KfK 2666
KfK-CAD 71
September 1978

REGENT-Handbuch

E. G. Schlechtendahl, K. H. Bechler, G. Enderle,
K. Leinemann, W. Olbrich

Institut für Reaktorentwicklung
Projekt Rechnerunterstütztes Entwickeln,
Konstruieren und Fertigen

Kernforschungszentrum Karlsruhe

Als Manuskript vervielfältigt
Für diesen Bericht behalten wir uns alle Rechte vor

KERNFORSCHUNGSZENTRUM KARLSRUHE GMBH

KERNFORSCHUNGSZENTRUM KARLSRUHE
Institut für Reaktorentwicklung
Projekt Rechnerunterstütztes Entwickeln
Konstruieren und Fertigen

KfK 2666
KfK-CAD 71

R E G E N T - H A N D B U C H

E.G.Schlechtendahl
K.H.Bechler
G.Enderle
K.Leinemann
W.Olbrich

Kernforschungszentrum Karlsruhe G.m.b.H.,Karlsruhe

Übersicht

Das REGENT-Handbuch gibt eine Einführung in die Anwendung des integrierten Programmsystems REGENT Version 1.3. Es enthält eine vollständige Beschreibung der Syntax und Semantik aller dem Benutzer zur Verfügung stehenden Fähigkeiten des Systemkerns und der wichtigsten systemzugehörigen Subsysteme. Die Fehlermeldungen des Systemkerns und der Subsysteme werden erläutert.

Summary

E. G. Schlechtendahl, K. H. Bechler, G. Enderle,
K. Leinemann, W. Olbrich

REGENT - Manual

The REGENT-Manual introduces the user into the capabilities of the integrated program system REGENT Release 1.3. The manual provides a complete description of syntax and semantics of all functions available to the user, both for the system kernel itself and its main subsystems. The error messages of the kernel and the subsystems are described.

An der Erstellung des REGENT-Systems und des Handbuchs haben mitgewirkt:

K.H.Bechler
A.Brunner
G.Enderle
F.Katz
H.Kühl
K.Leinemann
W.Olbrich
A.Pee
A.Riffel
E.G.Schlechtendahl
U.Schumann
R.Schuster
A.Steil
I.Wittemann

Das REGENT-System wurde vom Institut für Reaktorentwicklung im Kernforschungszentrum Karlsruhe von 1973 bis 1975 entwickelt. Seit 1974 geschah die Entwicklung in Abstimmung mit dem Projekt CAD des Bundesministers für Forschung und Technologie.

Inhalt

Dies ist REGENT-Handbuch

Exemplar Nr. .....

Anmerkung:

Exemplare ohne eingetragene Nummer werden nicht verfolgt.

Für Exemplare mit eingetragener Nummer wird eine Verteilerliste geführt. Änderungen am Handbuch werden diesem Verteiler automatisch zugestellt. Inhaber solcher Exemplare werden gebeten, sich an

Herrn Dr. Schlechtendahl
Institut für Reaktorentwicklung
Kernforschungszentrum Karlsruhe
Postfach 3640
7500 Karlsruhe 1

zu wenden, wenn sich die Anschrift ändert oder kein Handbuch mehr benötigt wird.

Anleitung zur Benutzung des Handbuchs

Das REGENT-Handbuch ist weder ein Lehrbuch noch eine vollständige Systembeschreibung. Das REGENT-Handbuch soll dem Anwender - das ist hier insbesondere der Entwickler von Subsystemen - folgendes mitteilen:

- Welche Fähigkeiten bietet REGENT, die über die reine Verwendung von PL/1 hinausgehen?
- Wie ist die korrekte Schreibweise von Anweisungen, um diese Fähigkeiten zu benutzen?
- Welcher Grund kann eine von REGENT gemeldete Fehlersituation hervorgerufen haben?

Entsprechend ist auch die Gliederung des Handbuchs gestaltet.

In Kap.1 wird eine Übersicht über das Gesamtsystem gegeben, so daß ein mit den Fähigkeiten integrierter Systeme im Prinzip Vertrauter sich notfalls allein einarbeiten kann. Dennoch ist dies kein gleichwertiger Ersatz für Einführungskurse oder ein Lehrbuch.

