahl wird umgewandelt in eine Festkommazahl und dann abgespeichert.
Bei der Umwandlung muß zuerst geprüft werden, ob die im Akkumulator stehende Gleitkommazahl die für 2-Byte-Zahlen erlaubte Größe (32 767) übersteigt. Wenn ja, wird eine Fehlermeldung (ER1) ausgegeben, und das Programm angehalten, sonst wird durch eine Bibliotheksfunktion der ganzzahlige Wert der Gleitkommazahl im Akkumulator (Reg. '02 und '03) eingetragen und von dort aus abgespeichert.

```
STORE !,! ! C REAL  $
UMWANDLUNG REAL INT$
ST @ REF 12 D !30  $
$
UMWANDLUNG REAL INT$
        STRG,@,'42;!F14$
        AR;!F14$
        SBCG,@,'7FFF;!F14$
        ANP,'05,    ,Z!00;!F14$
        2=*VTC,'00,'0A0D;!F14$
        2E=*VTA,'00,ER1;!F14$
         JPA,    ,ENDE;!F14$
Z!00:   LDBG,@,'42;!F14$
        IT;!F14$ .
$
```

STORE { INT / DUR / CLO } symbol(Fix offset) [+] C REAL Ø

2) Die im Akkumulator stehende Festkommazahl wird in eine Gleitkommazahl umgewandelt und dann abgespeichert.
Da es hierzu für den DIETZ 621 keine Bibliotheksfunktion gibt, muß die Umwandlung schrittweise vorgenommen werden.

```
STORE REAL ! ! C !  $
UMWANDLUNG INT REAL$
ST @ REF 12 C !20  $
$
UMWANDLUNG INT REAL$
        LDB,'04,'03;!F14$
        LDC,'05,0;!F14$
        ANC,'04,'80;!F14$
        ANC,'03,'7F;!F14$
$
```

Darstellung von
2-Byte-Zahlen

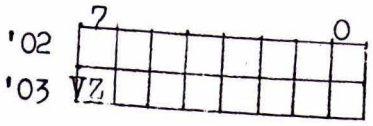
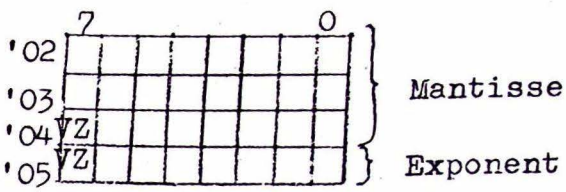


Bild 13

Darstellung von
4-Byte-Gleitkommazahlen



Hierzu muß das Vorzeichen aus Reg. '03 nach Reg. '04 gebracht werden. Das bisherige Vorzeichenbit in Reg. '03 und die Bits 0 bis 6 in Reg. '04 sowie Reg. '05 müssen 0 gesetzt werden

STORE REAL symbol(fix offset)[+] C {INT / CLO / DUR}ø

3) Fälle, in denen keine Umwandlung notwendig ist, da intern INT, DUR und CLO als Festkommazahlen dargestellt sind.

```
STORE ! ! ! C ! $
ST @ REF 12 D !30  $
$
```

STORE {INT / CLO / DUR} symbol(fix offset)[+] C {INT / CLO / DUR}ø

Die Abspeicherung von CHAR-Größen wird bei der Beschreibung der Verkettungsbefehle mitbehandelt.

3.4.3 Laden ins Indexregister.

```
LDX INT ! !$
LD '40 REF 12 D !20  $
$
```

Der zugehörige Befehl kann über ein entsprechendes Referenzmacro mit Register '40 als Arbeitsregister ohne Benutzung des Akkumulators erzeugt werden.

LDX INT {reference 1/2}ø

3.4.4 Abspeichern des Indexregisters.

```
STX INT ! !$
ST '40 REF 12 D !20  $
$
```

Auch hier muß der Akkumulator nicht benützt werden. Es kann direkt aus dem Indexregister abgespeichert werden.

STX INT {reference 1/2}ø

3.5 Arithmetische Befehle.

REAL-Zahlen (Gleitkommazahlen) und Ganzzahlen werden nach Bild 13 dargestellt (Abschnitt 3.4.2).

3.5.1 Bildung des Absolutbetrags.

Der Absolutbetrag von REAL-Zahlen kann über eine Bibliotheksfunktion gebildet werden.

```
ABS REAL    $
        AR;!F14$
$
```

ABS REAL Ø

Bei negativen Integerzahlen muß über das Macro "NEG INT Ø" das Vorzeichen umgekehrt werden.

```
ABS INT     $
        BP,'03,     ,Z!00;!F14$
NEG INT     $
Z!00:   NOP;!F14$
$
```

ABS INT Ø

3.5.2 Vorzeichenumkehr.

Bei REAL-Größen wird Bit 7 von Register '04 (Bild 13), das das Vorzeichen der Mantisse angibt, invertiert, d.h. zu Bit 7 muß 1 addiert werden.

```
NEG REAL    $
        ADC,'04,'80;!F14$
$
```

NEG REAL Ø

Für die Vorzeichenumkehr der Integerzahl z gilt:

$-z = \overline{(z - 1)}$, d.h. bei der Vorzeichenumkehr muß das Einerkomplement der Zahl (z - 1) gebildet werden, was einer exklusiven ODER-Verknüpfung der jeweiligen Register mit 1111 1111 gleichbedeutend ist.

```
NEG INT     $
VORZEICHENUMKEHR @$
$

VORZEICHENUMKEHR !$
        2=#SEC,!10,'0001;!F14$
        2<#OC,!10,'FF;!F14$
$
```

NEG INT Ø

3.5.3 Addition, Subtraktion, Multiplikation, Division und Exponentiation.

INT, CLO und DUR werden als 2-Byte-Ganzzahlen und REAL-Zahlen als 4-Byte-Gleitkommazahlen dargestellt. Die Zuordnung zur richtigen Darstellung wird über das "MODE-Macro" vorgenommen.

Für die Addition gilt also:

```
ADD ! ! !$
AD MODE !10 !30  $
$
```

ADD {INT / DUR / CLO / REAL} {reference 1/2} ø

und für die Subtraktion:

```
SUB ! ! !$
SB MODE !10 !30  $
$
```

SUB {INT / DUR / CLO / REAL} {reference 1/2} ø

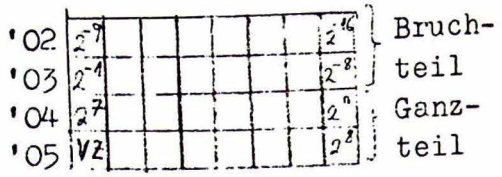
Für die Multiplikation können Bibliotheksbefehle zur Multiplikation von Gleitkommazahlen bzw. Ganzzahlen verwendet werden, jedoch muß man berücksichtigen, daß bei der 2-Byte-Multiplikation ein 4-Byte-Ganzzahl-Produkt entsteht, das nicht größer als die größtmögliche 2-Byte-Ganzzahl sein darf, sonst muß eine Fehlermeldung (ER1) ausgegeben und das Programm angehalten werden. Zu dieser Überprüfung muß der Akkumulatorinhalt vorübergehend in den Registern '46 ff. abgelegt werden. Bei erlaubter Größe des Produkts muß das Vorzeichenbit (Bit 7 in Reg. '05) nach Bit 7 in Reg. '03 gebracht werden.

```
MPY ! ! !$
MP MODE !10 !30  $
IF !11 NE 2 SKIP 7$
      4=*STR,@,'46;!F14$
      4=*SBC,@,PRK;!F14$
      BNP,'05,     ,Z!00;!F14$
      2=*WTC,'00,'0A0D;!F14$
      2@=*WTA,'00,ER1;!F14$
      JPA,     ,ENDE;!F14$
Z!00: 4=*LDR,@,'46;!F14$
      ORR,'03,'05;!F14$
$
```

MPY {INT / DUR / CLO / REAL} {reference 1/2} ø

Bei der Division von 2-Byte-Ganzzahlen entsteht ein 4-Byte-Quotien mit Mittelkomma:

Darstellung des 4-Byte Quotienten:



Der Bruchteil wird abgeschnitten, d.h. die Register '05 und '04 werden nach Reg. '03 bzw. '02 geladen

```
DIV ! ! !$
DV MODF !10 !30  $
IF !11 NE 2 SKIP 4$
       BE,'03,    ,7!00;!F14$
       LDRD,@,'04;!F14$
$
```

DIV {INT / DUR / CLO / REAL} {reference 1/2}Ø

Zur Exponentiation kann ein vorhandener Bibliotheksbefehl für 4-Byte-Gleitkommazahlen verwendet werden.

```
EXP REAL ! !$
PO MODF REAL !20  $
$
```

EXP REAL {reference 1/2}Ø

2-Byte-Ganzzahlen müssen zuerst in 4-Byte-Gleitkommazahlen umgewandelt und nach ausgeführter Operation wieder in 2-Byte Ganzzahlen zurückverwandelt werden.

```
EXP ! ! !$
UMWANDLUNG INT REAL$
       4=*STR,@,'42;!F14$
EL MODF !10 !30  $
UMWANDLUNG INT REAL$
ST MODE !10 !30  $
EO MODF REAL !30  $
UMWANDLUNG REAL INT$
$
```

EXP {INT / CLO / DUR} {reference 1/2}Ø

Addition und Subtraktion mit dem Indexregister können direkt im Indexregister ausgeführt werden.

```
ADX INT ! !$
AD '40 REF 12 D !20  $
$
```

ADX INT {reference 1/2}Ø

```
SBX INT ! !$
SB '40 REF 12 D !20  $
$
```

SBX INT {reference 1/2}Ø

Gleiches gilt für die Addition des Akkumulatorinhalts zum Indexregister. Der Akkumulatorinhalt bleibt hierbei erhalten.

```
ADDAX INT    $
       2=*ADR,'40,'02;!F14$
$
```

ADDAX INT Ø

Zur Multiplikation ist ein Bibliotheksbefehl nötig, der nur im Akkumulator ausgeführt werden kan. Hierzu muß der Akkumulatorinhalt gerettet werden. Die Operation kann dann mit Macro "MPY ! ! !Ø" behandelt werden.

```
MPX INT ! !$
        4=*STR,@,'42;!F14$
        2=*LDR,@,'40;!F14$
MPY INT !10 !20$
        2=*STR,@,'40;!F14$
        4=*LDR,@,'42;!F14$
$
```

MPX INT {reference 1/2}Ø

3.6 Bitoperationen.

Für Bitketten werden 16 Bit belegt, d.h. Reg. '02 und '03 werden benützt.

