DESY DV-78/04
Mai 1978

Ein allgemeiner Cross-Assembler für unterschiedliche Rechnertypen

von W. Wimmer

Inhalt

Zusammenfassung

Um die Lücke bei der Entwicklung von Software für Mikroprozessoren bzw. für aus ihnen aufgebauten Minicomputern zu verkleinern, wurde ein allgemeiner Cross-Assembler entwickelt. Er ermöglicht, Syntax und Codegeneration für weite Klassen von Assemblersprachen und Rechnertypen zu definieren, um dann die definierte Sprache zu übersetzen.

Abstract

There is not much help for the development of software for microprocessors or minicomputers. To give a starting point a general cross-assembler was developed. This cross-assembler allowes to define syntax and generated machine code for a wide class of assembler languages and computers.

1. Einleitung

Die Entwicklung hochintegrierter elektronischer Bausteine, insbesondere von Mikroprozessoren und Halbleiterspeichern zu immer niedrigeren Preisen leitete eine neue Ära der Elektronik ein. Sogar Hobbybastler haben sich dieser Bausteine schon bemächtigt und werden in Zukunft einen nicht zu vernachlässigenden Markt dafür bilden.

In erster Linie jedoch werden die "Mikros" momentan von Großfirmen zur Herstellung "intelligenter" Geräte eingesetzt. Die Herstellung intelligenter Terminals ist erst durch Mikroprozessoren zu vernünftigen Preisen möglich geworden. Hierbei werden Geräte, die vorher im Prinzip schon da waren, mit zusätzlicher Intelligenz ausgerüstet. Dies hat zur Folge, daß die Bedienungsmöglichkeiten (Fernseher, Waschmaschine) benutzerfreundlicher gestaltet werden können und teilweise zusätzliche Fähigkeiten entstehen, die bei Terminalsystemen einen Zentralrechner entlasten können (Schlagwort: verteilte Intelligenz).

Der günstige Preis solcher Systeme ergibt sich aus dem ebenfalls günstigen Preis der Mikros sowie der Tatsache, daß die Kosten einer Programmierung nur einmal anfallen und sich deshalb auf viele Geräte verteilen.

Eine andere Einsatzmöglichkeit für Mikros, die jedoch erst zögernd benutzt wird, ist die Herstellung von Kleinserien mit benutzerindividuell zugeschnittener Programmierung. Die Hardwarepreise lassen den Bau von Kleinserien insbesondere für mittlere und kleine Firmen durchaus attraktiv erscheinen, jedoch die vergleichsweise noch hohen Softwarekosten wirken abschreckend.

Auch in wissenschaftlichen Forschungslabors wurden der Wert und die Möglichkeiten hochintegrierter Bausteine erkannt, die es ermöglichen, intelligente Systeme für Spezialaufgaben zu entwickeln. Hier spielen die Kosten der Programmierung in der Regel keine große Rolle, da Hardware und Software gemeinsam im Hinblick auf die zu lösende Aufgabe entwickelt werden und somit eine Trennung von Soft- und Hardwarekosten schlecht möglich ist.

Jedoch wird meist Wert auf eine schnelle Fertigstellung gelegt, insbesondere also auf eine schnelle Realisierung des vorgegebenen Softwarekonzeptes.

Bei Kleinserien und Spezialanfertigungen sind für die Programmierung nur Cross-Assembler und Cross-Compiler wirtschaftlich verwendbar, da die entwickelten Geräte in der Regel sowohl von den rechnerischen Fähigkeiten als auch von der Peripherie her nicht für Übersetzungsaufgaben eingerichtet sind. Auch können bei der Verwendung von Cross-Übersetzern schon vorhandene sprachenunabhängige Software-Erstellungssysteme und Editoren benutzt werden.

Diese Systeme, die meist auf dem Prinzip der strukturierten Programmierung und des Top-Down-Design beruhen, setzen eine Programmierprache und damit einen Cross-Übersetzer voraus, meist sogar eine höhere Programmiersprache.

Diese Lücke kann durch einen allgemeinen Cross-Assembler verringert werden, der es einem Benutzer erlaubt, sowohl eine Assemblersprache als auch den daraus bei der Übersetzung zu generierenden Code zu definieren. Im folgenden sollen Realisierungsmöglichkeiten hierfür untersucht und ein erfolgreich arbeitender Lösungsvorschlag näher dargestellt werden.

2. Erfordernisse

Um relativ einfach und schnell Programme bzw. Mikroprogramme für einen neuen oder neu zu bauenden Rechner zu schreiben, ist der Maschinencode denkbar ungeeignet. Selbst beim Erzeugen von Mikroprogrammen, einer relativ selten durchzuführenden Tätigkeit, hilft eine symbolische Schreibweise mit, Kodier- und Adressierfehler zu vermeiden. Die Benutzung von Symbolen statt Zahlenwerten, insbesondere von Adreßsymbolen, erleichtert außerdem das Ändern und Erweitern vorhandener Programme. Die Vorteile eines Assemblers auch für die Erzeugung von Mikroprogrammen liegen also auf der Hand. Allerdings kann es gerade bei der Mikroprogrammierung vorkommen, daß die Entwicklung eines Assemblers länger dauert als die des Mikroprogramms. In diesen Fällen hilft ein Werkzeug, welches es einem schnell und bequem erlaubt, eine Assemblersprache und den daraus resultierenden Maschinencode zu definieren.

Was sind die Anforderungen an solch ein Werkzeug?

1. Es soll aus einer symbolischen Assemblersprache Maschinencode generieren. Dies ist die übliche Aufgabe eines Assemblers. Da hier im wesentlichen an mit Mikroprozessoren aufgebaute Systeme gedacht ist, genügt es, absoluten Code zu erzeugen, dessen Adressen bei der Assemblierung festliegen bzw. festgelegt werden. Insbesondere wichtig ist die symbolische Adressierung.

2. Es soll Code für verschiedene Maschinentypen generiert werden. Es ist klar, daß kein Werkzeug so allgemein sein kann, daß es nicht doch Ausnahmen gibt, welche nicht mit ihm bearbeitet werden können. Aber es ist möglich, für eine große Klasse von Rechnern ein passendes Werkzeug zu liefern.
   Dieses muß also einstellbar sein auf z. B. verschiedene Längen der im Speicher zugreifbaren "adressierbaren Einheiten" (AE), verschiedene Befehlslängen (in AE), verschiedene Adressierungsschemata usw.

3. Eine gewünschte Sprache soll schnell und einfach definierbar sein. Insbesondere sollen eventuell schon existierende Assemblersprachen leicht übersetzbar gemacht werden können.
Im allgemeinen sieht eine Sprache für Mikroprogrammierung anders aus als eine für Assemblerprogrammierung, da es in einem Mikroprogrammbefehl meist viel mehr mögliche Operanden gibt, von denen nur einige ausgesucht werden. Dadurch können sich prinzipiell syntaktische Unterschiede ergeben.

4. Es sollten nach Möglichkeit allgemeine rechnerunabhängige Eigenschaften vorhanden sein, die in jedem Assembler wiederkehren. Dazu gehören z. B. :

   Wahl einer Zahlbasis für Eingabe und Listen, Eingabe von Zahlen verschiedener Zahlbasis, Umwandlung von Zeichenketten in ASCII-Code, bedingtes Assemblieren, Setzen des Programmzählers, arithmetische Ausdrücke anstelle von Operanden, Umschalten des Eingabestromes, usw.

   Ein zusätzlicher, jedoch oft wünschenswerter Komfort ist es, auf einer Host-Maschine Code für einen anderen, neu zu bauenden Rechner zu generieren, um diesen simulieren zu können. Der Simulator kann dann in der Hostsprache geschrieben sein, das simuliert ablaufende Programm in der Guestsprache.

3. Realisationsmöglichkeiten

Die Forderung nach freier Sprachdefinition legt die Verwendung eines Compiler-Compilers nahe. Dies ist ein Programm, welches als Eingabe die Syntaxdefinition einer Sprache erhält und einen Compiler für diese Sprache generiert. Diese Methode bietet zwar die Möglichkeit beliebiger Sprachdefinition, aber kann nur Compiler für einen festgelegten Rechnertyp liefern, scheidet damit also aus.

Eine andere Möglichkeit besteht darin, einen allgemeinenen Makroassembler zu benutzen. Funktionsweise und Einsatzmöglichkeiten von Makroprozessoren sind in Brown (1) sehr übersichtlich dargestellt. Als Ergebnis eines Übersetzungslaufes würde man den Maschinencode in Form von Zahlen erhalten, welche anschließend von einem speziell geschriebenen, aber relativ einfachen Programm in eine ladbare Form verwandelt werden. Diese Methode hat folgende Nachteile.

- Die Definition der Makros für einzelne Maschinenbefehle ist recht langwierig und fehleranfällig, so daß der Aufwand z.B. für die Mikroprogrammierung eines Rechners meist zu groß ist.
- Da Makros durch Zeichenersetzungen aufgelöst werden, ist ein Makroassembler in der Regel sehr rechenintensiv.

Aus diesem Grunde wurde schon 1966 von Ferguson (2) ein "Meta-Assembler" genannter verallgemeinerter Makroprozessor beschrieben, welcher nicht mehr Rechenzeit benötigt als ein durchschnittlicher normaler Assembler. Die Definition der Assemblerbefehle ist jedoch semantik-orientiert und üblichen Makroprozessormöglichkeiten nachempfunden und damit nur von Experten in aller Schönheit zu handhaben.