Kap.2 beschreibt den generellen Gang der Subsystementwicklung

Definition der Subsystemdatenstruktur
Definition der Subsystemmodule
Definition der Subsystemsprache.

Aus Gründen der Übersichtlichkeit ist die detaillierte Beschreibung der Modulentwicklung mit dem Modulgenerator (Kap.3) und der Programmiersprache PLR (Kap.4), sowie die Definition von Subsystemanweisungen mit PLS (Kap.5) zu eigenen Kapiteln zusammengefaßt.

Kap.6 erläutert die Fähigkeiten, die das REGENT-System dem Subsystem-Anwender zur Verfügung stellt, ohne daß der Subsystem-Entwickler dafür besondere Maßnahmen ergreifen muß.

Kap.7 gibt Empfehlungen für die Subsystementwicklung.

Kap.8 erläutert den Zusammenhang zwischen dem REGENT-System und dem Betriebssystem. Insbesondere werden hier die Dateien beschrieben, die für ein Funktionieren des REGENT-Systems eingerichtet und durch die Steuersprache des Betriebssystems dem REGENT-System verfügbar gemacht werden müssen. Dieser Teil beschränkt sich zur Zeit auf Anlagen der Serien IBM/370 und IBM/360 unter OS.

Kap.9 enthält die Handbücher für die Datei-Verwaltungs-Subsysteme DABAL und EDIT.

Kap.10 enthält in alphabetischer Reihenfolge die Anweisungen und speziellen Ausdrücke der Programmiersprache PLR, soweit sie über PL/1 hinausgeht. Neben einer formalen Syntaxbeschreibung werden Erläuterungen und Beispiele gegeben. Dieser Teil ist als Nachschlagwerk gedacht.

Kap.11 enthält in alphabetischer Reihenfolge die Anweisungen und spezielle Ausdrücke der Programmiersprache PLS.
Kap.12 erläutert die graphische Schnittstelle des Systemkerns.
Kap.13 ist eine Sammlung der Fehlernachrichten des Systemkerns.

Kap.14 ist eine Liste veröffentlichter und unveröffentlichter Dokumentation zu REGENT.

Die jetzt vorliegende Version 1.3 des Handbuches ist konsistent mit der REGENT-Version 1.3 vom 1.4.1978. Gegenüber der vorläufigen Version 1.2 wurden die am REGENT-System in zwei Jahren intensiven Einsatzes vorgenommenen Verbesserungen berücksichtigt.

K A P I T E L 1

Einführung in das REGENT-System

1.1 Einleitung

REGENT wurde von 1973 bis 1975 im Institut für Reaktorentwicklung (IRE) im Kernforschungszentrum Karlsruhe entwickelt. Die Entwicklung richtet sich vor allem auf die Erstellung eines sogenannten Systemkerns im Sinne von ICES . Obwohl ICES als Konzeptgrundlage benutzt wurde, sind wesentliche Verbesserungen in folgender Hinsicht festzustellen:

- eine mächtigere Basissprache (PL/1 anstelle von FORTRAN) erleichtert die Erstellung von Subsystemen

- die Basissprache PL/1 und alle ihre Vorteile sind Bestandteil aller problemorientierten Sprachen innerhalb des REGENT-Systemes. Dies erleichtert die Anwendung von Subsystemen.

1.2 Entwurfsgrundlage

ICES kann als Prototyp einer Reihe integrierter CAD-Systeme angesehen werden : Hierzu gehören GENESYS , IST , POLO und ebenso REGENT. An dieser Stelle sollen daher diejenigen Entwurfsgrundlagen erläutert werden, die nicht allen diesen Systemen mehr oder weniger gemeinsam sondern eher für REGENT spezifisch sind.

1.2.1 Die Ebenen der Rechneranwendung im CAD-Bereich

Im IRE gesammelte Erfahrung mit einer großen Anzahl von CAD Programmen aus dem Bereich der Reaktortechnik und mit verschiedenen ICES-Versionen (MIT , PSU , Mc-Auto) hat deutlich gezeigt, daß wir zwischen 3 verschiedenen Ebenen der Rechneranwendung zu unterscheiden haben:

Subsystementwicklung,
Subsystemanwendung,
parametrische Benutzung.