3.6.1 Komplementbildung.

Bei der Komplementbildung werden die Register '02 und '03 durch exklusives ODER mit 1111 1111 verknüpft. Hierbei wird nicht berücksichtigt, daß Bitketten auch weniger als 16 Bit lang sein können.

```
NOT BIT(!)    $
        2<&EOC,@,'FF;!F14$
$
```

NOT BIT(length) Ø

3.6.2 Logische Verknüpfungen.

Bei der AND-, ODER- und Exklusiv-ODER-Verknüpfung wird bei einer Konstanten als Operand über das entsprechende MODE-Macro zuerst vom STAGE 2 eine Hexazahl gebildet. Sonst wird die Adresse des Operanden angesprochen. Die Verknüpfungen sind 16-Bit-Verknüpfungen Bitvariablen werden 16 Bits zugewiesen. Nicht belegte Bits haben dabei Nullinhalt.

```
AND ! ! !  $
AN MODE !10 !30  $
$
```

AND BIT(length) {referencel/2}Ø

```
OR ! ! !  $
OR MODE !10 !30  $
$
```

OR BIT (length) {referencel/2}Ø

```
EXOR ! ! !  $
EO MODE !10 !30  $
$
```

EXOR BIT(length){referencel/2}Ø

3.6.3 Shift-Befehl.

Ist der Operand kleiner 0, so wird nach links geshiftet, sonst nach rechts. Die Zahl der Bits um die geschoben wird, wird durch den Betrag des Operanden bestimmt. Hierzu wird der Operand (2-Byte-Ganzzahl) in die Hilfsregister '42 und '43 geladen. Das Vorzeichen des Operanden steht dann in Bit 7 von Register '43. Hat Register '43 positiven Inhalt (Sprung nach Marke Z!00), so wird die in den Registern '02 und '03 stehende 16-Bitkette offen nach links geschoben, sonst muß der Betrag des Operanden gebildet werden, der die Zahl der Bits angibt um die dann die Register '02 und '03 offen nach rechts geshiftet werden.

```
SHIFT INT ! ! $
LI '42 HEX 12 D !30 $
      PP,'43,    ,Z!00;!F14$
VORZEICHENUMKEHR '42$
Z!01: 2<*SRO,@;!F14$
      2=*SRC,'42,1;!F14$
      2<*BNZ,'42,    ,Z!01;!F14$
      JPA,    ,Z!03;!F14$
Z!00: 2<*SLO,@;!F14$
      2=*SRC,'42,1;!F14$
      2<*BNZ,'42,    ,Z!00;!F14$
Z!03: NOP;!F14$
$
```

SHIFT INT {reference 1/2}$

3.6.4 Ausblendbefehle.

Der Operand a ist eine Integer-Größe mit $1 \leq a \leq n$, wobei n die Länge der Bitkette im Akkumulator ist. Das Ergebnis ist das a-te Bit des Akkumulatorinhalts von rechts bzw. links.

```
RBIT INT ! ! $
LI '42 HEX 12 D !30 $
Z!00: PPC,'42,1,Z!01;!F14$
      2=*SRC,'42,1;!F14$
      2<*SRO,@;!F14$
      JPA,    ,Z!00;!F14$
Z!01: NOP;!F14$
      [illegible]
$
```

Das gewünschte Bit wird durch Rechtsschieben nach Bit 0 von Reg. '02 gebracht. Die restlichen Bits des Akkumulators werden Null gesetzt.

RBIT INT {reference 1/2}$

```
LBIT INT ! ! $
LI '42 HEX 12 D !30 $
Z!00: 2<*BNZ,'42,    ,Z!01;!F14$
      2=*SRC,'42,1;!F14$
      2<*SLC,@;!F14$
      JPA,    ,Z!00;!F14$
Z!01: NOP;!F14$
      [illegible]
$
```

Das gewünschte Bit wird hier durch zyklisches Linksschieben nach Bit 0 vor Register '02 gebracht.

LBIT INT {reference 1/2}$

3.6.5 Verkettungsbefehle.

Beim Verketten von Bitfolgen können nur Bitketten von maximal 16 Bit Länge erzeugt werden. Bei längeren Bitketten erfogt eine Fehlermeldung (ER2) und das Programm wird angehalten. Zur Verketung wird der Operand in die Hilfsregister '42 und '43 geladen, und der Akkumulator um die Länge des Operanden nach links geschoben. Danach wird der Inhalt der Register '42 und '43 mit den Registern '02 und '03 ODER-verknüpft. Die erzeugte Bitkette steht dann rechtsbündig in den Registern '02 und '03.

```
CONC BIT(!) ! !  $
IF !10+BITL GT 16 SKIP 5$
LD '42 REF 12 D !30  $
!10!F7$
       2<*SLO,@;!F14$
!F8$
       2=*ORR,@,'42;!F14$
SKIP 3$
       2=*WTC,'00,'0A0D;!F14$
       28=*WTA,'00,ER2;!F14$
       JPA,     ,ENDE;!F14$
$
```

CONC BIT(length) {reference 1/2}ø

Beim Verkettungsbefehl für Zeichenfolgen wird nur der Name und die Zeichenzahl des Operanden unter CHAR2 bzw. CHARL2 im STAGE 2-Speicher eingetragen. Diese beiden Informationen werden zusammen mit dem beim Laden einer CHAR-Größe eingetragenen Informationen beim nachfolgenden STORE-Befehl abgefragt, der erst die eigentliche Verkettung bewirkt.

```
CONC CHAR(!) ! !()  $
CHAR2 EQU !30$
CHARL2 EQU !10$
$
```

CONC CHAR(length) {reference 1/2}ø

Beim STORE-Befehl werden die zu verknüpfenden Zeichenketten nacheinander Zeichen für Zeichen in den Akkumulator geladen und von dort aus nach dem bei der Basisadresse Symbol beginnenden Adressbereich geladen.

```
STORE CHAR(!) ! !()  $
       4<2LDC,'42,0;!F14$
CHARL1!F7$
MEMORYCH CHAR1 !30 '42$
!F8$
CHARL2!F7$
MEMORYCH CHAR2 !30 '44$
       2=*ADC,'44,1;!F14$
!F8$
$

MEMORYCH ! ! !$
       LDA,@,!10,!30;!F14$
       STA,@,!20,'42;!F14$
       2=*ADC,'42,1;!F14$
$
```

STORE CHAR(length) {reference 1/2}ø

3.7 Vergleichsbefehle.

Beim Vergleichsbefehl wird ein Kennwort im Akkumulator abgelegt, das der nachfolgende bedingte Sprungbefehl auswertet.
Beim Vergleich von INT-, DUR-, CLO- und REAL-Größen wird der Operand vom Akkumulatorinhalt subtrahiert.

```
CMP REAL ! !  $
SE MODE REAL !20     $
       3<*BZ,@,      ,Z!00;!F14$
       LDE,@,'04;!F14$
       ORC,@,'01;!F14$
Z!00:  NOP;!F14$
$
```

Darstellung von REAL-Größen nach Bild 13. Wenn die Mantisse des Ergebnisses (REAL-Größe) ungleich Null ist, wird das Vorzeichen-Bit von Reg. '04 nach Bit 7 in Reg. '02 gebracht. Außerdem wird zur sicheren Kennzeichnung eines positiven Ergebnisses der Subtraktion Bit 0 in Register '02 nach 1 gesetzt.

CMP REAL {reference 1/2} $

Entsprechendes gilt für 2-Byte-Ganzzahlen.

```
CMP ! ! !  $
SE MODE !10 !30     $
       2<*BZ,@,     ,Z!00;!F14$
       LDE,@,'03;!F14$
       ORC,@,'01;!F14$
Z!00:  NOP;!F14$
$
```

CMP {INT / DUR / CLO} {reference 1/2} $

Beim Vergleich von BIT-Größen (2-Byte-Darstellung) werden Operand und Akkumulator (2 Bytes) mit Exklusiv-ODER verknüpft. Bei Gleichheit haben Register '02 und '03 Nullinhalt. Durch anschließende ODER-Verknüpfung wird in Register '02 ein entsprechendes Kennwort abgelegt. (0 bei Gleichheit, ≠0 bei Ungleichheit)

```
CMP BIT(!) ! !  $
FO MODE (!) ! !    $
       ORR,@,'03;!F14$
$
```

CMP BIT(length) {reference 1/2} $

Entsprechend wird beim Vergleich von CHAR-Größen verfahren, wobei die Ketten zeichenweise mit Exklusiv-ODER verknüpft werden. Der Vergleich wird nur bei gleicher Zeichenzahl ausgeführt und bei den ersten beiden ungleichen Zeichen abgebrochen. Register '02 hat dann einen Inhalt ≠ 0.

```
CMP CHAR(!) ! !()  $
IF !10 EQUAL CHAR1 SKIP 2$
       LDC,@,1;!F14$
SKIP 2$
        2<*LDC,'44,0;!F14$
COMPCHAR CHAR1 !30 !10$
```

CMP CHAR(length) {reference 1/2} $

```
COMPCHAR ! ! !$
Z!01:  LDA,@,!11,'44;!F14$
       LDA,'42,!20,'44;!F14$
       EOR,@,'42;!F14$
       BNZ,@,     ,Z!00;!F14$
       2=*INEC,'44,!30,Z!01;!F14$
Z!00:  NOP;!F14$
$
```

3.8 Sprungbefehle.

Register '02, in dem durch einen vorhergehenden Vergleichsbefehl ein Kennwort abgelegt wurde, wird auf Nullinhalt, positives oder negatives Vorzeichen abgefragt. Das Kennwort wird nicht verändert und steht einem nachfolgenden Sprungbefehl noch zur Verfügung.