Eine weitere Möglichkeit ist die Verwendung eines Makrocompilers. Dies ist ein Assembler, welcher mit Hilfe einer speziellen Makromöglichkeit erlaubt, neue Befehle zu definieren. Das Spezielle der Makromöglichkeit liegt darin, daß das definierte Makro direkt in Maschinencode umgesetzt wird, ohne daß zwischendurch irgendwelche Zeichenersetzungen geschehen müssen. Damit übersetzt ein

Makro-Compiler bedeutend schneller als ein allgemeiner Makroassembler. Ein Makrocompiler ist auch den Erfordernissen eines Assemblers bedeutend besser angepaßt als ein Compiler-Compiler. Er besitzt jedoch ebenfalls einen Nachteil: es kann nur Code für Maschinen generiert werden, welche die gleiche Speicherstruktur wie der Rechner besitzen, für den der Makrocompiler ursprünglich geschrieben wurde.

Die im folgenden vorgeschlagene Lösung, die GEASM (General Assembler) genannt wurde, ist ein modular aufgebauter, weitgehend parametrisierter verallgemeinerter Makrocompiler. Er erfüllt die in Kapitel II aufgestellten Erfordernisse und ist, weil er modular aufgebaut und in der höheren Programmiersprache PL/I geschrieben ist, leicht und schnell auch ausgefallenen Bedürfnissen anzupassen.

Die Kodegeneration im GEASM geschieht folgendermaßen:
Es wird angenommen, daß jeder Befehl aus einer Bitkette bestimmter Länge besteht, welche aus einer ganzzahligen Zahl adressierbarer Einheiten zusammengesetzt ist.

Der "Operationskode" eines Befehls liefert einen Befehlsgrundwert, der durch "Schlüsselworte" weiter verändert werden kann. Operandenwerte werden als zusammenhängende Bitstücke in noch "freie" Stellen des Befehls eingesetzt.

Die Verknüpfung der Kodegeneration mit der Syntaxdefinition geschieht über Syntaxendelemente. Solche Syntaxendelemente heißen "Kodemodifikatoren". Weiter gibt es noch "Begrenzer" als Syntaxendelemente. Sie dienen allein der eindeutigen Syntaxanalyse. Die weiteren Details sind in den folgenden Kapiteln dargelegt.

Sind keine Programmerweiterungen zu machen, was der Regelfall sein dürfte, so liegt nach bisherigen Erfahrungen der Aufwand zum Implementieren einer Sprache je nach Sprachumfang und Erfahrung des Implementeurs zwischen einer Stunde bis zu 2 Tagen.

Bei notwendigen Programmänderungen sind es einige Tage bis eine Woche. Schon verfügbar sind Assemblersprachen für Nova, M 6800 und für die Simulation eines geplanten noch nicht fertiggestellten Rechners sowie einige spezielle Sprachen zur Mikroprogrammierung.

4. Der GEASM - eine Realisation

4.1. Sprachaufbau

4.1.1 Allgemeines

Ein allgemeiner Assembler muß zwei verschiedene Sprachen verarbeiten können: Einmal eine Sprache, mit deren Hilfe eine Assemblersprache und die Art, wie sie in Maschinencode umgesetzt werden soll, definiert wird, und zweitens die so definierte Sprache, um daraus den Maschinencode und Listen mit Fehlermeldungen u. ä. zu produzieren.

Einige allgemeine Eigenschaften der Definitions-(Meta-)Sprache und der definierten Assemblersprache können im GEASM über "Systemvariable" (siehe 6.1, 6.2) festgelegt werden. Für beide Sprachen gilt ein in 4.1.2.1 und 4.1.2.2 beschriebenes allgemeines Format. Die Definitionssprache wird im restlichen Teil von 4.1.2 beschrieben. Sie gestattet es, eine Assemblersprache innerhalb des in 4.1.2.2 gegebenen allgemeinen Formats zu definieren.

4.1.1.1 Schreibweise

Für die im folgenden beschriebenen Definition und Formate gelte:

> Große Buchstaben werden so geschrieben, wie sie dastehen.
> Kleine Buchstaben bzw. kleingeschriebene Worte stehen für allgemeine Begriffe und werden im Einzelfall sinngemäß ersetzt.
> In eckigen Klammern Geschriebenes [ ] ist optional, kann also im Einzelfall weggelassen werden.

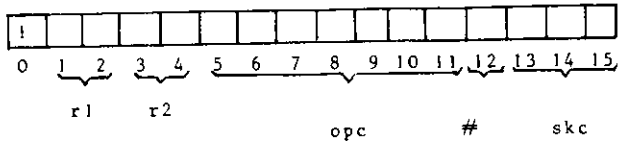
4.1.1.2 Beispiele

Um die folgenden Definitionen und ihre Benutzung zu erläutern, wird als Beispiel die Definition eines Register-Register-Befehls aus dem Nova-Assembler jedem Abschnitt folgen.

Dieser Befehl hat folgendes Format:

[label :] opc [#] r1, r2 [,skc] [; kommentar]

Dabei wird die Maschinenfunktion opc mit den Registern r1 und r2 durchgeführt und das Ergebnis in r2 geladen, wenn das No-Load-Zeichen# fehlt. Der Skipcode skc gibt an, bei welchen Werten des Ergebnisses der nächste Befehl übersprungen werden soll. Fehlt er, so wird der nächste Befehl immer ausgeführt. Der dazugehörige Maschinencode ist so aufgebaut:



4.1.2 Sprachbeschreibung

Die Abschnitte 4.1.2.1 und 4.1.2.2 gelten sowohl für Definitions- als auch für die definierte Sprache.

Die Abschnitte 4.1.2.3, 4.1.2.4 und 4.1.2.5 beschreiben die Syntax der Definitionssprache und weisen auf die Bedeutung der Definitionsanweisungen für die definierte Assembler-Sprache hin.

4.1.2.1 Zeichen

Die Eingabe von Definitions- und definierter Sprache erfolgt im selben Eingabestrom. Im Eingabestrom sind alle Zeichen zulässig, die eingegeben werden können. Die Zeichen werden in die folgenden Klassen unterteilt:

I Buchstaben
II Numerische Zeichen
III Alphanumerische Zeichen - Vereinigung von I und II
IV Operationszeichen für arithmetische Ausdrücke
V Begrenzer arithmetischer Ausdrücke
VI Begrenzer - Vereinigung von IV und V

Die Klassen I, II, IV und V sind disjunkt. Die Klassen IV, II und III können vom Benutzer über Systemvariable (6.1) definiert werden. Alle Zeichen, die nicht in IV oder III enthalten sind, fallen in Klasse V. Dadurch sind auch die Klassen I (als Differenz von III und II) und VI (als Vereinigung von IV und V) definiert.

Das Leerzeichen (Blank) hat eine Sonderrolle. Mehrere aufeinanderfolgende Leerzeichen entsprechen einem einzigen Leerzeichen. Ist dieses Leerzeichen rechts und links von alphanumerischen Zeichen umgeben, so ist es ein Begrenzer. Ansonsten wird es ignoriert. Deshalb können Leerzeichen zur besseren Lesbarkeit beliebig verwendet werden.

Beispiel:

Beim Nova-Assembler sind folgende Zeichen Buchstaben:

ABCDEFGHIJKLMNOPQRSTUVWXYZ.¤

Numerische Zeichen sind 0123456789
Beide würden in den GEASM mit folgenden Definitionen eingeführt:

```
SYSCHAR = ABCDEFGHIJKLMNOPQRSTUVWXYZ.¤0123456789 ; KLASSE 3
SYSZAHL = 0123456789       ; KLASSE 2
```

Operationszeichen sind die Zeichen +-/*!&"
Sie würden eingeführt als

```
SYSARITH = +-/* !&"+    ; KLASSE 4
```

Alle Zeichen, die nicht in einer der obigen Anweisungen auftreten, fallen automatisch in Klasse V (z.B. (),'#)

4.1.2.2 Definitionen

Ein Name ist eine nicht durch Begrenzer getrennte Kette alphanumerischer Zeichen, von denen mindestens ein Zeichen ein Buchstabe ist.

Eine Zahl ist eine nicht durch Begrenzer getrennte Kette numerischer Zeichen.

Ein arithmetischer Ausdruck besteht aus einer Aneinanderreihung von Namen, Zahlen und Operationszeichen. Dabei muß zwischen zwei Namen, Zahlen bzw. Name und Zahl mindestens ein Operationszeichen stehen. Ein arithmetischer Ausdruck wird rechts und links von Begrenzern arithmetischer Ausdrücke (Zeichen aus Klasse V) eingerahmt.

Eine Karte ist eine Zeichenkette aus 80 Zeichen. Ein Symbol ist ein Name, dem eine Bedeutung zugewiesen wird.

Eine Karte wird als in sich abgeschlossene Einheit betrachtet. Jede Karte hat folgendes allgemeine Format:

[label lbgr] [anweisung] [cbgr kommentar]

Dabei ist:

label: Ein Name, dem der entsprechende Wert des Programmzählers zugewiesen wird.

lbgr: Ein spezieller Begrenzer aus Klasse V, der den davorstehenden Namen als label kennzeichnet (Labelbegrenzer).

anweisung: Eine Anweisung an den Assembler.

cbgr: Ein spezieller Begrenzer aus Klasse V, der nachfolgende Zeichen als Kommentar kennzeichnet.

kommentar: Kommentar.

Es gibt die Möglichkeit, das Label aufgrund seiner Stellung zu erkennen (erstes Zeichen der Karte = Leerzeichen, wenn das Label fehlt). Dann kann lbgr das Leerzeichen sein.

Es gibt zwei verschiedene Formate für Anweisungen. Eine dient der Definition der Assemblersprache oder der Definition in ihr benutzter Symbole. Das 2. Format ist das allgemeine Format der Assemblersprache.