Die Trennung von Subsystementwicklung und Subsystemanwendung ist keine Neuerung gegenüber üblicher Praxis. Auch bei reinen FORTRAN- oder PL/1-Programmen ist es oft so, daß Programme von anderen Gruppen benutzt werden als nur von der Gruppe, die das Programm erstellte. Diese Trennung bringt - gutgenutzt - alle Vorteile sinnvoller Arbeitsteilung mit sich, bedeutet andererseits aber, daß bei der Programm- bzw. Subsystemerstellung mehr Sorgfalt walten muß, als wenn ein Programm nur von seinem Ersteller benutzt wird:

- Das Subsystem muß überprüfen, ob die ihm vom Anwender übergebene Aufgabe innerhalb des vom Ersteller vorgesehenen Anwendungsbereichs liegt. Dies erfordert vom Ersteller eine sehr präzise formale Definition des Gültigkeitsbereichs des Subsystemes.

- Das Subsystem muß so dokumentiert sein, daß sein Anwendungsbereich deutlich erkennbar ist, seine korrekte und effektive Handhabung leicht erlernt werden kann.

Es sei deutlich gesagt, daß der REGENT-Systemkern dem Subsystemersteller diese Arbeiten nicht abnimmt, ihm aber die Konzention auf diese wichtigen Arbeiten erleichtert, da er ihn bei der Erstellung und beim Austesten des Subsystems selbst entlastet.

Die Anwendung eines Subsystems in einem integrierten System wie REGENT unterscheidet sich sehr wesentlich von der Anwendung eines FORTRAN-Programmes. Der Programm-Anwender kann in der Programmeingabe nur einige Parameter zahlenmäßig definieren, während der REGENT-Subsystem-Anwender eine ganz neue - vom Subsystemersteller definierte - Programmiersprache zur Verfügung hat, die auch alle Möglichkeiten von PL/1 beinhalten kann. Dies erweist sich als besonders vorteilhaft, wenn der Subsystemanwender die Subsystemfähigkeiten durch eigene Zusatzalgorithmen ergänzen muß, wenn eine Aufgabe in ähnlicher Weise mit variierten Parametern wiederholt werden muß (parametrische Benutzung). Dies kann zum Beispiel innerhalb einer Optimierungsrechnung oder bei interaktiver Benutzung über einen Fernschreiber oder ein Bildschirmgerät geschehen.

1.2.2 Annahmen hinsichtlich Hardware und Software

Hinsichtlich der Hardware wird angenommen, daß eine ständig steigende Zahl von möglichen Anwendern über Hilfsmittel der Datenfernverarbeitung leichter und billiger auf Rechner mit großem Arbeitsspeicher (über 120 K Bytes) werden zugreifen können, als dies 1978 möglich ist.

Hinsichtlich der Mensch-Maschine-Kommunikation werden in REGENT in der vorliegenden Form nur Stapelverarbeitung und Terminals vom Fernschreibertyp zugrundegelegt. Auf der Ebene der Subsysteme kann jedes andere Kommunikationsmittel unterstützt werden. Interaktive graphische Kommunikation ist jedoch noch nicht soweit standardisiert, daß ihre Einbeziehung in den Systemkern zweckmäßig wäre.

Hinsichtlich der Software ist eine der Annahmen, die REGENT zugrundeliegen, daß auch in technischen Anwendungsbereichen PL/1 künftig FORTRAN in steigendem Maße verdrängen wird. Die Standardisierungsbemühungen von ECMA und ANSI sind weit fortgeschritten . Mehrere Rechnerhersteller unterstützen entweder bereits heute PL/1 oder haben dies angekündigt . Auf Seiten der Anwender ist zu beobachten, daß kaum ein Ingenieur, der einmal von FORTRAN auf PL/1 übergewechselt ist, diesen Schritt bereut. Im Gegenteil, auch im technischen Bereich ist festzustellen, daß die prinzipiellen Möglichkeiten von PL/1, vor allem aber die in PL/1 verfügbaren Testhilfen die Programmentwicklung wesentlich erleichtern. Der Aufruf von FORTRAN-Unterprogrammen ist bei Beachtung einiger Regeln möglich.