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

Direkte Angabe der Sprungadresse:

opcode,I INSTR CODE symbol() ø

wobei symbol eine durch LOC (vgl. 3.9.1) definierte Marke ist;
oder Referenz zu einem Speicherplatz, der die Sprungadresse enthält (vgl. 3.9.7).

opcode ADDR CODE {reference 2} ø

Zur Erzeugung des Assemblercodes für den DIETZ 621 sind die folgenden Macros nötig, die unbedingte Sprungbefehle enthalten. Diese werden durch die entsprechenden CIMIC/1-Sprungbefehle in Abhängigkeit vom in Register '02 abgelegten Kennwort aufgerufen.

```
JPA,I INSTR CODE !() $
      JPA,     ,!10;!F14$
$
```

opcode,I INSTR CODE symbol() ø

Die durch eine Referenz zu einem Speicherplatz angegebene Sprungadresse wird in die Register '42 und '43 geladen, auf die sich der unbedingte Sprungbefehl über indirekte Adressierung bezieht.

```
JPA ADDR CODE !() $
      2=&LDA,'42,!10;!F14$
      JPX,     ,'42;!F14$
$
```

```
JPA ADDR CODE !(!) $
      2=&LDA,'42,!10;!F14$
!20MEM$
FIX OFFSET HILF$
      JPX,     ,'42;!F14$
$
```

Der Fix offset bzw. der Inhalt Indexregister oder beide werden der durch symbol angegebenen Adresse addiert.

```
FIX OFFSET !$
       2=*ADC,'42,!11;!F14$
$
JPA ADDR CODE !()+  $
       2=&LDA,'42,!10;!F14$
       2=*ADR,'42,'40;!F14$
       JPX,     ,'42;!F14$
$
```

opcode ADDR CODE { reference 2 } Ø

```
JPA ADDR CODE !()+  $
       2=&LDA,'42,!10;!F14$
!20MEM$
FIX OFFSET FILE$
       2=*ADR,'42,'40;!F14$
       JPX,     ,'42;!F14$
$
```

Für den unbedingten Sprungbefehl gilt somit:

```
JMP!  $
JPA!10  $
$
```

JMP { Referenz zum Sprungziel } Ø

Die bedingten Sprungbefehle fragen zuerst das in Register '02 stehende Kennwort ab und bewirken bei erfüllter Bedingung einen unbedingten Sprung.

Sprung bei Gleichheit

```
JEQ!  $
       BNZ,@,     ,Z!00;!F14$
JPA!10  $
Z!00:  NOP;!F14$
$
```

JEQ { Referenz zum Sprungziel } Ø

Sprung bei Ungleichheit

```
JNE!  $
       BZ,@,     ,Z!00;!F14$
JPA!10  $
Z!00:  NOP;!F14$
$
```

JNE { Referenz zum Sprungziel } Ø

Sprung, wenn der Akkumulatorinhalt größer oder gleich dem Operanden des CMP-Befehls war (Bit 7 Reg. '02 = 0).

```
JGE!  $
       BNP,@,     ,Z!00;!F14$
JPA!10  $
Z!00:  NOP;!F14$
$
```

JGE { Referenz zum Sprungziel } Ø

Sprung, wenn der Akkumulatorinhalt kleiner oder gleich dem Operanden war. (Bit 7 Reg. '02 = 0 oder alle Bits = 0).

```
JLE!  $
       BP,@,     ,Z!00;!F14$
JPA!10  $
Z!01:  NOP;!F14$
JEQ!10  $
$
```

JLE { Referenz zum Sprungziel } Ø

Sprung, wenn der Akkumulatorinhalt größer als der Operand war. (Bit 7 = 0, aber nicht alle anderen Bits = 0).

```
JGT!  $
       BNP,@,     ,Z!00;!F14$
       BZ,@,      ,Z!00;!F14$
JPA!10  $
Z!00:  NOP;!F14$
$
```

JGT { Referenz zum Sprungziel } Ø

Sprung, wenn der Akkumulatorinhalt kleiner als der Operand war (Bit 7 ≠ 0)

```
JLT! $
      BP,@,     ,Z!00;!F14$
JPA!10 $
Z!00: NOP;!F14$
$
```

JLT {Referenz zum Sprungziel} ø

3.9 Deklarationen.

3.9.1 Marken.

CIMIC/1-Marken werden an eine Leeranweisung gebunden.

```
LOC   !() $
!F14$
11111: NOP;$
$
```

LOC symbol() ø

3.9.2 Platzreservierung ohne Initialisierung.

Als Schlüsselwörter sind nur LOCAL und MODUL möglich. Beide werden gleich behandelt.
Für INT, CLO, DUR, BIT und REAL-Größen wird eine dem Wert des MODE-Identifiers (im STAGE 2-Speicher abgelegt) entsprechende Anzahl von Bytes reserviert und mit Null vorbelegt.

```
SPACE ! ! !() $
IF !11 NE  SKIP 1$
BIT!16$
!30:   !11*V;!F14$
$
```

SPACE { REAL / INT / CLO / DUR / BIT() } { MODUL / LOCAL } symbol() ø

CHAR-Größen werden mit 1 Byte je Zeichen dargestellt. Es wird eine der Länge der Zeichenkette entsprechende Zahl von Bytes reserviert und mit Null vorbelegt.

```
SPACE CHAR(!) ! !() $
!30:   !10*V;!F14$
$
```

SPACE CHAR(length) { MODUL / LOCAL } symbol() ø

3.9.3 Platzreservierung mit Initialisierung.

Hier wird für eine Größe durch eine dem jeweiligen MODE entsprechende Belegungsanweisung eine Anzahl von Bytes mit dem angegebenen Wert vorbelegt, der gegebenenfalls mittels dem STAGE 2 in eine für den DIETZ direkt ladbare Form gebracht wird.

Außerdem ist vorgesehen, daß bei Feldelementen (Schlüsselwort ELEM) kein Name vorgestellt wird.

```
SPACE INT ! !() A !$
IF !10 = ELEM SKIP 2$
!20:!26$
SKIP 1$
        !26$
!20     2*D,!34;!F14$
$
```

SPACE INT { CONST | LOCAL | VLLOC | MODUL | VLMOD } symbol() A value$

```
SPACE INT ! !() E !$
!SOMEM$
SPACE INT !10 !20() A ELEM$
$
```

SPACE INT { CONST | LOCAL | VLLOC | MODUL | VLMOD } symbol() E value$

```
SPACE REAL ! !() A !$
IF !10 = ELEM SKIP 2$
!20:!26$
SKIP 1$
        !26$
!20     G,!30;!F14$
$
```

SPACE REAL { CONST | LOCAL | VLLOC | MODUL | VLMOD } symbol() A value$

```
SPACE CHAR(!) ! !() A !$
IF !20 = ELEM SKIP 2$
!30:!36$
SKIP 1$
        !36$
!30     !10*T,"!40";!F14$
$
```

SPACE CHAR(length) { CONST | LOCAL | VLLOC | MODUL | VLMOD } symbol() A value$

Bei CLO- und DUR-Größen, für die 2 Bytes reserviert werden, wird der Wert mittels STAGE 2 in Sekunden umgerechnet. Hierzu werden die in Abschnitt 3.3 beschriebenen Hilfsmacros verwendet. Im STAGE 2 - Speicher wird in Adresse DURTEST 1 abgelegt. Diese Speicheradresse kann bei Feldreservierung mit Initialisierung abgefragt werden, womit die mehrmalige Umrechnung des gleichen Werts in Sekunden vermieden werden kann.

```
SPACE CLO ! !() A !:!:!$
EVALDUR !30*3600+!40*60+!50$
CLOSPACE !10 !20 STACKDUR$
$

CLOSPACE ! ! ! $
IF !10 = ELEM SKIP 2$
!20:!26$
SKIP 1$
        !26$
!20     2*D,!31;!F14$
DURTEST EQU 1$
$
```

SPACE CLO { CONST | LOCAL | VLLOC | MODUL | VLMOD } symbol() A clock$

```
SPACE DUR ! !() A !$
DURATION !30$
CLOSPACE !10 !20 STACKDUR$
$
```

SPACE DUR {CONST / LOCAL / VLLOC / MODUL / VLMOD} sybol() A duration∅

Bei BIT-Größen wird aus dem Bitmuster mittels STAGE 2 eine Hexadezimalzahl berechnet (vgl. 3.3). Die mehrmalige Umrechnung des gleichen Bitmusters bei Feldreservierung mit Initialisierung wird durch Ablegen von 1 in Adresse BITTEST im STAGE 2 - Speicher verhindert.

```
SPACE HEXA ! ! ! ! ! !$
IF !10 = ELEM SKIP 2$
!20:!26$
SKIP 1$
       !26$
!20    2*H,'!31!41!51!61;!F14$
$
```

SPACE BIT(length) {CONST / LOCAL / VLLOC / MODUL / VLMOD} symbol() A ≠ bit-value≠ B1∅

Auf die Verarbeitung von Oktalzahlen wurde verzichtet.

```
SPACE BIT(!) ! !() A *!*B1$
NULLSETZEN 16$
BREAK BITLIST !40 17-!10$
BREAK HEXA B1 B2 B3 B4*B5 B6 B7 B8*B9 B10 B11 B12*B13 B14 B15 B16$
SPACE HEXA !20 !30 B1 B2 B3 B4$
BITTEST EQU 1$
$
```

Da der DIETZ 621 Konstantenoperationen für 2-Byte Ganzzahlen und Hexazahlen direkt ausführen kann, könnte durch Verwendung entsprechender Macros auf die Platzreservierung für Konstanten vom MODE INT, CLO, DUR und BIT verzichtet werden.

3.9.4 Feldreservierung ohne Initialisierung.

Mit bereits in Abschnitt 3.2 beschriebenen Macros wird durch den STAGE 2 die Zahl der zu reservierenden Bytes, die mit 0 vorbelegt werden, aus der Zahl der Feldelemente und dem Wert des MODE-Identifiers berechnet.

```
SPACE ! ! !(!)  $
!40MEM$
FIELDSPACE !10 !30 HILF$
$

FIELDSPACE ! ! !$
IF !10 = ELEM SKIP 2$
!20:!26$
SKIP 1$
       !26$
!20    !31*V;!F14$
$
```

SPACE {REAL / INT / BIT / DUR / CLO / CHAR} {MODUL / LOCAL} Symbol(length)∅

3.9.5 Feldreservierung mit einheitlicher Initialisierung.

Hierzu werden für jedes Feldelement die in Abschnitt 3.9.3 beschriebenen Macros aufgerufen, wobei mit dem STAGE 2 eine Iteration über die Zahl der Feldelemente durchgeführt wird. Nur dem ersten Element eines Feldes wird dessen Name zugeordnet (Schlüsselwort ELEM). Durch Abfragen des Inhalts der STAGE 2 - Speicheadressen BITTEST und DURTEST werden überflüssige Umrechnungen vermieden (vgl. 3.9.3).

```
SPACE ! ! !(!*!) A !$
DURTEST EQU 0$
BITTEST EQU 0$
!40!F?$
IF DURTEST EQUAL 0 SKIP 2$
CLOSPACE !20 !30 STACKDUR$
SKIP 5$
IF BITTEST EQUAL 0 SKIP 2$
SPACE HEXA !20 !30 B1 B2 B3 B4$
SKIP 2$
SPACE !10 !20 !30() A !60$
ELEM!26$
!F8$
$
```

SPACE { REAL / INT / BIT / DUR / CLO / CHAR } { MODUL / LOCAL } symbol(length)
A value Ø