1. Wertzuweisung, Symboldefinition
   name zbgr definitionswerte

name: ein Name, dem eine Bedeutung innerhalb des Assemblers gegeben werden soll,

zbgr: ein spezieller Begrenzer aus Klasse V, der Zuweisungsbegrenzer,

definitionswerte: definiert die Bedeutung von Name in der zu definierenden Assemblersprache.

2. Befehl der Assemblersprache

   aufhänger [bgr operandenliste]

aufhänger: definiert, welche Assembler-Funktion durchgeführt und wie die Operandenliste abgearbeitet wird, falls eine vorhanden ist.

bgr: ein Begrenzer ungleich lbgr, cbgr, zbgr.

operandenliste: eine Operandenliste.

Beispiele:

Nach den Definitionen im vorigen Teil des Beispiels gilt:

Namen: HUGO , B102 , 1B5 , αS. , .EQ. , .
Zahlen: 0123 , 456 , 37000
Arithmetische Ausdrücke: HUGO + 1B5 * 12 + -DREI
753 * 5 / 1B - 15 +/*! 1777

Label- und Kommentarbegrenzer für den Nova-Assembler werden so definiert:

SYSLABBGR = :
SYSCOMBGR = ;

Dabei wird schon der Zuweisungsbegrenzer = benutzt. = ist der Default-Wert des Zuweisungsbegrenzers, der mit

SYSZUWEIS = >

z.B. auf > geändert werden kann. Eine nachfolgende Zahlendefinition müßte dann so aussehen:

SYSZAHL > 01234567 ; Werte für Oktalzahlen

Als Beispiel für eine Assembleranweisung gelte der Nova-Register-Register Befehl

opc [#] r1,r2,skc

Dabei entspricht opc dem Aufhänger, r1,r2,skc der Operandenliste und # oder das Leerzeichen dem Begrenzer bgr. Aber zu den Assembleranweisungen gehören nicht nur Maschinenbefehle, sondern auch Assemblerbefehle.

4.1.2.3 Syntax-Definition der Assemblersprache

In diesem und den folgenden Kapiteln 4.1.2.4 und 4.1.2.5 werden die möglichen Formate der Wertzuweisungen, insbesondere also der Definitionswerte nach 4.1.2.2 besprochen.

4.1.2.3.1 Syntax der Operandenlistendefinition

Die Operandenlistendefinition ist eine Wertzuweisung, über die die Syntax einer Operandenliste für Assemblerbefehle definiert wird. Die Syntaxdefinition geschieht in einer Backus-Naur-Form. Mögliche Alternativen sind durch | voneinander getrennt.

Format:

op1 zbgr op11 [bgr op11 [bgr op11 ...]] [|op11 [bgr op11...]] [| op11...]

Hierdurch wird dem opl die Form der rechts von zbgr stehenden Operandenliste zugewiesen. Die Striche | trennen verschiedene für diese Operandenliste zugelassene Möglichkeiten. Die opll sind entweder selbst Namen einer Operandenliste, die wie oben das opl definiert sind, oder aber Endelemente eel. Endelemente werden so definiert:

eel zbgr EL [,[fkt][([opl] [, [op2] [, op3]])]]

Über die Endelementdefinition wird den formalen opll eine Bedeutung für die Codegenereation gegeben. Die Bedeutung ist durch fkt (opl,op2,op3) festgelegt. fkt gibt die Funktion an, die der Assembler ausführen soll, wenn er eel entdeckt. opl,op2,op3 sind Parameter für fkt. In 4.1.4.1 wird auf die Verknüpfung der Endelementdefinition mit der Codegeneration eingegangen.

Folgendes muß beachtet werden, wenn eine Operandenliste rekursiv definiert wird, d.h. derselbe Name taucht rechts und links von zbgr auf:

- Ist dieser Name der erste Name nach zbgr, so wird er ignoriert und der darauffolgende Begrenzer bgr als linke Begrenzung der Operandenliste angesehen. Dies ist die einzige Möglichkeit, einen bestimmten Begrenzer als linke Begrenzung einer Operandenliste zu definieren.

- Ist dieser Name nicht an erster Stelle, so liegt eine echte rekursive Definition vor.

  Eine weitergehende indirekte Rekursion ist nicht möglich, da Syntaxelemente opll definiert sein müssen, bevor sie in einer Definition auf der rechten Seite verwendet werden dürfen.

  Spezielle Begrenzer wie zbgr, lbgr, cbgr und die Operationszeichen sind bei der Definition einer Operandenliste als bgr nur mit großer Vorsicht zu verwenden, da sonst unerwünschte Effekte auftreten können.

Beispiel:

Die Beispiel-Operandenliste könnte folgendermaßen definiert werden:

```
R1=EL,SH(13,1,2)  ; Bedeutung von SH und Parametern siehe 2.4
R2=EL,SH(11,1,2)  ;
SKC=EL,SH(0,1,3)

ORR=R1,R2 | R1,R2,SKC   ; opl=eel,eel | eel,eel,eel
```

Diese Operandenliste ist nur mit Endelementen definiert. Eine andere Definitionsmöglichkeit wäre z.B.:

```
ORRH=R1,R2    ; Hilfsoperandenliste (opl=eel,eel)
ORR=ORRH | ORRH,SKC  ;  opl=opll | opll,eel
```

4.1.2.4 Typendefinition

Über einen Typ werden einem Aufhänger eine Operandenliste sowie Schlüsselworte (Keywords) zugeordnet. Ein Schlüsselwort ist eine Zeichenkette (ohne Leerzeichen), welche an beliebiger Stelle einer Operandenliste stehen kann und dort in Bezug auf Begrenzung die Rolle eines Leerzeichens hat. Jedem Schlüsselwort ist ein Wert zugeordnet. Taucht ein Schlüsselwort in einer Operandenliste auf, so wird sein Wert auf den Wert des Aufhängers addiert.

4.1.2.4.1 Syntax der Typendefinition

typ zbgr (opl,beflngth [,KEYW zbgr keyw(keyval,..),keyw(keyval,...),.])

Dabei ist beflngth die Länge des erzeugten Maschinencodes in Bitstücken (näheres siehe 4.1.3).

keyw ist der Name des Schlüsselwortes und die dahinterstehenden keyval seine Werte in den ihrer Stellung entsprechenden Bitstücken. Die Zahl der Werte darf nicht größer als beflngth sein.

Beispiel:

In unserem Beispiel existiert nur ein Schlüsselwort, das No-Load-Zeichen #. Die Länge der Register-Register-Befehle bei der Nova ist ein Maschinenwort, das entspricht einer adressierbaren Einheit, die 16 Bit lang ist.
Das Setzen des No-Load-Bits im Maschinenwort geschieht, wenn die Oktalzahl 10 auf den Maschinencodewert addiert wird. Um im folgenden Oktalzahlen eingeben zu können, muß erst die Zahlbasis des GEASM vom Default-Wert 10 auf 8 gesetzt werden. Dies geschieht mit

```
SYSBASIS = 8    ; Zahlbasis ist 8 ab hier (siehe 6.1)
```

Die Typedefinition für Register-Register-Befehle sieht dann so aus:

```
TRR = (ORR,1,KEYW= # (10)) ; R-R-Typ
```

Dem Typ TRR sind zugeordnet die Operandenliste ORR, die Maschinencodelänge 1 adressierbare Einheit und das Schlüsselwort # mit dem Oktalwert 10.

4.1.2.5 Symboldefinitionen

Symboldefinitionen dienen dazu, Namen einen Zahlenwert, einen Typ und eine Funktion zuzuordnen. Typ und Funktion sind für ein Symbol nur wesentlich, wenn es als Aufhänger benutzt wird.

4.1.2.5.1 Syntax der Symboldefinition

```
symb zbgr arad [,[typ][,cfkt]]
```

Dabei ist

arad: eine Zahl oder ein arithmetischer Ausdruck, der den Zahlenwert des Symbols ausmacht.

typ: ein durch eine Typendefinition definierter Typ
Default: keine Operandenliste, keine Schlüsselworte, beflngth=1

cfkt: eine Funktion, die der Assembler ausführen soll, wenn er das Symbol als Aufhänger antrifft. Falls cfkt=OC ist, kann eine Aufhänger-Wertliste angegeben werden:

OC ($[w_1],[w_2],\ldots w_n$) mit $n \leq$ PSYSBEFL (siehe 4.1.4)
w_i ist der Wert des i-ten Bitstückes.

Default: EXPR , d.h. werte den mit diesem Symbol beginnenden arithmetischen Ausdruck aus.

Beispiele:

Einfache Beispiele für Symboldefinitionen sind

```
SZR=4    ; Code für SKC: skip if Register Zero
SNR=5    ; Code für SKC: skip if Register not Zero
```

Einige Register-Register-Befehle werden so definiert:

```
ADD=103000,TRR,OC  ; Addiere R1 auf R2
MOVL=101100,TRR,OC  ; Schiebe R1 eine Stelle nach links,
                    ; R2=Ergebnis
```

Dabei gibt die vorne stehende Oktalzahl den Wert des entsprechenden Maschinencodes an, wenn sämtliche möglichen Operanden und Schlüsselworte den Wert Null haben. TRR gibt den Typ an, so wie er im vorigen Beispiel definiert wurde. Damit ist den Symbolen ADD und MOVL auch die Operandenliste ORR zugeordnet.