3.9.6 Feldreservierung mit unterschiedlicher Initialisierung.

Hierbei wird beim einleitenden Befehl nur der Name in den STAGE 2-Speicher eingetragen. Dieser wird nach dem ersten Feldelement mit Blanks überschrieben. Außerdem wird beim ersten Feldelement als Schlüsselwort für die Referenz ELEM eingetragen.

```
SPACE ! ! !(!*!)+  $
FELD EQU !30$
FEL EQU !20$
$
```

SPACE { REAL / INT / DUR / CLO / BIT / CHAR } { MODUL / LOCAL } symbol(length)+ Ø

Bei Feldelementen werden die in 3.9.3 aufgeführten Macros aufgerufen mit Name "Blanks" und Schlüsselwort "ELEM".

```
SPACE ! ! (!)  $
MEMFELDO !10 FEL FELD !30$
FELD EQU  $
FEL EQU ELEM$
$

MEMFELDO ! ! ! !$
SPACE !10 !21 !31(!40)  $
$
```

SPACE { REAL / INT / DUR / CLO / BIT / CHAR } { MODUL / LOCAL } (length) Ø

```
SPACE ! ! (!*!) A !$
MEMFELD1 !10 FEL FELD !30 !40 !50$
FELD EQU   $
FEL EQU ELEM$
$

MEMFELD1 ! ! ! ! ! !$
SPACE !10 !21 !31(!40*!50) A !60$
$

SPACE ! ! (!*!) E !*!  $
IF !61 NE  SKIP 1$
BIT!66$
!50*!61!56$
MEMFELD1 !10 FEL FELD !30 !40 !54$
FELD EQU   $
FEL EQU ELEM$
$
```

SPACE { REAL / INT / DUR / CLO / BIT / CHAR } { MOLUL / LOCAL } (length) A value Ø

3.9.7 Adreßfelder

Adreßfelder werden entsprechend 3.9.6 behandelt. Es werden 2 Bytes reserviert und mit den angegebenen Adressen belegt. Die gespeicherten Adressen werden z.B. als Sprungziel für die in 3.8 beschriebenen Sprungbefehle verwendet.

```
SPACE ADDR ! !(!*!)+  $
FELD EQU !20$
FEL EQU ADDR$
$
```

SPACE ADDR CODE [symbol] ([size])

{ + / A value } Ø

```
SPACE ADDR ! () A !$
MEMADDR FEL FELD !20$
FELD EQU   $
FEL EQU ELEM$
$

MEMADDR ! ! !$
IF !10 = ELEM SKIP 2$
!21:!26$
SKIP 1$
      !26$
!20   2*A,!30;!F14$
$
```

.10 Testprogramm.

Zur Erzeugung einer lauffähigen Assemblerversion eines bereits in CIFIC/1 übersetzten PEARL-Testprogramms zur Matrizenmultiplikation waren noch folgende weitere Macros nötig.

```
TART   !() $
:F EQU !10$
)000 EQU 0$
)001 EQU 1$
)010 EQU 2$
)011 EQU 3$
)100 EQU 4$
)101 EQU 5$
)110 EQU 6$
)111 EQU 7$
1000 EQU 8$
1001 EQU 9$
1010 EQU A$
1011 EQU B$
1100 EQU C$
1101 EQU D$
1110 EQU E$
1111 EQU F$
INT EQU 2$
CLO EQU 2$
DUR EQU 2$
AIT EQU 2$
REAL EQU 4$
       0,'4000;!F14$
WK:   4*C,'00007FFF;!F14$
NN1:   28*T,"*FEHLER: INT.-ZAHL ZU GROSS*";!F14$
NR2:   28*T,"**FEHLER: BITKETTEN ZU LANG*";!F14$
$
```

Mit diesem Macro, das zu Beginn des Testprogramms aufgerufen wird, werden alle zur Übersetzung nötigen Größen im STAGE 2 - Speicher eingetragen (z.B Zuordnung von Bitmustern zu Hexadezimalzahlen, Wert von MODE-Identifiern). Es erzeugt außerdem den Code für den Anfang eines Assemblerprogramms und für den im ganzen Programm benötigten Text von Fehlermeldungen wird Speicherplatz vorbelegt.

START symbol() ;

```
/*!*/  $
/!10;!F14$
$
```

Kommentarmacro

```
FINISH   !() $
END ANF$
$

END !$
ENDE : HLT;!F14$
       JPA,    ,!11;!F14$
Y;!F14$
!F0$
$$
```

Mit diesen Macros wird der Code für das Ende eines Assemblerprogramms erzeugt.

FINISH symbol() ;

```
LINK TASK MODUL !(!)  $
      NOP;!F14$
$
BEGIN TASK MODUL !(!)  $
      NOP;!F14$
$
LEVEL    !$
$
TERM TASK MODUL !(!)  $
      NOP;!F14$
$
END TASK MODUL !(!)  $
      NOP;!F14$
$
```

Für eine Reihe von CIMIC/1-Befehlen, für die kein bestimmter Assemblercode nötigt ist, wird eine Leeranweisung erzeugt.

Zur Ausgabe von Größen auf dem Teletype wurden folgende Ausgabemacros geschrieben:
Jede Ausgabe beginnt mit Wagenrücklauf und Zeilenvorschub.

```
FFORM$
      2=*WTC,'00,'0A0D;!F14$
15.5!$WERG,'00,'02;!F14$
$
```

Ausgabe der im Akkumulator stehenden Gleitkommazahl.

```
FFORM$
      2=*WTC,'00,'0A0D;!F14$
6!$WDRD,'00,'02;!F14$
$
```

Ausgabe der im Akkumulator stehenden Ganzzahl.

```
FIR$
      2=*WTC,'00,'0A0D;!F14$
6!$WDRD,'00,'40;!F14$
$
```

Ausgabe einer im Indexregister stehenden Ganzzahl.

```
FTEXT$
      2=*WTC,'00,'0A0D;!F14$
      LDC,'42,0;!F14$
TEXT CHARL1-1 CHAR1$
$
TEXT ! !$
CHARL1!F7$
Z!00:  WTA,'00,!20,'42;!F14$
       2=*ADC,'42;!F14$
       BNEC,'42,!14,Z!00;!F14$
$
```

Ausgabe von Text. Der der Anfangsadresse entsprechende Name ist unter CHAR1 im STAGE 2 - Speicher abgelegt.

```
FBIT$
      2=*WTC,'00,'0A0D;!F14$
      2=*WHR,'00,'03;!F14$
      2=*WHR,'00,'02;!F14$
$
```

Ausgabe der Bitkette im Akkumulator.

```
FCLO$
FFORM$
      4=*WTC,'00,"SEC";!F14$
$
FDUR$
FCLO$
$
```

Ausgabe einer im Akkumulator stehenden CLO- bzw. DUR-Größe in Sekunden.

4. Zusammenfassung

In dieser Arbeit wurde der von der ASME zur Übersetzung von CIMIC/1 in die Assemblersprache eines Zielrechners gewählte Macroübersetzer STAGE 2 untersucht.

Seine Eigenschaften sind in Abschnitt 2.3 nach /1/ ausführlich beschrieben und konnten an Hand von Anwendungsbeispielen mit einer FORTRAN-Version des Übersetzers auf der CDC 6600 bestätigt werden. Es hat sich hierbei herausgestellt, daß der Macroübersetzer STAGE 2 die in Abschnitt 2.2 zusammengestellten Forderungen an einen Übersetzer weitgehend erfüllt und durch fest einprogrammierte Übersetzerfunktionen und Parameterumformung ausreichend viele Möglichkeiten zur Codeoptimierung bietet.

Der Vergleich mit anderen Macroübersetzern in Abschnitt 2.5 zeigt, daß der STAGE 2 von den dort betrachteten Übersetzern wegen seiner Sprachunabhängigkeit, seiner Benutzerfreundlichkeit und vor allem wegen seiner Portabilität am ehesten zur Übersetzung von CIMIC/1 in Assemblercode in Frage kommt. Unbefriedigend bei diesem Vergleich und bei im Rahmen dieser Arbeit durchgeführten Rechenzeituntersuchungen war, daß keine anderen auf der CDC 6600 lauffähigen Übersetzer vorlagen, sodaß keine Aussagen über Rechenzeit- und Speicherbedarf des STAGE 2 in Vergleich zu diesen gemacht werden konnten.

Aus den in Abschnitt 2.5 zusammengestellten Ergebnissen von Rechenzeituntersuchungen ergaben sich Anhaltspunkte für die bezüglich Rechenzeit optimale Erstellung eines Codegenerators mit dem STAGE 2 und Richtwerte für den Rechenzeitbedarf von Übersetzungen.

Der in Abschnitt 3 beschriebene Macroteil eines Codegenerators zur Übersetzung von CIMIC/1-Programmen in Assemblercode wurde auf der CDC 6600 mit einem Testprogramm zur Matrizenmultiplikation überprüft (siehe Anhang). Allerdings ist der hierbei abge-

setzte Code noch nicht optimal und enthält zum Teil noch überflüssige Anweisungen, die z.B. durch einen Optimierungslauf beseitigt werden könnten. Zur Übersetzung von Programmen mit einem solchen Codegenerator direkt auf einem Kleinrechner muß versucht werden, den STAGE 2 ebenfalls in den Assemblercode des jeweiligen Rechners zu übersetzen, da die FORTRAN-Version einen für diese Rechnerklasse unerträglichen Aufwand an Rechenzeit und Speicherplatz erfordet.

LITERATURVERZEICHNIS:

/1/ P.C. Poole, W.M. Waite. The STAGE 2 Macroprocessor user Reference Manual.

/2/ M.D. McIlroy. Macro extensions of compiler languages. Comm. ACM. 3 (April, 1960).

/3/ M.I. Halpern. XPOP: a metalanguage without metaphysics. Proc. AFIPS 1964 Fall Joint Computer Conference, Vol. 26, p. 57.

/4/ C. Strachey. A general purpose macrogenerator. Computer J., 8 (October, 1965) p. 225.

/5/ C.N. Mooers. TRAC, a procedure-describing language for the reactive type writer. Comm. ACM, 9 (March, 1966) p. 215.

/6/ W.M. Waite. A language-independent macro processor. Comm. ACM, 10 (July, 1967).

/7/ P.J. Brown. The MI/I macro processor. Comm. ACM, 10 (October, 1967) p. 618.

/8/ R.J. Orgass, W.M. Waite. A base for a mobile programming system. Comm. ACM, 12 (September, 1969) p. 507.

/9/ R.E. Griswold, I.P. Polensky. String pattern matching in the programming language SNOBOL. Unveröffentlicht.

/10/ P.J. Brown. MI/I user's manual. University Math. Lab. Cambridge, England, July 1968.

/11/ Siemens 330, Benutzerhandbuch.

/12/ H.-D. Worm. Makro-Generator System für EPR 1100. Krupp Atlas-Elektronik, Bremen Juli 1974.

/13/ D. Gries. Compiler construction for digital computers. Cornell University. (John Wiley & Sons, Inc., New York).

/14/ ASME-interne Notiz: CIMIC/1 vom Juli 1974.

/15/ ASME-interne Notiz: PEARL-Subset für die erste Entwicklungsstufe. (30. 05. 1974).

/16/ M. Alt. Beschreibung der Code-Erzeugung für den Prozeßrechner AEG 60-10 aus der Prozeßrechnersprache PEARL. ASME-Bericht vom 3. 10. 1974.

/17/ DIETZ 621, Handbuch 4/74.

/18/ J. Brandes, S. Eichentopf, P. Elzer, L. Frevert, V. Haase, H. Mittendorf, G. Müller, P. Rieder. PEARL, the concept of a process- and experiment-oriented programming language. Elektronische Datenverarbeitung, Heft 10/1970

/19/ B. Eichenauer, V. Haase, P. Holleczek, K. Kreuter, G. Müller. PEARL, eine prozeß- und experimentierte Programmiersprache. Angewandte Informatik 9/73.

Anhang I: PEARL - Testprogramm (matrizenmultiplikation)

PEARL-SUBSET LISTING

```
 1   MODULE MATMU;
 2   PROBLEM;
 3     T01:TASK
 4   DECLARE A(2,3) FLOAT INITIAL (11.,21.,12.,22.,13.,23.);
 5   DECLARE B(3,1) FLOAT INITIAL (1.,2.,3.);
 6   DECLARE R(2,1) FLOAT;
 7   DECLARE (I,J,K) FIXED;
 8   DECLARE (M,N,L) FIXED;
 9   M=2;
10   N=3;
11   L=1;
12   FOR J TO M REPEAT  /* S1 */
13   FOR K TO L REPEAT  /* S2 */
14   R(J,K)=0.;
15   FOR I TO N REPEAT  /* S3 */
16   R(J,K)=R(J,K)+A(J,I)*B(I,K);
17   END;       /* S3 */;
18   END;       /* S2 */;
19   END;       /* S1 */;
20   END;.  /* TASK T01 */
21   MODEND;
22   S
```