OC bedeutet, daß der Assembler beim Auftreten der Symbole ADD und MOVL als Aufhänger die für Operationscodes gültige Funktion durchführen soll (siehe 4.1.4, 6)

4.1.2.6 Auswertung arithmetischer Ausdrücke

Arithmetische Ausdrücke werden ohne Rücksicht auf die Art der Operatoren von links nach rechts abgearbeitet. Folgen zwei Operatoren direkt aufeinander, so wird zwischen ihnen eine Null als Operand angenommen. Einzige Ausnahme hiervon ist ".
Folgende Operatoren sind zulässig: + - /* & | " $.
Die Zeichen für die Operatoren können über die Systemvariable SYSARITH geändert werden. Im Beispiel in 4.1.2.1 wurde das Zeichen | für "oder" in ! umgewandelt, wie es im Nova-Assembler üblich ist.

Die Zeichen beinhalten folgende Operationen:

+ : Addition
- : Subtraktion
/ : Division
* : Multiplikation
& : Logisches Und - wird mit jedem Bit der Operanden durchgeführt
¦ : Logisches Oder - wird mit jedem Bit der Operanden durchgeführt
" : gibt an, daß als Wert des Operanden der ASCII-Code des unmittelbar auf " folgenden Zeichens genommen wird. Alle weiteren alphanumerischen Zeichen und Leerzeichen bis zum nächsten Begrenzer werden ignoriert.
$: Der nachfolgende Name ist eine hexadezimale Zahl

4.1.3 Maschinenbeschreibung

Für die Umsetzung einer Assemblersprache in den Maschinencode sind zwei Dinge ausschlaggebend:

I die Länge der kleinsten von der Maschine adressierbaren Speichereinheit

II die Zuordnung der ermittelten Symbol- bzw. Operandenwerte zu einzelnen Bits im Operationscode

Die Länge der kleinsten adressierbaren Einheit (in Bits) wird je Sprachdefinition als konstant angesehen und kann über Systemparameter definiert werden. Dabei wird mit SYSEINHEIT die maximale Länge eines Bit-Abschnitts definiert, der beim Listen des Maschinencodes in einer Zeile stehen soll, und mit SYSWOCNT die Anzahl dieser Stücke je adressierbarer Einheit. Der maximale Wert für SYSEINHEIT ist 24, der für SYSWOCNT ist begrenzt durch die System-Makro-Variable PSYSBEFL.

Der Zusammenbau der Symbolwerte zu einem Maschinencode wird von den Symbolwerten, der Syntax einer Anweisung und den zugehörigen Funktionen fkt in der Endelementdefinition der Syntax gesteuert. Dabei kann ein Befehl mehrere adressierbare Einheiten lang sein. Die Begrenzung bildet wiederum PSYSBEFL, denn es muß gelten:

Befehlslänge (in adressierb. Einheiten) * SYSWOCNT $\leq$ PSYSBEFL

4.1.4 Verbindung von Sprache und Maschinencode

Die Umsetzung der Assemblersprache in den Maschinencode geschieht, wie in 4.1.2 und 4.1.3 angedeutet, über die Funktionen fkt und cfkt. fkt ist dem Syntax-Endelement zugeordnet, cfkt dagegen dem einzelnen Symbol. Stößt der Assembler bei der Analyse einer Karte auf einen Aufhänger, so wird dessen cfkt ausgeführt.

4.1.4.1 Die wesentlichen Funktionen

Für die Erstellung des Maschinencodes sind folgende eingebaute cfkt-Funktionen wesentlich:

I EXPR
Hat ein Aufhänger die cfkt EXPR, so wird die gesamte auf der Karte stehende Anweisung als ein arithmetischer Ausdruck aufgefaßt, dessen Wert ermittelt und in einer adressierbaren Einheit untergebracht. Der Programmzähler wird um 1 erhöht. Eine Zahl, ein Label oder ein mit Operationszeichen beginnender Aufhänger haben automatisch die cfkt EXPR.

II OC
Die cfkt OC macht den Aufhänger zum Operationscode. Die Karte wird auf Schlüsselworte untersucht und, falls vorhanden, deren Wert auf den Wert des Aufhängers addiert. Der so ermittelte Wert bildet die Basis für den Maschinencode. Dieser wird weiter aufgebaut durch Abarbeitung der Operandenliste und Ausführen der Endelement-Funktionen fkt mit den Werten der den Syntaxelementen entsprechenden Operanden. Der Programmzähler wird um die Befehlslänge beflngth erhöht.
Da die meisten Maschinencodes jeden Operandenwert in einer zusammenhängenden Bitkette unterbringen, existiert eine eingebaute Endelementfunktion SH, welche es ermöglicht, Maschinencodes aus Bitketten aufzubauen. Lage und Länge der Bitkette innerhalb des Maschinencodes werden in der Endelement-Definition über die Parameter op1, op2 und op3 festgelegt:

eel = EL [,[SH][([shift] [,[wortnr] [, oplngth]])]]

Dabei bedeutet:

shift: Anzahl der Bits, die der Operand nach links geschoben werden soll. Default: 0

wortnr: Nummer des Bitstücks innerhalb des Befehls relativ zum Befehlsanfang, in das der Operand eingesetzt werden soll. Default: 1 ,d.h. 1. Bitstück des Befehls. wortnr darf nicht größer als die dem Aufhänger zugeordnete Befehlslänge (beflngth) sein.

oplngth: Länge des Operanden in Bits, Default: Länge des Bitstücks minus shift.

SH definiert die Einsetzfunktion. Es kann bei der Endelementdefinition weggelassen werden, da es der Default-Wert für fkt ist. Ist shift + oplngth >SYSEINHEIT und wortnr $\geq$2, so wird der vordere Teil des Operanden in das vorhergehende Bitstück gepackt.

Für komplizierte Möglichkeiten des Code-Zusammenbaus sowie für variable Befehlslängen kann der Benutzer eigene fkt bzw. cfkt in PL/I schreiben, die entweder vorhandene Funktionen ersetzen oder als neue Funktionen in den GEASM integriert werden können. Näheres über existierende Funktionen steht in 6., über die Möglichkeiten eigene Funktionen zu schreiben in 4.3.

III. VOC

Für variabel lange Befehle kann die cfkt VOC benutzt werden. Ist die Länge dabei nicht von der Zahl bzw. Syntax der Operanden, sondern von ihrem Wert abhängig, so muß i. a. ein zusätzlicher Assemblerpass vorgesehen werden. Ansonsten arbeitet VOC wie OC.

Beispiel:

Das Beispiel der Endelementdefinitionen für den Register-Register-Befehl steht in 4.1.2.3.1. Es lautete

```
R1=EL,SH(13,1,2)
R2=EL,SH(11,1,2)
SKC=EL
```


Nehmen wir an, daß irgendwo im Programm der Befehl

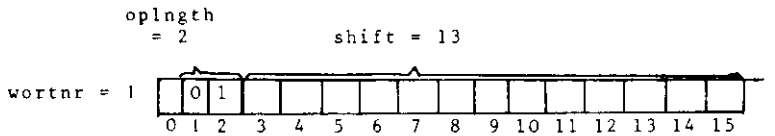
```
ADD  1,2,SNR          auftaucht.
```

Der GEASM entdeckt als Aufhänger das Symbol ADD. Über dessen Definition erfährt er, daß dies ein Operationscode mit dem Typ TRR ist. Da das Schlüsselwort # nicht im Befehl vorkommt, wird dem zu erstellenden Maschinencode der Wert des Symbols ADD zugewiesen und anschließend die Operandenliste ORR abgearbeitet. Wäre # im Befehl gewesen, so wäre auf den bisherigen Wert des Maschinencodes der Wert des Schlüsselwortes aufaddiert worden.

Bei der Analyse der Operandenliste wird folgende Zuordnung zwischen Syntax-Endelementen und auf der Karte angegebenen Werten gefunden:

```
R1  → 1
R2  → 2
SKC → SNR mit Wert 5
```

Bei der Definition von R1 und R2 wurde die Funktion SH angegeben, die aufgrund der Werte für shift, wortnr und oplngth die Werte 1 und 2 an die entsprechenden Bitstellen im Maschinenwort einsetzt. Beispiel R1:



Bei der Definition von SKC wurde nur EL angegeben.

Die Default-Werte entsprechen einer Definition

```
SKC = EL,SH(0,1,16)
```

da bei der Nova eine adressierbare Einheit = 1 Bitstück 16 Bit lang ist. Deshalb wird der Wert 5 in die rechten drei Bitstellen des Maschinenwortes eingesetzt. Will man sichergehen, daß der Skipcode nicht länger als drei Bitstellen wird, so wäre folgende Definition vorzuziehen:

```
SKC = EL, SH(0,1,3)
        oder
SKC = EL,(,,3)        ; Default-Werte
```

4.2 Programmaufbau

4.2.1 Allgemeines

Definitionssprache und definierte Assemblersprache werden im selben Eingabestrom in den GEASM eingegeben. Die Umsetzung der Eingabekarten in den Maschinencode erfolgt in zwei Schritten.

Im ersten Schritt wird die Definitionssprache verarbeitet, Befehlslängen festgestellt und den Labeln Werte zugewiesen. Außerdem können im ersten Schritt Definitionen von einer Datei geholt bzw. auf eine Datei geschrieben werden, Eingabekarten von einer Bibliothek in den Eingabestrom kopiert werden und Operationscode- und Syntax-Definitionen gelistet werden.

Im zweiten Schritt werden Maschinencodes berechnet und Assemblerfunktionen (außer den oben erwähnten) durchgeführt. Syntax- und Typdefinitionen werden nicht ausgewertet, Wertzuweisungen jedoch nochmals durchgeführt.

Aus diesem Ablauf ergibt sich, daß in Typ- und Syntaxdefinitionen auftretende Namen vorher definiert sein müssen. Bei Wertzuweisungen ist dies nicht nötig, sofern diese Werte keine Rolle bei der Feststellung der Länge der Maschinencodes spielen. Anderenfalls kann eine falsche Adresszuordnung zu Labeln die Folge sein.

4.2.2 Benutzte Tabellensysteme

Es werden drei verschiedene Tabellensysteme benutzt, je eins für

- Typdefinitionen
- Operandenlistensyntaxdefinitionen
- Symboldefinitionen

Jeder Tabelleneintrag in jeder Tabelle besteht aus einem Namen und einem Pointer auf die zum Namen gehörende Beschreibung. Die Einträge in jeder Tabelle sind alphabetisch nach dem Namen geordnet.

Die zu jedem Namen gehörende Beschreibung hat für jede Tabelle ein anderes Format. Alle diese Beschreibungen werden dynamisch in einer AREA angelegt, d.h. in einem speziell für diesen Zweck vorgesehenen Speicherbereich. Dies ermöglicht es, einmal festgelegte Definitionen zu retten bzw. wieder zu laden.

Die Formate der verschiedenen Beschreibungstypen sind im Folgenden in PL/I angegeben.

4.2.2.1 Syntax-Definition

```
DCL
1 GREL BASED(PGR),
  2 PSO OFFSET(TAREA),    /*Zeiger auf Sohn*/
  2 PDA OFFSET(TAREA),    /*Zeiger auf Fortsetzung*/
  2 PBR OFFSET(TAREA),    /*Zeiger auf Alternative*/
  2 ENDFKTN CHAR(4),      /*Endelement-Funktion*/
  2 WNAME,                /*Parameter für fkt*/
    3 SH    BIN FIXED,
    3 WORTNR BIN FIXED,
    3 OPLNGTH BIN FIXED,
  2 BEGR CHAR(1);         /*Begrenzung des Elements*/
```

Für ein Endelement ist PSO = Null, d.h. er zeigt das Ende einer Liste an. In diesem Fall wird die Endelementfunktion ausgeführt, die zu ENDFKTN gehört. Dabei kann WNAME als Parameter für die Funktion benutzt werden.

Für ein Nicht-Endelement zeigt PSO auf ein weiteres Syntaxelement. PDA gibt die Fortsetzung einer Operandenliste an, PBR eine alternative Operandenliste.

BEGR ist für jedes Syntax-Element gültig und gibt die rechte Begrenzung der zugeordneten Operanden an. Ist diese auf einer Karte ungleich BEGR, so wird die alternative Definition genommen oder, falls diese nicht vorhanden ist, ein Fehler angezeigt.

4.2.2.2 Typdefinition

```
DCL
1 TYPEL BASED(PTYP),
  2 PTSYNTEL OFFSET(TAREA),  /*Zeiger auf Operandenlistendef.*/
  2 TYPBEFLNGTH BIN FIXED,   /*Befehlslänge (in Bitstücken)*/
  2 PKEYW OFFSET(TAREA);     /*zeigt auf Schlüsselwortliste*/
```

4.2.2.3 Symboldefinition

```
DCL
1 CODEL BASED(PCOD),
  2 CODWERT BIN FIXED(31),   /*Symbolwert */
  2 PCODTYP OFFSET(TAREA),   /*zugehöriger Typ*/
  2 PXREF  POINTER,          /*Zeiger auf Cross Reference Table */
  2 CFKTN  CHAR(4);          /*Aufhängerfunktion*/
```

4.2.2.4 Zusammenhang der Tabellen

Figur 1 gibt den logischen Zusammenhang der Tabellen an. Dabei ist zu beachten, daß von jedem definierten Syntaxelement bei der Benutzung eine Kopie gemacht wird, um eine Mehrfachbenutzung schon definierter Syntaxelemente möglich zu machen.

Dies ist nötig, da sich bei jeder Wiederbenutzung PDA, PBR und BEGR ändern können. Da Assemblersprachen im allgemeinen eine verhältnismäßig einfache Grammatik haben, ist der zusätzliche Speicherbedarf von 10 Bytes je Verwendung in Kauf genommen worden, um den Listenaufbau nicht durch verschiedene Kontrollblocktypen zu kompliziert zu gestalten.

4.2.3 Beispiel der Tabellenbenutzung

Als Beispiel sei der Nova-ADD-Befehl genommen. Die Definitionen dafür sahen so aus:

```
R1=EL,SH(13,1,2)
R2=EL,SH(11,1,2)
SKC=EL,SH(0,1,3)

ORR=R1,R2 | R1,R2,SKC
SYSBASIS=8    ; ab jetzt Oktalwerte
TRR=(ORR,1,KEYW=# ,10)

SZR=4
ADD=103000,TRR,OC
```

Das daraus sich ergebende Tabellensystem zeigt Figur 1.
Die Abarbeitung eines ADD-Befehls, z.B.

```
ADD   1,2,SZR
```

geschieht so:

Der GEASM entdeckt den Aufhänger ADD und erfährt über die Symboltabelle, daß er die Aufhängerfunktion OC ausführen soll. Diese Funktion errechnet den Maschinencode für Befehle fester Länge. Über das Symbolelement erfährt der GEASM den Codewert für ADD und die Adresse des Typs. Im Typ stehen die für ADD möglichen Schlüsselworte. Da keines auf der Karte vorhanden ist, wird gleich die Operandenliste abgearbeitet. Über den Sohn-Pointer PSO des Syntaxelements ORR wird das Syntaxelement R1 gefunden. Dessen Sohn-Pointer ist NULL, es ist also ein Endelement. Nun wird auf der Karte der erste Operand gesucht und dessen Begrenzung mit der Soll-Begrenzung von R1 (,) verglichen. Beides stimmt überein, und die Endelementfunktion SH wird mit dem Operandenwert 1 durchgeführt, d.h. die 1 an entsprechender Stelle in den Maschinencode eingesetzt. Als nächstes wird über den Data-Pointer PDA von R1 das Endelement R2 und der zugehörige Operand auf der Karte gefunden. Der Sollbegrenzer von R2 ist aber N, d.h. kein Begrenzer mehr oder ein Kommentarbegrenzer.

Dies stimmt nicht mit dem Komma auf der Karte überein, also wird bei R2 eine Alternative gesucht. Da PBR bei R2 Null ist, wird wieder zu R1 zurückgegangen und dort eine Alternative gesucht und auch gefunden. Diese wird nun wie eben beschrieben abgearbeitet und kommt diesmal zu einem richtigen Ende, da nach dem Skipelement SKC kein Begrenzer mehr auf der Karte folgt. Wäre dies nicht der Fall, so läge entweder ein Syntaxfehler auf der Karte oder eine falsche Syntaxdefinition vor.

Wie man sieht, ist die Eindeutigkeit einer Operandenlisten-definition nur abhängig von der Länge der Operandenliste und der Art der Begrenzer. Soll ein Symbol zwei verschiedene Operandenlisten gleicher Länge haben, so muß mindestens ein Begrenzer bei beiden Alternativen verschieden sein, sonst wird nur die erste oder gar keine genommen.

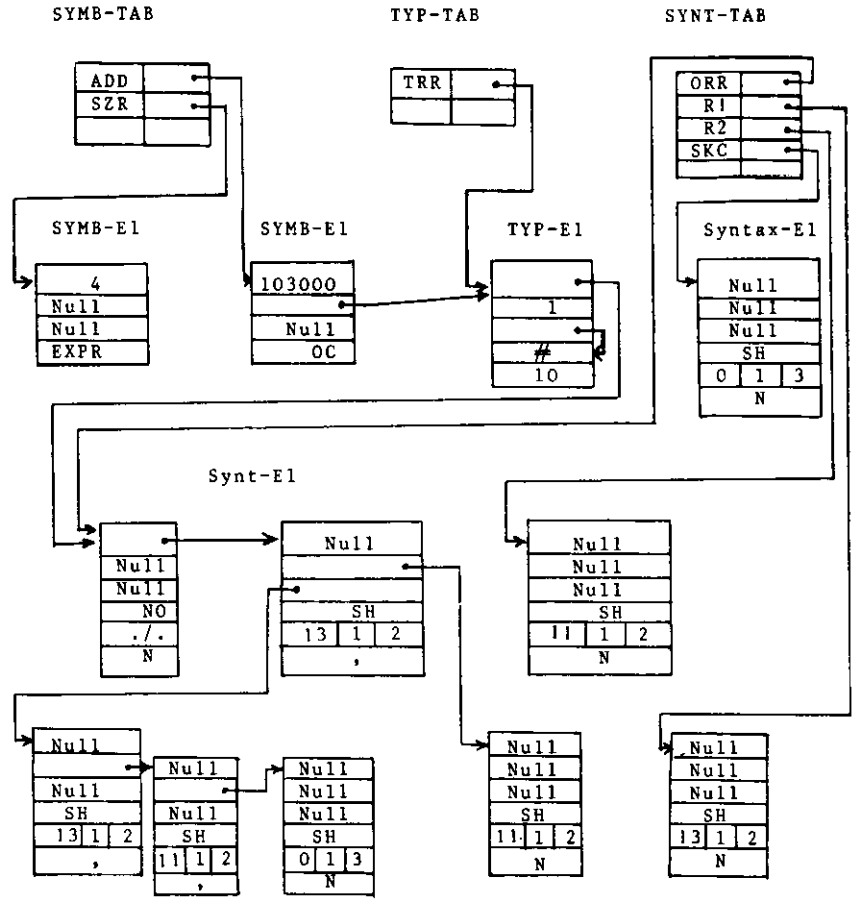


Fig. 1 Beispiel-Tabellensystem: alle Zahlen oktal

4.3 Schreiben von Funktionen durch den Benutzer

Benutzerfunktionen sollten in PL/I geschrieben sein. Es können über Makros und spezielle Deklarationen vom GEASM Werte zur Verfügung gestellt werden, welche die Benutzerfunktion benutzen und manipulieren kann.