Anhang II: Macroteil

```
$!$!0 (+-*/)
! EQU !$
!F3$
$
SKIP !$
!F4$
$
IF ! = ! SKIP !$
!F50$
$
IF ! NE ! SKIP !$
!F51$
$
IF ! EQUAL ! SKIP !$
!F60$
$
IF ! GT ! SKIP !$
!F6+$
$
! ! REF 12 ! !() A !  $
IF !30 = D SKIP 2$
        !10A!30,!20,!40;!F14$
SKIP 3$
IF !50 NE HILF SKIP 1$
!51!56$
        !10C!30,!20,!50;!F14$
$
! ! REF 12 ! !() E !  $
!50MEM$
HILF!56$
        !10C!30,!20,!51;!F14$
$
!*!MEM$ .
IF !21 NE  SKIP 1$
BIT!26$
!10*!21!16$
HILF EQU !14$
$
!*CHAR(!)MEM$
!10*!20!16$
HILF EQU !10$
$
!MEM$
HILF EQU !11$
$
CHAR(!)MEM$
HILF EQU !10$
$
BIT(!)MEM$
HILF EQU 2$.
$
! ! REF 12 ! !()    $
        !10A!30,!20,!40;!F14$
$
! ! REF 12 ! !()+    $
        !10A!30,!20,!40,'40;!F14$
$
```

```
! ! REF 12 ! !(!)    $
!50MEM$
LD '42 REF 12 D   () A HILF  $
       !10A!30,!20,!40,'42;!F14$
$
! ! REF 12 ! !(!)+    $
!50MEM$
LD '42 REF 12 D   () A HILF  $
       2=*ADR,'42,'40:!F14$
       !10A!30,!20,!40,'42;!F14$
$
! MODE ! !  $
!10 @ REF 12 D !30  $
$
! MODE REAL !  $
!10 @ REF 12 G !20  $
$
! MODE CLO !() A !:!:!  $
EVALDUR !30*3600+!40*60+!50$
!10 @ REF 12 D !20() A STACKDUR  $
$
STN !$
STORE !10$
$
EVALDUR !$
STACKDUR EQU !14$
$
! MODE BIT(!) !  $
BITL EQU !20$
!10 @ REF 12 D !30  $
$
! MODE DUR !() A !  $
DURATION !50$
!10 @ REF 12 D !20() A STACKDUR  $
$
DURATION !HRS$
EVALDUR !10*3600$
$
DURATION !HRS !MIN$
EVALDUR !10*3600+!20*60$
$
DURATION !HRS !SEC$
EVALDUR !10*3600+!20$
$
DURATION !HRS !MIN !SEC$
EVALDUR !10*3600+!20*60+!30$
$
DURATION !MIN$
EVALDUR !10*60$
$
DURATION !MIN !SEC$
EVALDUR !10*60+!20$
$
DURATION !SEC$
EVALDUR !10$
$
! MODE BIT(!) !() A *!*B1  $
BITL EQU !20$
NULLSETZEN 16$
BREAK BITLIST !40 17-!20$
BREAK HEXA B1 B2 B3 B4*B5 B6 B7 B8*B9 B10 B11 B12*B13 B14 B15 B1($
STORE HEXA B1 B2 B3 B4$
'10 @ REF 12 D !30() A HILF2  $
$
NULLSETZEN !$
1 EQU 0$
!10!F7$
B1+1 EQA 0$
!F8$
$
B! EQA !$
B!14 EQU !20$
```

```
I EQU !14$
$
BREAK BITLIST ! !$
I EQU !24$
!10!17$
BI+1 EQA !10$
!F8$
$
BREAK HEXA !$
I EQU 0$
!10!17"$
HEXAMEM1 !10$
!F8$
$
HEXAMEM1 ! ! ! !$
HEXAMEM2 !11!21!31!41$
$
HEXAMEM2 !$
BI+1 EQA !11$
$
STORE HEXA ! ! ! !$
HILF2 EQU '!11!21!31!41$
$
LOAD ! ! !$
LD MODE !10 !30  $
$
LOAD CHAR(!) ! !()  $
NULL EQU 0$
CHARL1 EQU !20$
CHAR1 EQU !30$
$
STORE ! ! !$
ST MODE !10 !30  $
$
STORE REAL ! ! C !  $
UMWANDLUNG INT REAL$
ST @ REF 12 C !20  $
$
UMWANDLUNG INT REAL$
       LDR,'04,'03;!F14$
       LDC,'05,0;!F14$
       ANC,'04,'80;!F14$
       ANC,'03,'7F;!F14$
$
STORE ! ! ! C REAL  $
UMWANDLUNG REAL INT$
ST @ REF 12 D !30  $
$
UMWANDLUNG REAL INT$
       STRG,@,'42;!F14$
       AB;!F14$
       SBCG,@,'7FFF;!F14$
       BNP,'05,    ,Z!00;!F14$
       2=*WTC,'00,'0A0D;!F14$
       28=*WTA,'00,FR1;!F14$
        JPA,    ,ENDE;!F14$
Z!00:  LDRG,@,'42;!F14$
       IT;!F14$
$
STORE ! ! ! C !  $
ST @ REF 12 D !30  $
$
LDX INT ! !$
LD '40 REF 12 D !20  $
$
STX INT ! !$
ST '40 REF 12 D !20  $
$
ABS REAL    $
       AB;!F14$
$
ABS INT     $
```

```
       BP,'03,     ,Z!00;!F14$
NEG INT    $
Z!00:  NOP;!F14$
$
NEG REAL    $
       ADC,'04,'80;!F14$
$
NEG INT    $
VORZEICHENUMKEHR @$
$
VORZEICHENUMKEHR !$
       2=*SBC,!10,'0001;!F14$
       2<EOC,!10,'FF;!F14$
$
ADD ! ! !$
AD MODE !10 !30  $
$
SUB ! ! !$
SB MODE !10 !30  $
$
MPY ! ! !$
MP MODE !10 !30  $
IF !11 NE 2 SKIP 7$
       4=*STR,@,'46;!F14$
       4=*SBC,@,PHK;!F14$
       BNP,'05,     ,Z!00;!F14$
       2=*WTC,'00,'0A0D;!F14$
       28=*WTA,'00,ER1;!F14$
       JPA,     ,ENDE;!F14$
Z!00:  4=*LDR,@,'46;!F14$
       ORR,'03,'05;!F14$
$
DIV ! ! !$
DV MODE !10 !30  $
IF !11 NE 2 SKIP 3$
       BP,'03,     ,Z!00;!F14$
       2=*ADC,'04,1;!F14$
Z!00:  LDRD,@,'04;!F14$
$
EXP REAL ! !$
PO MODE REAL !20  $
$
EXP ! ! !$
UMWANDLUNG INT REAL$
       4=*STR,@,'42;!F14$
LD MODE !10 !30  $
UMWANDLUNG INT REAL$
ST MODE !10 !30  $
PO MODE REAL !30  $
UMWANDLUNG REAL INT$
$
ADX INT ! !$
AD '40 REF 12 D !20  $
$
SBX INT ! !$
SB '40 REF 12 D !20  $
$
MPX INT ! !$
       4=*STR,@,'42;!F14$
       2=*LDR,@,'40;!F14$
MPY INT !10 !20$
       2=*STR,@,'40;!F14$
       4=*LDR,@,'42;!F14$
$
ADDAX INT    $
       2=*ADR,'40,'02;!F14$
$
NOT BIT(!)    $
       2<?EOC,@,'FF;!F14$
$
AND ! ! !  $
AN MODE !10 !30  $
```

```
$
OR ! ! !  $
OR MODE !10 !30  $
$
FXOR ! ! !  $
EO MODE !10 !30  $
$
SHIFT INT ! !  $
LD '42 REF 12 D !30  $
       FP,'43,     ,Z!00;!F14$
VORZEICHENUMKEHR '42$
Z!01:  2<*SRO,@;!F14$
       2=*SBC,'42,1;!F14$
       2<*BNZ,'42,     ,Z!01;!F14$
       JPA,     ,Z!03;!F14$
Z!00:  2<*SLO,@;!F14$
       2=*SBC,'42,1;!F14$
       2<*BNZ,'42,     ,Z!00;!F14$
Z!03:  NOP;!F14$
$
RBIT INT ! !  $
LD '42 REF 12 D !30  $
Z!00:  BEC,'42,1,Z!01;!F14$
       2=*SBC,'42,1;!F14$
       2<*SRO,@;!F14$
       JPA,     ,Z!00;!F14$
Z!01:  NOP;!F14$
$
LBIT INT ! !  $
LD '42 REF 12 D !30  $
Z!00:  2<*BZ,'42,     ,Z!01;!F14$
       2=*SRC,'42,1;!F14$
       2<*SLC,@;!F14$
       JPA,     ,Z!00;!F14$
Z!01:  NOP;!F14$
$
CONC BIT(!) ! !  $
IF !10+BITL GT 16 SKIP 5$
LD '42 REF 12 D !30  $
!10!F7$
       2<*SLO,@;!F14$
!F8$
       2=*ORR,@,'42;!F14$
SKIP 3$
       2=*W1C,'00,'0A0D;!F14$
       28=*WTA,'00,ER2;!F14$
       JPA,     ,ENDE;!F14$
$
CONC CHAR(!) ! !()  $
CHAR2 EQU !30$
CHARL2 EQU !10$
$
STORE CHAR(!) ! !()  $
       4<?LDC,'42,0;!F14$
CHARL1!F7$
MEMORYCH CHAR1 !30 '42$
!F8$
CHARL2!F7$
MEMORYCH CHAR2 !30 '44$
       2=*ADC,'44,1;!F14$
!F8$
$
MEMORYCH ! ! !$
       LDA,@,!10,!30;!F14$
       STA,@,!20,'42;!F14$
       2=*ADC,'42,1;!F14$
$
CMP REAL ! !  $
SB MODE REAL !20     $
       3<*BZ,@,     ,Z!00;!F14$
       LDR,@,'04;!F14$
       ORC,@,'01;!F14$
```

```
Z!00:  NOP;!F14$
$
CMP ! ! !  $
SB MODE !10 !30    $
       2<*BZ,@,    ,Z!00;!F14$
       LDR,@,'03;!F14$
       ORC,@,'01;!F14$
Z!00:  NOP;!F14$
$
CMP BIT(!) ! !  $
EO MODE (!) ! !    $
       ORR,@,'03;!F14$
$
CMP CHAR(!) ! !()  $
IF !10 EQUAL CHARL1 SKIP 2$
       LDC,@,1;!F14$
SKIP 2$
        2<*LDC,'44,0;!F14$
COMPCHAR CHAR1 !30 !10$
$
COMPCHAR ! ! !$
Z!01:  LDA,@,!11,'44;!F14$
       LDA,'42,!20,'44;!F14$
       EOR,@,'42;!F14$
       BNZ,@,    ,Z!00;!F14$
       2=*INEC,'44,!30,Z!01;!F14$
Z!00:  NOP;!F14$
$
JPA,I INSTR CODE !()  $
       JPA,    ,!10;!F14$
$
JPA ADDR CODE !()  $
       2=?LDA,'42,!10;!F14$
       JPX,    ,'42;!F14$
$
JPA ADDR CODE !(!)  $
       2=?LDA,'42,!10;!F14$
!20MEM$
FIX OFFSET HILF$
       JPX,    ,'42;!F14$
$
FIX OFFSET !$
       2=*ADC,'42,!11;!F14$
$
JPA ADDR CODE !()+  $
       2=?LDA,'42,!10;!F14$
       2=*ADR,'42,'40;!F14$
       JPX,    ,'42;!F14$
$
JPA ADDR CODE !(!)+  $
       2=?LDA,'42,!10;!F14$
!20MEM$
FIX OFFSET HILF$
       2=*ADR,'42,'40;!F14$
       JPX,    ,'42;!F14$
$
JMP!  $
JPA!10  $
$
JEO!  $
       BNZ,@,    ,Z!00;!F14$
JPA!10  $
Z!00:  NOP;!F14$
$
JNE!  $
       BZ,@,    ,Z!00;!F14$
JPA!10  $
Z!00:  NOP;!F14$
$
JGE!  $
       BNP,@,    ,Z!00;!F14$
JPA!10  $
```

```
Z!00:  NOP;!F14$
$
JLE!  $
       BP,@,     ,Z!00;!F14$
JPA!10  $
Z!01:  NOP;!F14$
JEQ!10  $
$
JGT!  $
       BNP,@,     ,Z!00;!F14$
       BZ,@,     ,Z!00;!F14$
JPA!10  $
Z!00:  NOP;!F14$
$
JLT!  $
       BP,@,     ,Z!00;!F14$
JPA!10  $
Z!00:  NOP;!F14$
$
LOC   !()  $
!F14$
11111: NOP;$
$
SPACE ! CONST !$
$
SPACE ! ! !()  $
IF !11 NE  SKIP 1$
BIT!16$
!30:   !11*V;!F14$
$
SPACE CHAR(!) ! !()  $
!30:   !10*V;!F14$
$
SPACE INT ! !() A !$
IF !10 = ELEM SKIP 2$
!20:!26$
SKIP 1$
       !26$
!20    2*D,!34;!F14$
$
SPACE INT ! !() E !$
!30MEM$
SPACE INT !10 !20() A HILF$
$
SPACE REAL ! !() A !$
IF !10 = ELEM SKIP 2$
!20:!26$
SKIP 1$
       !26$
!20    G,!30;!F14$
$
SPACE CHAR(!) ! !() A !$
IF !20 = ELEM SKIP 2$
!30:!36$
SKIP 1$
       !36$
!30    !10*T,"!40";!F14$
SPACE CLO ! !() A !:!:!$
$
EVALDUR !30*3600+!40*60+!50$
CLOSPACE !10 !20 STACKDUR$
$
CLOSPACE ! ! !  $
IF !10 = ELEM SKIP 2$
!20:!26$
SKIP 1$
       !26$
!20    2*D,!31;!F14$
DURTEST EQU 1$
$
SPACE DUR ! !() A !$
DURATION !30$
```

```
CLOSPACE !10 !20 STACKDUR$
$
SPACE BIT(!) ! !() A *!*B1$
NULLSETZEN 16$
BREAK BITLIST !40 17-!10$
BREAK HEXA B1 B2 B3 B4*B5 B6 B7 B8*B9 B10 B11 B12*B13 B14 B15 B16$
SPACE HEXA !20 !30 B1 B2 B3 B4$
BITTEST EQU 1$
$
SPACE HEXA ! ! ! ! ! !$
IF !10 = ELEM SKIP 2$
!20:!26$
SKIP 1$
       !26$
!20.   2*B,'!31!41!51!61;!F14$
$
SPACE ! ! !(!)  $
!40MEM$
FIELDSPACE !10 !30 HILF$
$
FIELDSPACE ! ! !$
IF !10 = ELEM SKIP 2$
!20:!26$
SKIP 1$
       !26$
!20    !31*V;!F14$
$
SPACE ! ! !(!*!) A !$
DURTEST EQU 0$
BITTEST EQU 0$
!40!F7$
IF DURTEST EQUAL 0 SKIP 2$
CLOSPACE !20 !30 STACKDUR$
SKIP 5$
IF BITTEST EQUAL 0 SKIP 2$
SPACE HEXA !20 !30 B1 B2 B3 B4$
SKIP 2$
SPACE !10 !20 !30() A !60$
ELEM!26$
!F8$
$
SPACE ! ! !(!*!)+  $
FELD EQU !30$
FEL EQU !20$
$
SPACE ! ! (!)  $
MEMFELD0 !10 FEL FELD !30$
FELD EQU   $.
FEL EQU ELEM$
$
MEMFELD0 ! ! ! !$
SPACE !10 !21 !31(!40)  $
$
SPACE ! ! (!*!) A !$
MEMFELD1 !10 FEL FELD !30 !40 !50$
FELD EQU   $
FEL EQU ELEM$
$
MEMFELD1 ! ! ! ! ! !$
SPACE !10 !21 !31(!40*!50) A !60$
$
SPACE ! ! (!*!) E !*!  $
IF !61 NE  SKIP 1$
BIT!66$
!50*!61!56$
MEMFELD1 !10 FEL FELD !30 !40 !54$
FELD EQU   $
FEL EQU ELEM$
$
SPACE ADDR ! !(!*!)+  $
FELD EQU !20$
FEL EQU ADDR$
```

```
$
SPACE ADDR ! () A !$
MEMADDR FEL FELD !20$
FELD EQU   $
FEL EQU ELEM$
$
MEMADDR ! ! !$
IF !10 = ELEM SKIP 2$
!21:!26$
SKIP 1$
       !26$
!20    2*A,!30:!F14$
$
EFORM$
       2=*WTC,'00,'0A0D;!F14$
15.5!$WERG,'00,'02;!F14$
$
FFORM$
       2=*WTC,'00,'0A0D;!F14$
6!$WDRD,'00,'02;!F14$
$
FIR$
       2=*WTC,'00,'0A0D;!F14$
6!$WDRD,'00,'40;!F14$
$
FTEXT$
       2=*WTC,'00,'0A0D;!F14$
       LDC,'42,0;!F14$
TEXT CHARL1-1 CHAR1$
$
TEXT ! !$
CHARL1!F7$
Z!00:  WTA,'00,!20,'42;!F14$
       2=*ADC,'42;!F14$
       BNEC,'42,!14,Z!00;!F14$
$
FBIT$
       2=*WTC,'00,'0A0D;!F14$
       2=*WBR,'00,'03;!F14$
       2=*WBR,'00,'02;!F14$
$
FCLO$
FFORM$
       4=*WTC,'00,"SEC";!F14$
$
FDUR$
FCLO$
$
START   !()  $
ANF EQU !10$
0000 EQU 0$
0001 EQU 1$
0010 EQU 2$
0011 EQU 3$
0100 EQU 4$
0101 EQU 5$
0110 EQU 6$
0111 EQU 7$
1000 EQU 8$
1001 EQU 9$
1010 EQU A$
1011 EQU B$
1100 EQU C$
1101 EQU D$
1110 EQU E$
1111 EQU F$
INT EQU 2$
CLO EQU 2$
DUR EQU 2$
BIT EQU 2$
REAL EQU 4$
       0,'4000;!F14$
```

```
        CSL,@,!10;!F14$
!10:    SBCD,@,!10;!F14$
PRK:    4*C,'00007FFF';!F14$
ER1:    28*T,"*FEHLER: INT.-ZAHL ZU GROSS*";!F14$
ER2:    28*T,"**FEHLER: BITKETTEN ZU LANG*";!F14$
$
/*!*/  $
/!10;!F14$
$
LINE   !$
/   LINE   !10;!F14$
$
!$
DIE FOLGENDE ZEILE WURDE NICHT ERKANNT..!F14$
!10!F14$
$
LINK TASK MODUL !(!)  $
        NOP;!F14$
$
BEGIN TASK MODUL !(!)  $
        NOP;!F14$
$
LEVEL    !$
$
TERM TASK MODUL !(!)  $
        NOP;!F14$
$
END TASK MODUL !(!)  $
        NOP;!F14$
$
FINISH  !()  $
END ANF$
$
END !$
ENDE : HLT;!F14$
        JPA,    ,!11;!F14$
Z;!F14$
!FO$
$$
```