4.3.1 Vom Assembler erhältliche Werte

Über die Makros SYSASM und TABDEF können Systemvariable und Tabellen mit Syntax-, Typ- und Symboldefinitionen zur Verfügung gestellt werden. Weiter kann mit

```
DCL
1 INPASS2 EXT,
  2 CARD2 CHAR(80),              /*Eingabekarte*/
  2 ERROR BIN FIXED,             /*Fehlernummer*/
  2 CODLNGTH BIN FIXED,          /*Länge des Codes*/
  2 LOCATION BIN FIXED;          /*Adresse des Codes*/
```

die Eingabekarte sowie Fehlernummer und Code-Länge erhalten bzw. gesetzt werden.

Der Befehlscode kann nach einem % INCLUDE SYSASM; mit
DCL BEFCODE(PSYSBEFL) BIN FIXED(31) EXT; erhalten werden.
Die Bedeutung der Systemvariablen und Tabellendefinitionen ist dem Anhang zu entnehmen.

4.3.2 Endelement-Funktionen

Endelement-Funktionen sind die Funktionen, die einem Syntax-Endelement angehören. Diese Funktionen erhalten folgende Parameter:

1.) ein Pointer auf die Beschreibung des Syntax-Endelementes. Insbesondere können die drei Halbworte op1, op2, op3 erhalten werden, die in der Endelement-Definition angegeben werden können.

2.) der bisher erhaltene berechnete Wert (i.a. Maschinencode)

3.) der Wert des Operanden auf der Eingabekarte, der verarbeitet werden soll.

Darüberhinaus stehen die in 4.3.1 erwähnten Werte zur Verfügung. Soll eine Funktion neu hinzugefügt werden, so muß eine entsprechende Änderung im Modul ENDFKT durchgeführt werden, wobei der Funktion ein maximal 4 Zeichen langer Name gegeben werden muß, unter dem sie später einem Endelement zugeordnet werden kann. Dieser Name kann verschieden vom Namen der Subroutine sein, die die Funktion durchführt.

4.3.3 Aufhänger-Funktionen

Eine Aufhänger-Funktion ist jedem Symbol zugeordnet und wird durchgeführt, wenn dieses Symbol als Aufhänger angetroffen wird.
Einer Aufhänger-Funktion werden folgende Parameter übergeben:

1.) Ein Pointer auf die dem Symbol zugehörende Beschreibung - darüber sind insbesondere Wert und Typ des Symbols zu erfahren.

2.) Die Karte, die gerade bearbeitet wird

3.) Ein Index auf der Karte, der auf den zuletzt nach diesem Symbol gefundenen Begrenzer zeigt.

Wird direkt nach dem Namen der Benutzer-Aufhängerfunktion das Makro CODFKT gerufen, so werden die Deklarationen für die Parameter sowie die Werte, die sonst über das Makro SYSASM zu erhalten sind, zur Verfügung gestellt. Außerdem können die übrigen in 4.3.1 erwähnten Größen verwendet werden.

Wird eine neue Aufhänger-Funktion den bestehenden hinzugefügt, so muß eine entsprechende Änderung im Modul CFKT durchgeführt werden. Dabei wird der Funktion ein maximal vier Zeichen langer Name gegeben, unter dem sie später einem Symbol zugeordnet

werden kann. Dieser Name kann verschieden von dem der Subroutine sein, welche die Funktion durchführt.

4.3.4 Setzen von Fehlercodes

Tritt bei der Durchführung einer Benutzerfunktion ein Fehler auf, so kann durch die Subroutine FEHLER ein Fehlercode gesetzt werden, der mit der Karte, auf der der Fehler auftrat, gelistet wird. Ein Text, der den Fehler beschreibt, kann in einer Datei mit Fehlermeldungen hinterlegt werden und erscheint unter den Fehlermeldungen in der Ausgabeliste. Der Fehlercode besteht aus einer Zahl zwischen 1 und 90. Diese Zahl entspricht der Nummer der Fehlermeldung in der Fehlermeldungs-Datei. Deshalb müssen diese Fehlernummern sequentiell ansteigend von den bereits existierenden an vergeben werden. Eine Liste der schon existierenden Fehlermeldungen befindet sich im Anhang (6.6).

4.3.5 Loadmodul-Ausgabe

Die Loadmodul-Ausgabe erfolgt durch die Subroutine PUNCH. Sie ist momentan für absolute Nova-Loadmoduln ausgelegt, insbesondere zum Laden der Novas durch das IPS-System am DESY-Rechenzentrum. Die Subroutine PUNCH erstellt den Loadmodul aus einem Loadblock, der folgendermaßen aussieht.

```
DCL
1 LOADBLK EXT,
  2 MINWC  BIN FIXED,       /*DATA WORD COUNT*/
  2 LOC    BIN FIXED,       /*START ADDRESS OF THIS BLOCK*/
  2 CHECKSUM BIN FIXED,
  2 DATA(16) BIN FIXED;
```

Es ist zu beachten, daß für SYSEINHEIT > 16 die adressierbaren Einheiten verstümmelt in LOADBLK enthalten sind. In diesem Falle müßte LOADBLK geändert werden, ebenso wie die Routinen LOADM, ASMBLK, ASMLOC, ASMEND. Jedes DATA-Wort enthält eine adressierbare Einheit. Diese folgen ab der Adresse LOC sequentiell aufeinander. Ein neuer Block wird geschrieben, wenn alle

16 DATA-Worte voll sind oder eine Assembleranweisung durchgeführt wird, die eine der Routinen ASMBLK, ASMLOC bzw. ASMEND ruft und so die sequentielle Erhöhung des Programmzählers unterbricht.

Die CHECKSUM muß, falls benötigt, durch die Subroutine PUNCH berechnet werden. Nach Ausschreiben des Loadmoduls muß PUNCH den DATA-Wort-Zähler wieder auf Null setzen.

5. Zusammenfassung

Der GEASM ist ein Werkzeug, welches es ermöglicht, Assemblersprachen und den zu generierenden Code für unterschiedliche Rechner zu definieren. Damit stehen dem Benutzer die folgenden Einsatzmöglichkeiten zur Verfügung. Welche davon im Einzelfall praktikabel ist und welche nicht, muß von Fall zu Fall entschieden werden.

1. Eine Sprache - ein Zielrechnertyp

Dies ist der Normalfall, wo für einen Rechner ein Cross-Assembler gewünscht wird.

2. Eine Sprache - viele Zielrechnertypen

Der GEASM bietet wegen seiner Makrofähigkeiten die Möglichkeit, eine Art "höhere Assemblersprache" zu definieren, die auf unterschiedlichen Rechnertypen benutzt werden kann.
Programme werden damit auf diesen Rechnern portabel und ein Benutzer braucht nur eine Assemblersprache zu lernen. Das Verfahren ist in der Regel nur bei strukturell und vom Operationsumfang her ähnlichen Rechnertypen praktikabel. In allen anderen Fällen bleiben Spezialfähigkeiten einiger Rechner ungenutzt oder der generierte Maschinencode ist nicht effektiv genug. Beides sind jedoch Hauptanliegen des Assemblerprogrammierers.

Spielen Speicher- und Laufzeitoptimierungen keine so große Rolle, empfiehlt sich meist die Verwendung einer rechnerunabhängigen höheren Programmiersprache. Trotzdem kann dieser Ansatz sinnvoll sein, wenn man versucht, allen gemeinsam vorhandenen Befehlen gleichen Operationscode und gleiche Syntax zu geben. Dies erleichtert die Lernarbeit und verringert die Fehleranfälligkeiten beim Wechsel von der Programmierung eines Rechners zum anderen.

3. Viele Sprachen - ein Zielrechnertyp

Diese Einsatzmöglichkeit kann für die Simulation von Rechnern nützlich sein. Zu simulierendes Programm und der Simulator können gemeinsam übersetzt und in einem Stück geladen werden. Meist wird allerdings nur die Möglichkeit genützt, das zu simulierende Programm auf den Host-Rechner ladbar zu machen. Eine andere Anwendung ist die "Erfindung" einer neuen Kommando-Sprache z. B. für Display-Controller oder andere spezielle Hard- oder Software-Interpreter. So kann die Lesbarkeit von Assemblerprogrammen bedeutend verbessert werden.
Über die eigentliche Forderung 1) hinaus, die der Ausgangspunkt für die Entwicklung eines allgemeinen Assemblers war (siehe Kapitel 2), gibt es damit noch weitere lohnende Einsatzgebiete.

6. Anhang

6.1 System-Assembler-Variable

Name	Default-Wert	Beschreibung
SYSLABBGR	:	Label-Begrenzer (lbgr - 4.1.1.2)
SYSCOMBGR	;	Kommentar-Begrenzer (cbgr - 4.1.1.2)
SYSZUWEIS	=	Zuweisungsbegrenzer (zbgr - 4.1.1.2)
SYSPCHAR	.	Zeichen für gegenwärtiger Wert des Programmzählers
SYSPC		Programmzähler
SYSEINHEIT	16	Länge des je Zeile zu listenden Bitstücks des Maschinencodes (≤ 24)
SYSBASIS	10	Zahlbasis für Ein- und Ausgabe
SYSLASCHAR	72	letzte gültige Stelle auf der Karte (≤ 80)
SYSCHAR	'ABCDEFGHIJKLMN OPQRSTUVWXYZ a# $1234569890.'	alphanumerische Zeichen (4.1.1.1)
SYSWOCNT	1	Zahl der Bitstücke (SYSEINHEIT) je adressierbarer Einheit
SYSZAHL	'0123456789'	numerische Zeichen
SYSARITH	'+-/*&|"+'	arithmetische Zeichen - String muß 8 Zeichen lang ohne Leerzeichen sein - deshalb Wiederholung des +
SYSXREF	'0'B	= 1 : Cross Reference erwünscht
SYSLIST	'1'B	Liste den Eingabestrom.
SYSLABPOS	'0'B	= 1 : Label beginnt in Spalte 1 der Karte, kein Labelbegrenzer nötig.
SYSHEXLIST	'0'B	= 1 : Liste Adresse und Befehlscode als hexadezimale Zahlen. = 0 : Liste als oktale Zahlen.
SYSVERGL	"	Vergleichsoperatoren für bedingtes Assemblieren. Reihenfolge: =, ¬=, <, <=,>=,>

Die System-Assembler-Variablen können durch entsprechende Anweisungen bei jedem Lauf des GEASM über dessen Eingabekarten geändert werden.