Anhang III: CIMIC/1 - Testprogramm (Matrizenmultiplikation)

```
START   MATM()  $
SPACE INT CONST IE04() E REAL$
SPACE INT CONST IK01() A 2$
SPACE INT CONST IK02() A 3$
SPACE INT CONST IK03() A 1$
SPACE REAL CONST K001() A 11.$
SPACE REAL CONST K00C() A 21.$
SPACE REAL CONST K011() A 12.$
SPACE REAL CONST K016() A 22.$
SPACE REAL CONST K021() A 13.$
SPACE REAL CONST K026() A 23.$
SPACE REAL CONST K031() A 1.$
SPACE REAL CONST K035() A 2.$
SPACE REAL CONST K039() A 3.$
SPACE INT CONST K043() A 2$
SPACE INT CONST K046() A 3$
SPACE INT CONST K049() A 1$
SPACE REAL CONST K052() A 0.$
LINE     3$
LINE     4$
BEGIN TASK MODUL N010()  $
LEVEL     1$
SPACE REAL LOCAL N001(6*REAL)  $
SPACE REAL LOCAL N002(3*REAL)  $
SPACE REAL LOCAL N003(2*REAL)  $
SPACE INT LOCAL N004()  $
SPACE INT LOCAL N005()  $
SPACE INT LOCAL N006()  $
SPACE INT LOCAL N007()  $
SPACE INT LOCAL N008()  $
SPACE INT LOCAL N009()  $
LINK TASK MODUL N010()  $
LOAD REAL CONST K001() A 11.$
STORE REAL LOCAL N001()  $
LOAD REAL CONST K00C() A 21.$
STORE REAL LOCAL N001(1*REAL)  $
LOAD REAL CONST K011() A 12.$
STORE REAL LOCAL N001(2*REAL)  $
LOAD REAL CONST K016() A 22.$
STORE REAL LOCAL N001(3*REAL)  $
LOAD REAL CONST K021() A 13.$
STORE REAL LOCAL N001(4*REAL)  $
LINE     5$
LOAD REAL CONST K026() A 23.$
STORE REAL LOCAL N001(5*REAL)  $
LOAD REAL CONST K031() A 1.$
STORE REAL LOCAL N002()  $
LOAD REAL CONST K035() A 2.$
STORE REAL LOCAL N002(1*REAL)  $
LINE     6$
LINE     7$
LOAD REAL CONST K039() A 3.$
STORE REAL LOCAL N002(2*REAL)  $
LOAD INT CONST K043() A 2$
STN INT LOCAL N007()  $
STNN INT LOCAL N008()  $
STOORE INT LOCAL N009()  $
LIINE     8$
```

```
LINE      9$
LINE      10$
LOAD INT CONST IK85() A 1$
LOC    L001()  $
STN INT LOCAL N005()  $
CMP INT LOCAL N007()  $
JGT,I INSTR CODE L002()  $
LINE      11$
LOAD INT CONST IK83() A 1$
LOC    L003()  $
STN INT LOCAL N006()  $
CMP INT LOCAL N009()  $
JGT,I INSTR CODE L004()  $
LINE      12$
LDX INT LOCAL N005()  $
LOAD INT LOCAL N006()  $
MPY INT CONST IK81() A 2$
ADDAX INT     $
SBX INT CONST IK81() A 2$
MPX INT CONST IE04() E REAL$
LOAD REAL CONST K052() A 0.$
STORE REAL LOCAL N003(-1*REAL)+  $
LINE      13$
LOAD INT CONST IK83() A 1$
LOC    L005()  $
STN INT LOCAL N004()  $
CMP INT LOCAL N008()  $
JGT,I INSTR CODE L006()  $
LINE      14$
LDX INT LOCAL N005()  $
LOAD INT LOCAL N004()  $
MPY INT CONST IK81() A 2$
ADDAX INT     $
SBX INT CONST IK81() A 2$
MPX INT CONST IE04() E REAL$
LOAD REAL LOCAL N001(-1*REAL)+  $
STORE REAL LOCAL R000()  $
LDX INT LOCAL N004()  $
LOAD INT LOCAL N006()  $
MPY INT CONST IK82() A 3$
ADDAX INT     $
SBX INT CONST IK82() A 3$
MPX INT CONST IE04() E REAL$
LOAD REAL LOCAL R000()  $
MPY REAL LOCAL N002(-1*REAL)+  $
STORE REAL LOCAL R000(1*REAL)  $
LDX INT LOCAL N005()  $
LOAD INT LOCAL N006()  $
MPY INT CONST IK81() A 2$
ADDAX INT     $
SBX INT CONST IK81() A 2$
MPX INT CONST IE04() E REAL$
LOAD REAL LOCAL N003(-1*REAL)+  $
ADD REAL LOCAL R000(1*REAL)  $
STORE REAL LOCAL R000(2*REAL)  $
```

```
LDX INT LOCAL N005()  $
LOAD INT LOCAL N006()  $
MPY INT CONST IK81() A 2$
ADDAX INT     $
SBX INT CONST IK81() A 2$
MPX INT CONST IE04() E REAL$
LOAD REAL LOCAL N000(2*REAL)  $
STORE REAL LOCAL N003(-1*REAL)+  $
LOAD INT LOCAL N004()  $
ADD INT CONST IK83() A 1$
JMP,I INSTR CODE L005()  $
LOC   L006()  $
LINE      15$
LOAD INT LOCAL N006()  $
ADD INT CONST IK83() A 1$
JMP,I INSTR CODE L003()  $
LOC   L004()  $
LINE      16$
LOAD INT LOCAL N005()  $
ADD INT CONST IK83() A 1$
JMP,I INSTR CODE L001()  $
LOC   L002()  $
LINE      17$
LINE      18$
LOAD REAL LOCAL N003()  $
EFORM$
LOAD REAL LOCAL N003(1*REAL)  $
EFORM$
LOAD REAL LOCAL N003(2*REAL)  $
EFORM$
TERM TASK MODUL N010()  $
SPACE REAL LOCAL N000(3*REAL)  $
END TASK MODUL N010()  $
FINISH   MATHU()  $
LINE      19$
```