6.2 System-Makro-Variable

Name	Default	Beschreibung
SYSNAML	12	max. Länge der Namen (in Zeichen)
PSYSSYNTM	100	maximale Zahl von Syntaxelementen
PSYSTYPM	100	max. Zahl von Typdefinitionen
PSYSBEFL	36	max. Zahl durch SYSEINHEIT def. Bitstücke je Befehl
PSYSCODM	2000	max. Zahl von Symbolen
PSYSAREAL	32700	Speicherplatz (Bytes) für Definitionen

System-Makro-Variable definieren GEASM-Eigenschaften während dessen Kompilierung. Werden sie neu definiert, so muß anschließend der GEASM neu übersetzt und gebunden werden.

6.3 Verfügbare Funktionen

In Klammern geschriebene Subroutinen-Namen existieren nicht als Subroutine, sondern sind Teil der Subroutinen CFKT bzw. ENDFKT. Spezielle Funktionen für den Nova-Assembler sind durch (NOVA) gekennzeichnet.

6.3.1 Aufhänger-Funktionen

Name	Code	Beschreibung
ASMBLK	ABLK	Erhöht den Programmzähler um den Wert des einen Operanden
ASMEND	AEND	Schreibt den Endblock für den Loadmodul (Nova)
ASMLOC	ALOC	Setzt den Programmzähler auf den Wert des Operanden
ASMTXT	ATXT	Nimmt das 1. Zeichen ≠ Leerzeichen als Begrenzer der nachfolgenden Zeichenkette und wandelt sie in ASCII-Code um. In Abhängigkeit vom letzten vorher gegebenen ASMTXTM werden je 2 Zeichen in ein 16-Bit-Wort gepackt. Am Schluß der Kette folgt mind. ein 0-Byte (NOVA).
ASMTXTM	TXTM	Operand = 1: Text wird von links nach rechts in die Worte gepackt. Operand ≠ 1: Text wird von rechts nach links in die Worte gepackt.
BEGSIM	BSIM	Benutzung eines 2. Programmzählers für Simulationen sowie anderer Länge addr. Einheiten. Guestlänge=Hostlänge*Operand; neg. Operand: / (Operand)
COPY	COPY	Operanden: MEMBER, DSN COPY holt das MEMBER von der Bibliothek DSN und fügt es in die Eingabe ein. Wird DSN nicht spezifiziert, so wird der vorherige Wert von DSN verwendet. Fehlt MEMBER, so wird eine sequentielle Datei erwartet.
ENDSIM	ESIM	Zurückschalten auf alten Programmzähler bei Ende des zu simulierenden Programms
EXPR	EXPR	Auswertung der Karte als Datenwort. Sie darf außer Label und Kommentar nur einen arithmetischen Ausdruck enthalten (siehe 4.1.3)
GETDF	GETD	Hole Definitionen über die DD-Karte, deren DD-Name als Operand angegeben ist. (Default: SYSGETD)
LISTG	GR	Liste Grammatik -Definitionen
LISTO	LOPC	Liste alle Symbole mit der Funktion OPCODE

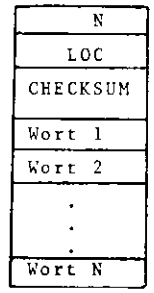
(NOFKT)	NO	Tue gar nichts (evtl. Operanden sind Kommentar)
OPCODE	OC	Werte die nachfolgende Operandenliste entsprechend der Typ-Definition aus und berechne den Maschinencode (siehe 4.1.3)
SAVE	SAVE	Rette alle bisherigen Definitionen über die DD-Karte, deren DD-Name als Operand angegeben ist (Default: SYSSAVE).
OPCODE	VOC	Wie OC, nur für Befehle variabler Länge-bei PASS1 wird die aktuelle Länge bestimmt.
PASSO	PASO	Extra Pass, um Symboltabelle aufzubauen. Wird benötigt vor Befehlen variabler Länge, bei denen die Länge von den Operandenwerten abhängt.
SETRETC	SRTC	Setze GEASM-Returncode
SCHWERR	SFSW	Setze Fehlerschwere - Fehlerschwere >4 hat Returncode 16 bei Eintritt des Fehlers zur Folge. Operanden: Fehlernr, Schwere, ...
AIF	AIF	Bedingtes Übersetzen Syntax: Arad Sysvergl Arad Asm-Befehl Führt Asm-Befehl aus, wenn Vergleich=True ist. Asm-Befehl darf keinen Code generieren.
AGO	AGO	Operand: Labelname Überliest alle nachfolgenden Karten bis zum Label-Befehl mit diesem Namen oder bis EOF.
(ALABEL)	ALAB	Kennzeichnet einen Labelnamen, zu dem mit dem AGO-Befehl gesprungen werden kann.

6.3.2 Endfunktionen

Name	Code	Beschreibung
ADRESS	ADDR	Setzt, falls der entsprechende Operand zwischen 0 und 255 liegt, die Adresse absolut in den Maschinencode ein. Anderenfalls wird eine Adressierung relativ zum Programmzähler vorgenommen (Nova).
(BEGR)	BEGR	Der Operand dient als Begrenzer und muß genauso geschrieben sein, wie in der Syntax definiert. Es gibt keine Auswirkung auf den Maschinencode. Die maximale Länge des Begrenzers beträgt 6 Zeichen.
SHIFT	SH	Siehe 4.1.4.1 (II)
(NOFKT)	NO	Macht nichts - der Operand ist Kommentar
SHIFTP	PSH	Wie SHIFT, aber der einzusetzende Wert hat ein Vorzeichen.
OFFSET	OFFS	Setzt (Wert-SYSPC) wie SHIFTP ein.

6.4 Loadmodul-Format

Ein durch die Subroutine LOADM erstellter Loadmodulblock hat folgendes Format:



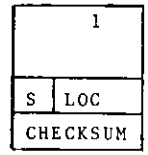
Anzahl der Maschinencode-Worte

Adresse des ersten Wortes

Dieser Wert, addiert zu allen anderen, ergibt als Summe Null

Maschinencode - Worte

Jedes Feld des Blockes ist 2 Bytes lang. Sollte dies nicht ausreichen, so ist das Format des LOADBLK zu ändern (siehe 4.3.5) Der Endblock, der durch die Subroutine ASMEND geschrieben wird, hat folgendes Format:



Kennzeichnung des Endblocks / nur für Nova eindeutig, da dort jeder Block statt N - N als erstes Wort enthält.

Wie oben

Hat das Bit S den Wert 1, so sollte das Ladeprogramm nach dem Endblock ein neues Programm laden können. Ist S dagegen 0, so springt das Ladeprogramm an die durch LOC angegebene Adresse und führt das geladene Programm von dort aus.

6.5 Benutzte Dateien

DD-Name	Beschreibung
LIST	Ausgabedatei für Listen von Syntax-Definitionen und OPCODE-Definitionen
SYSERR	Seq-Datei mit Fehlermeldungen
SYSGETD	Seq-Datei, von der Sprachdefinitionen geholt werden können
SYSIN	Eingabedatei für Karten
SYSPRINT	Ausgabedatei für Liste der Eingabekarten und des erstellten Codes sowie Fehlermeldungen und Cross Reference Tabelle
SYSSAVE	Datei zum Retten von Sprachdefinitionen
TEMP	temporäre Arbeitsdatei
XVOLSER	für jedes Volume XVOLSER, auf dem sich benutzte Makrobibliotheken befinden können, eine DD-Karte //XVOLSER DD UNIT=...,VOL=SER=XVOLSER,DISP=OLD
SYSIIN	Zwischendatei für PASSO, falls dieser angefordert wird (Kartenbilder).