Anhang IV: Testprogramm zur Matrizenmultiplikation in Assemblercode.

```
0000                    0    ,'4000 ;
0001                    CSL  ,0    ,MATM  ;
0002        MATM:       SBCD,0    ,MATM  ;
0003        PRK :    4* 0    ,'0000FFFF;
0004        ER1 :   28* T    ,"*FEHLER: INT.-ZAHL ZU GROSS*";
0005        ER2 :   28* T    ,"**FEHLER: DITKETTEN ZU LANG*";
0006        IE04:    2* D    ,      4;
0007        IK01:    2* D    ,      2;
0008        IK02:    2* D    ,      3;
0009        IK03:    2* D    ,      1;
0010        K001:       G    ,11.;
0011        K006:       G    ,21.;
0012        K011:       G    ,12.;
0013        K016:       G    ,22.;
0014        K021:       G    ,13.;
0015        K026:       G    ,23.;
0016        K031:       G    ,1.;
0017        K035:       G    ,2.;
0018        K039:       G    ,3.;
0019        K043:    2* D    ,      2;
0020        K046:    2* D    ,      3;
0021        K049:    2* D    ,      1;
0022        K052:       G    ,0.;
0023         /
               LINE     3;
0024         /
               LINE     4;
0025                    NOP ;
0026        N001:   24* V    ;
0027        N002:   12* V    ;
0028        N003:    8* V    ;
0029        N004:    2* V    ;
0030        N005:    2* V    ;
0031        N006:    2* V    ;
0032        N007:    2* V    ;
0033        N008:    2* V    ;
0034        N009:    2* V    ;
0035                    NOP ;
0036                    LDAG,0    ,K001  ;
0037                    STAG,0    ,N001  ;
0038                    LDAG,0    ,K006  ;
0039                    LDCD,'42  ,      4;
0040                    STAG,0    ,N001  ,'42 ;
0041                    LDAG,0    ,K011  ;
0042                    LDCD,'42  ,      8;
0043                    STAG,0    ,N001  ,'42 ;
0044                    LDAG,0    ,K016  ;
0045                    LDCD,'42  ,     12;
0046                    STAG,0    ,N001  ,'42 ;
0047                    LDAG,0    ,K021  ;
0048                    LDCD,'42  ,     16;
0049                    STAG,0    ,N001  ,'42 ;
0050         /
               LINE     5;
0051                    LDAG,0    ,K026  ;
0052                    LDCD,'42  ,     20;
0053                    STAG,0    ,N001  ,'42 ;
0054                    LDAG,0    ,K031  ;
0055                    STAG,0    ,N002  ;
0056                    LDAG,0    ,K035  ;
0057                    LDCD,'42  ,      4;
0058                    STAG,0    ,N002  ,'42 ;
0059         /
```

```
0060        /
              LINE      7;
0061                    LDAG,0    ,K039   ;
0062                    LDCD,'42  ,       8;
0063                    STAG,0    ,N002   ,'42 ;
0064                    LDCD,0    ,       2;
0065                    STAD,0    ,N006   ;
0066                    STAD,0    ,N005   ;
0067                    STAD,0    ,N004   ;
0068        /
              LINE      8;
0069        /
              LINE      9;
0070        /
              LINE      10;
0071                    LDCD,0    ,       1;
0072        L001:       NOP ;
0073                    STAD,0    ,N008   ;
0074                    SBAD,0    ,N004   ;
0075                  2<*BZ  ,0   ,       ,Z1  ;
0076                    LDR ,0    ,'03    ;
0077        Z1  :       NOP ;
0078                    BNP ,0    ,       ,Z2  ;
0079                    BZ  ,0    ,       ,Z2  ;
0080                    JPA ,     ,L002   ;
0081        Z2  :       NOP ;
0082        /
              LINE      11;
0083                    LDCD,0    ,       1;
0084        L003:       NOP ;
0085                    STAD,0    ,N009   ;
0086                    SBAD,0    ,N006   ;
0087                  2<*BZ  ,0   ,       ,Z3  ;
0088                    LDR ,0    ,'03    ;
0089        Z3  :       NOP ;
0090                    BNP ,0    ,       ,Z4  ;
0091                    BZ  ,0    ,       ,Z4  ;
0092                    JPA ,     ,L004   ;
0093        Z4  :       NOP ;
0094        /
              LINE      12;
0095                    LDAD,'40  ,N008   ;
0096                    LDAD,0    ,N009   ;
0097                    MPCD,0    ,       2;
0098                  4=*SBC ,0   ,PRK    ;
0099                  4=*STR ,0   ,'46    ;
0100                    BNP ,'05  ,       ,Z5  ;
0101                  2=*WTC ,'00 ,'0A0D  ;
0102                 28=*WTA ,'00 ,ER1    ;
0103                    JPA ,     ,ENDE   ;
0104        Z5  :     4=*LDR ,0   ,'46    ;
0105                    ORR ,'03  ,'05    ;
0106                  2=*ADR ,'46 ,'02    ;
0107                    SBCD,'40  ,       2;
0108                  4=*STR ,0   ,'42    ;
0109                  2=*LDR ,0   ,'40    ;
0110                    MPCD,0    ,       4;
0111                  4=*SBC ,0   ,PRK    ;
0112                  4=*STR ,0   ,'46    ;
0113                    BNP ,'05  ,       ,Z6  ;
0114                  2=*WTC ,'00 ,'0A0D  ;
0115                 28=*WTA ,'00 ,ER1    ;
0116                    JPA ,     ,ENDE   ;
0117        Z6  :     4=*LDR ,0   ,'46    ;
```

```
0118                ORR ,'03 ,'05   ;
0119              2=*STR ,0  ,'40   ;
0120              4=*LDR ,0  ,'42   ;
0121                LDAG,0   ,K052  ;
0122                LDCD,'42 ,    -4;
0123              2=*ADR ,'42 ,'40  ;
0124                STAG,0   ,N003  ,'42 ;
0125       /
             LINE     13;
0126                LDCD,0   ,     1;
0127       L005:    NOP ;
0128                STAD,0   ,N007  ;
0129                SBAD,0   ,N005  ;
0130              2<*DZ  ,0  ,      ,Z7   ;
0131                LDR ,0   ,'03   ;
0132       Z7   :   NOP ;
0133                BNP ,0   ,      ,Z8   ;
0134                DZ  ,0   ,      ,Z8   ;
0135                JPA ,    ,L006  ;
0136       Z8   :   NOP ;
0137        /
             LINE     14;
0138                LDAD,'40 ,N008  ;
0139                LDAD,0   ,N007  ;
0140                MPCD,0   ,     2;
0141              4=*SBC ,0  ,PRK   ;
0142              4=*STR ,0  ,'46   ;
0143                BNP ,'05 ,      ,Z9   ;
0144              2=*WTC ,'00 ,'0A0D ;
0145             28=*WTA ,'00 ,ER1  ;
0146                JPA ,    ,ENDE  ;
0147       Z9   :  4=*LDR ,0  ,'46   ;
0148                ORR ,'03 ,'05   ;
0149              2=*ADR ,'40 ,'02  ;
0150                SBCD,'40 ,     2;
0151              4=*STR ,0  ,'42   ;
0152              2=*LDR ,0  ,'40   ;
0153                MPCD,0   ,     4;
0154              4=*SBC ,0  ,PRK   ;
0155              4=*STR ,0  ,'46   ;
0156                BNP ,'05 ,      ,Z10  ;
0157              2=*WTC ,'00 ,'0A0D ;
0158             28=*WTA ,'00 ,ER1  ;
0159                JPA ,    ,ENDE  ;
0160       Z10  :  4=*LDR ,0  ,'46   ;
0161                ORR ,'03 ,'05   ;
0162              2=*STR ,0  ,'40   ;
0163              4=*LDR ,0  ,'42   ;
0164                LDCD,'42 ,    -4;
0165              2=*ADR ,'42 ,'40  ;
0166                LDAG,0   ,N001  ,'42 ;
0167                STAG,0   ,N000  ;
0168                LDAD,'40 ,N007  ;
0169                LDAD,0   ,N009  ;
0170                MPCD,0   ,     3;
0171              4=*SBC ,0  ,PRK   ;
0172              4=*STR ,0  ,'46   ;
0173                BNP ,'05 ,      ,Z11  ;
0174              2=*WTC ,'00 ,'0A0D ;
0175             28=*WTA ,'00 ,ER1  ;
0176                JPA ,    ,ENDE  ;
0177       Z11  :  4=*LDR ,0  ,'46   ;
0178                ORR ,'03 ,'05   ;
                  2=*ADR ,'40 ,'02  ;
```

```
0180                SBCD,'40 ,      3;
0181              4=*STR ,@  ,'42   ;
0182              2=*LDR ,@  ,'40   ;
0183                MPCD,@   ,      4;
0184              4=*SBC ,@  ,PRK   ;
0185              4=*STR ,@  ,'46   ;
0186                BNP ,'05 ,      ,Z12 ;
0187              2=*WTC ,'00 ,'0A0D ;
0188             28=*WTA ,'00 ,ER1  ;
0189                JPA ,    ,ENDE ;
0190      Z12 :   4=*LDR ,@  ,'46   ;
0191                ORR ,'03 ,'05   ;
0192              2=*STR ,@  ,'40   ;
0193              4=*LDR ,@  ,'42   ;
0194                LDAG,@   ,R000  ;
0195                LDCD,'42 ,     -4;
0196              2=*ADR ,'42 ,'40   ;
0197                MPAG,@   ,N002  ,'42 ;
0198                ORR ,'03 ,'05   ;
0199                LDCD,'42 ,      4;
0200                STAG,@   ,R000  ,'42 ;
0201                LDAD,'40 ,N008  ;
0202                LDAD,@   ,N009  ;
0203                MPCD,@   ,      2;
0204              4=*SBC ,@  ,PRK   ;
0205              4=*STR ,@  ,'46   ;
0206                BNP ,'05 ,      ,Z13 ;
0207              2=*WTC ,'00 ,'0A0D ;
0208             28=*WTA ,'00 ,ER1  ;
0209                JPA ,    ,ENDE ;
0210      Z13 :   4=*LDR ,@  ,'46   ;
0211                ORR ,'03 ,'05   ;
0212              2=*ADR ,'40 ,'02   ;
0213                SBCD,'40 ,      2;
0214              4=*STR ,@  ,'42   ;
0215              2=*LDR ,@  ,'40   ;
0216                MPCD,@   ,      4;
0217              4=*SBC ,@  ,PRK   ;
0218              4=*STR ,@  ,'46   ;
0219                BNP ,'05 ,      ,Z14 ;
0220              2=*WTC ,'00 ,'0A0D ;
0221             28=*WTA ,'00 ,ER1  ;
0222                JPA ,    ,ENDE ;
0223      Z14 :   4=*LDR ,@  ,'46   ;
0224                ORR ,'03 ,'05   ;
0225              2=*STR ,@  ,'40   ;
0226              4=*LDR ,@  ,'42   ;
0227                LDCD,'42 ,     -4;
0228              2=*ADR ,'42 ,'40   ;
0229                LDAG,@   ,N003  ,'42 ;
0230                LDCD,'42 ,      4;
0231                ADAG,@   ,R000  ,'42 ;
0232                LDCD,'42 ,      8;
0233                STAG,@   ,R000  ,'42 ;
0234                LDAD,'40 ,N006  ;
0235                LDAD,@   ,N009  ;
0236                MPCD,@   ,      2;
0237              4=*SBC ,@  ,PRK   ;
0238              4=*STR ,@  ,'46   ;
0239                BNP ,'05 ,      ,Z15 ;
0240              2=*WTC ,'00 ,'0A0D ;
0241             28=*WTA ,'00 ,ER1  ;
0242                JPA ,    ,ENDE ;
0243      Z15 :   4=*LDR ,@  ,'46   ;
```

```
0244                ORR ,'03 ,'05   ;
0245              2=*ADR ,'40 ,'02   ;
0246                SBCD,'40 ,     2;
0247              4=*STR ,0  ,'42   ;
0248              2=*LDR ,0  ,'40   ;
0249                MPCD,0   ,     4;
0250              4=*SUC ,0  ,PRK   ;
0251              4=*STR ,0  ,'40   ;
0252                 DNP ,'05 ,     ,Z16 ;
0253              2=*WTC ,'00 ,'0A0D ;
0254             28=*WTA ,'00 ,ERR  ;
0255                 JPA ,    ,ENDE  ;
0256      Z16 :    4=*LDR ,0  ,'40   ;
0257                 ORR ,'03 ,'05   ;
0258              2=*STR ,0  ,'40   ;
0259              4=*LDR ,0  ,'42   ;
0260                 LDCD,'42 ,     8;
0261                 LDAG,0   ,R000 ,'42 ;
0262                 LDCD,'42 ,    -4;
0263              2=*ADR ,'42 ,'40   ;
0264                 STAG,0   ,N003 ,'42 ;
0265                 LDAD,0   ,N007  ;
0266                 ADCD,0   ,     1;
0267                 JPA ,    ,L005  ;
0268      L006:      NOP ;
0269       /
             LINE     15;
0270                 LDAD,0   ,N009  ;
0271                 ADCD,0   ,     1;
0272                 JPA ,    ,L003  ;
0273      L004:      NOP ;
0274       /
             LINE     16;
0275                 LDAD,0   ,N008  ;
0276                 ADCD,0   ,     1;
0277                 JPA ,    ,L001  ;
0278      L002:      NOP ;
0279       /
             LINE     17;
0280       /
             LINE     18;
0281                 LDAG,0   ,N003  ;
0282              2=*WTC ,'00 ,'0A0D ;
0283           15.5 SWERG,'00 ,'02   ;
0284                 LDCD,'42 ,     4;
0285                 LDAG,0   ,N003  ,'42 ;
0286              2=*WTC ,'00 ,'0A0D ;
0287           15.5 SWERG,'00 ,'02   ;
0288                 LDCD,'42 ,     8;
0289                 LDAG,0   ,N003  ,'42 ;
0290              2=*WTC ,'00 ,'0A0D ;
0291           15.5 SWERG,'00 ,'02   ;
0292                 NOP ;
0293      R000:   12* V    ;
0294                 NOP ;
0295      ENDE:      HLT ;
0296                 JPA ,    ,MATM  ;
0297                 Z   ;
```