6.6. Fehlermeldungen

Fehler-Nr.	Text
1	Falsches Zuweisungsformat
2	Symbol nicht definiert
3	Symbol-Tabelle voll
4	Typ wird redefiniert
5	Falsches Format bei Typ-Definition
6	Aufhänger ist nicht definiert
7	Label ist mehrfach definiert
8	Typ ist nicht definiert
9	Symbol wird redefiniert
10	Syntax-Endfunktion ist nicht bekannt
11	Syntax-Element ist nicht bekannt
12	Syntax-Definition ist unvollständig
13	Syntax-Element wird redefiniert
14	Falsche Operandenliste
15	Der Vergleichsoperator ist nicht definiert.
16	Adresse <0 ist nicht möglich
17	Aufhänger-Funktion ist nicht bekannt
18	Unzulässiger Begrenzer am Kartenanfang
19	Ausdruck ist falsch
20	Text ist zu lang - erhöhe PSYSBEFL und übersetze CEASM neu
21	Operandenplatz ist schon belegt
22	Operandenwert ist zu groß
23	Fehler beim MACRO-Lesen - DSN,DDN,Member richtig?
24	Text - Syntax ist falsch - > fehlt
25	Falsche Parameterliste bei Aufhängerfunktion
26	Der Aufhänger ist hier nicht zulässig.
27	Einsetzen eines Operanden in den vorherigen Befehl ist nicht möglich.
28	Falsche Hexadezimalzahl
29	Der Operationscode variabler Länge hat keinen Typ.
30	Bei Simulation könnten Host- und Guestwerte nicht übereinstimmen.
31	Maximaler Wiederholungsfaktor ist 32767

Literatur

1. P.J. Brown: Macro Processors and Techniques for Portabel Software. John Wiley & Sons, 1974

2. Ferguson: The Evolution of the Meta-Assembly Program Comm. ACM, 9, 3, 1966

```
 1          0            ; BEISPIEL                                                                  00000100
 2          0            ; MICROPROGRAMM FUER EINEN I/O-KANAL VON PDP15 ZUR DATA LINE (DL)           00000200
 3          0            ; WORTLAENGE DES MICROPROGRAMMS IST 16 BIT                                  00000300
 4          0                                                                                        00000400
 5          0            ; MIKROPROGRAMM - ALLGEMEINE DEFINITIONEN                                   00000500
 6          0           LISTO=0,,LOPC                                                                00000600
 7          0           LISTG=0,,GR                                                                  00000700
 8          0           .LOC=0,,ALOC                                                                 00000800
 9          0                    SYSBASIS=10                                                         00000900
10          0                                                                                        00001000
11          0            ; SYNTAX                                                                    00001100
12          0           GOTO=EL,BEGR                                                                 00001200
13          0           PULSE=EL,BEGR                                                                00001300
14          0           LAB=EL,SH(12)                                                                00001400
15          0           PP=EL,SH(,,4)                                                                00001500
16          0           D=EL,SH(7,,1)                                                                00001600
17          0           LEV=EL,SH(,,7)                                                               00001700
18          0           GE=GOTO LAB                                                                  00001800
19          0           PE=PULSE PP                                                                  00001900
20          0           CC=EL,SH(8,,4)                                                               00002000
21          0           OIF=CC GE|CC GE,PE|CC GE,LEV 0                                               00002100
22          0           OPU=PP                                                                       00002200
23          0           OGO=LAB,PE | LAB                                                             00002300
24          0           OLE=D,PE | D                                                                 00002400
25          0                                                                                        00002500
26          0            ; TYPEN                                                                     00002600
27          0           TIF=(OIF,1)                                                                  00002700
28          0           TGO=(OGO,1)                                                                  00002800
29          0           TPU=(OPU,1)                                                                  00002900
30          0           TLE=(OLE,1)                                                                  00003000
31          0                                                                                        00003100
32          0            ; OPCODES                                                                   00003200
33          0              SYSBASIS=8  ; AB HIER OKTALZAHLEN                                         00003300
34          0           IF=0,TIF,OC                                                                  00003400
35          0           GOTO=400,TGO,OC                                                              00003500
36          0           PULSE=0,TPU,OC                                                               00003600
37          0           TWS=1*20 ,TLE,OC  ; TRANSMITTER WORD SELECT - DATA OR FROMTO                 00003700
38          0           DATAWORD=2*20 ,TLE,OC  ; DATA FROM PDP15 OR ZEROS                            00003800
39          0           MODEBIT=3*20 ,TLE,OC  ; WORD IS DATA OR END OF RECORD(EOR)                   00003900
40          0           RWS=4*20 ,TLE,OC  ; RECEIVER WORD SELECT - DATA OR FROMTO                    00004000
41          0                                                                                        00004100
42          0            ; MIKROPROGRAMM DL-> PDP15  (INPUT)                                         00004200
43          0            ; CONDITION CODES                                                           00004300
44          0           CHRUN=7           ; CHANNEL IN RUN STATE                                     00004400
45          0           NOT.HLT=10                                                                   00004500
46          0           NRREADY=11        ; DL-RECEIVER NOT READY                                    00004600
47          0           ENDCOND=12        ; END CONDITION FOR DATA TRANSFER                          00004700
48          0           DCHIBUSY=13       ; INPUT TO PDP15 BUSY                                      00004800
49          0           N.TWCZRO=14       ; WORD COUNT ¬= 0                                          00004900
50          0            ; D-FIELD                                                                   00005000
51          0           DATA=1          ; DATA WORD IST DATA,NOT ZERO                                00005100
52          0           FROMTO=0        ; WORDSELECT : FROMTO OR DATA                                00005200
53          0            ; PULSE                                                                     00005300
54          0           DEMAND=5        ; DL-QUITTUNG                                                00005400
55          0           STOCHIN=6       ; START INPUT OF ONE WORD TO PDP15                           00005500
56          0           APIIN=7         ; READY-EVENT TO PDP15                                       00005600
57          0           M.FT.PAR=10                                                                  00005700
58          0                                                                                        00005800
59          0              ; PROGRAMM                                                                00005900
60          0                    .LOC 0  ; STARTADRESSE 0                                            00006000
61          0      43500 STARTIN: IF CHRUN GOTO TSTDL, RWS FROMTO  ; TESTE AUFTRAG VON PDP15         00006100
```

CARDNO	LOCATION	VALUE	INPUT CARD		ERROR NR.
62	1	4000	IF NOT.HLT GOTO STARTIN	00006200	
63	2	4400	IF NRREADY GOTO STARTIN	00006300	
64	3	405	DLQUIT: GOTO STARTIN, PULSE DEMAND ; QUITTIERE DL-SENDUNG	00006400	
65	4		;	00006500	
66	4	4400	TSTDL: IF NRREADY GOTO STARTIN ; DL-INPUT?	00006600	
67	5		;	00006700	
68	5	10	PULSE M.FT.PAR ; YES	00006800	
69	6	300	RWS DATA ; RECEIVER WORDSELECT = DATA	00006900	
70	7	135000	IF ENDCOND GOTO CHEND ; CHANNEL END?	00007000	
71	10	6	PULSE STDCHIN ; NO - START INPUT TO PDP15	00007100	
72	11	115400	IF DCHIBUSY GOTO . ; WAIT UNTIL PDP15 READY	00007200	
73	12	36000	IF N.TWCZRO GOTO DLQUIT ; WORDCOUNT = 0?	00007300	
74	13		;	00007400	
75	13	30407	CHEND: GOTO DLQUIT, PULSE APIIN ; YES - EVENT TO PDP15	00007500	
76	14		;	00007600	
77	14			00007700	
78	14		; MICROPROGRAM PDP15 -> DL (OUTPUT)	00007800	
79	14		.LOC 0 ; BEGIN WITH ADDR 0	00007900	
80	0		; CONDITION CODES	00008000	
81	0		NOT.GO=2 ; NOTHING TO WRITE FROM PDP15	00008100	
82	0		N.TREADY=3 ; DL-TRANSMITTER NOT READY	00008200	
83	0		DCHOBUSY=4 ; WORDTRANSFER PDP15 TO CHANNEL IS BUSY	00008300	
84	0		RWCZERO=5 ; WORD COUNT IS ZERO	00008400	
85	0		NDLRESET=6 ; NO DL-RESET	00008500	
86	0		; D-FIELD (TWS)	00008600	
87	0		; (DATAWORD)	00008700	
88	0		PDP15=0 ; DATA FROM PDP15	00008800	
89	0		ZERO=1 ; DATA IS ZERO	00008900	
90	0		; (MODEBIT)	00009000	
91	0		DAT=1 ; WORD IS DATA	00009100	
92	0		EOR=0 ; WORD IS EOR	00009200	
93	0		; PULSES	00009300	
94	0		DCHOUT=1 ; GET WORD FROM PDP15	00009400	
95	0		APIOUT=2 ; READY-EVENT TO PDP15	00009500	
96	0		WRITE=3	00009600	
97	0		WRT.STRT=4 ; WRITE FROMTO, START WRITING TO DL	00009700	
98	0		;	00009800	
99	0	1000	STARTOUT:IF NOT.GO GOTO STARTOUT ; TESTE AUFTRAG VON PDP15	00009900	
100	1	1620	IF N.TREADY GOTO STARTOUT, TWS DATA	00010000	
101	2		;	00010100	
102	2	41	OUTPUT: DATAWORD PDP15, PULSE DCHOUT ; KANAL AKTIV - LESE WORT	00010200	
103	3	32260	IF DCHOBUSY GOTO . , MODEBIT DAT ; FERTIG?	00010300	
104	4	20	TWS FROMTO ; WORDSELECT=FROMTO	00010400	
105	5	4	PULSE WRT.STRT ; START DL-OUT WITH FROMTO	00010500	
106	6	122620	IF RWCZERO GOTO SENDEOR, TWS DATA ;WC = 0?	00010600	
107	7	3000	IF NDLRESET GOTO STARTOUT ; DL-RESET DURING OUTPUT?	00010700	
108	10		;	00010800	
109	10	402	LASTWORD:GOTO STARTOUT, PULSE APIOUT ; END,EVENT TO PDP15	00010900	
110	11		;	00011000	
111	11	1000	TSTBUSY: IF NOT.GO GOTO STARTOUT	00011100	
112	12	111640	SENDEOR: IF N.TREADY GOTO TSTBUSY, DATAWORD ZERO ;DL-TRANSM. READY?	00011200	
113	13	63	MODEBIT EOR, PULSE WRITE ; YES - SET EOR,WRITE WORD	00011300	
114	14	20	TWS FROMTO	00011400	
115	15	100404	GOTO LASTWORD, PULSE WRT.STRT ; WRITE LAST WORD,HALT CHAN.	00011500	