1.2.3 Entwurfsrichtlinien

REGENT soll die 3 Ebenen der Rechneranwendung (Problemprogrammierung, Anwendungsprogrammierung, parametrische Benutzung) gleichermaßen unterstützen.

REGENT soll auf allen Benutzerebenen sämtliche Fähigkeiten der Basissprache PL/1 bereitstellen.

REGENT soll das Subsystemkonzept, wie es ursprünglich in ICES angelegt war, unterstützen, jedoch mit wesentlich verbesserten

Möglichkeiten hinsichtlich des Datenaustausches.

REGENT soll besonders effektiv arbeiten, wenn eine Folge von Anweisungen unmittelbar hintereinander auszuführen ist. ("10 Minuten Antwortszeit für ein Problem, nicht 10 Sekunden für eine Anweisung").

Der REGENT-Systemkern soll soweit wie möglich in einer höheren Programmiersprache geschrieben sein, damit der Aufwand für Pflege und Übertragung auf andere Anlagen reduziert wird.

REGENT soll die Integration der kommerziellen Seite einer Produktentwicklung mit der technischen Seite unterstützen. Dies wird dadurch erleichtert, daß als Basissprache PL/1 gewählt wurde, welches die im kommerziellen Bereich übliche Sprache COBOL ebenso abdeckt wie FORTRAN.

1.3 Der Aufbau des REGENT-Systems

Jede Installation des REGENT-Systems besteht aus folgenden Teilen

- dem Systemkern
- dem Subsystem PLS für die Definition problemorientierter Sprachen
- dem Subsystem DABAL für Datenverwaltungszwecke
- einer Anzahl von problemorientierten oder firmenspezifischen Subsystemen

Der Systemkern selbst besteht aus Teilsystemen für

- die Modulverwaltung
- die Verwaltung dynamischer Datenstrukturen
- die Datenbankverwaltung
- die Nachrichtenverwaltung
- die Übersetzung problemorientierter Sprachen
- und die Systemimplementierung.

Die ersten vier dieser Teilsysteme bilden das REGENT-Laufzeitsystem. Ein Überblick über das Gesamtsystem wird in Abb. 1 gegeben. Je nach Anwendungsebene braucht der Benutzer von dieser Gesamtkonfiguration jedoch nur einen Teil zu kennen.

Abb. 2 veranschaulicht die Subsystementwicklung. Der Subsystementwickler erstellt

- eine Datenstruktur für das neue Subsystem mit Hilfe des Subsystemes PLS,

- Übersetzerprogramme für die Anweisungen der neuen problemorientierten Subsystemsprache mit Hilfe des Subsystemes PLS,

- Verarbeitungsmodule, die in PLR - einer Erweiterung von PL/1 - programmiert werden, mit Hilfe eines Modulgenerators.

Die drei so erzeugten Dateien (Anweisungsübersetzer, Datenstrukturbibliothek, Modulbibliothek) werden für die Subsystemanwendung benötigt.

Abb. 3 zeigt die Subsystemanwendung. Der Subsystemanwender übergibt dem REGENT-Übersetzer ein Programm, das in PL/1 und einer oder mehreren Subsystemsprachen geschrieben ist. Der REGENT-Übersetzer übersetzt dieses Programm in ein reines PL/1-Programm, welches dem PL/1-Übersetzer der Rechenanlage und anschließend dem Binder übergeben wird. Normalerweise gelangt das so erzeugte ausführbare Programm unmittelbar in den Arbeitsspeicher zur Ausführung. Es ist jedoch auch möglich, dieses Programm auf einer Privatbibliothek für spätere Anwendung zu speichern.

Abb. 4 zeigt die Ausführung eines REGENT-Programmes, wie sie sich auch dem parametrischen Benutzer darstellt. Während der Ausführung stehen dem Programm alle in PL/1 oder REGENT-unterstützten Möglichkeiten zur Verfügung, insbesondere also auch die Möglichkeit, interaktiv mit dem Anwender zu kommunizieren.

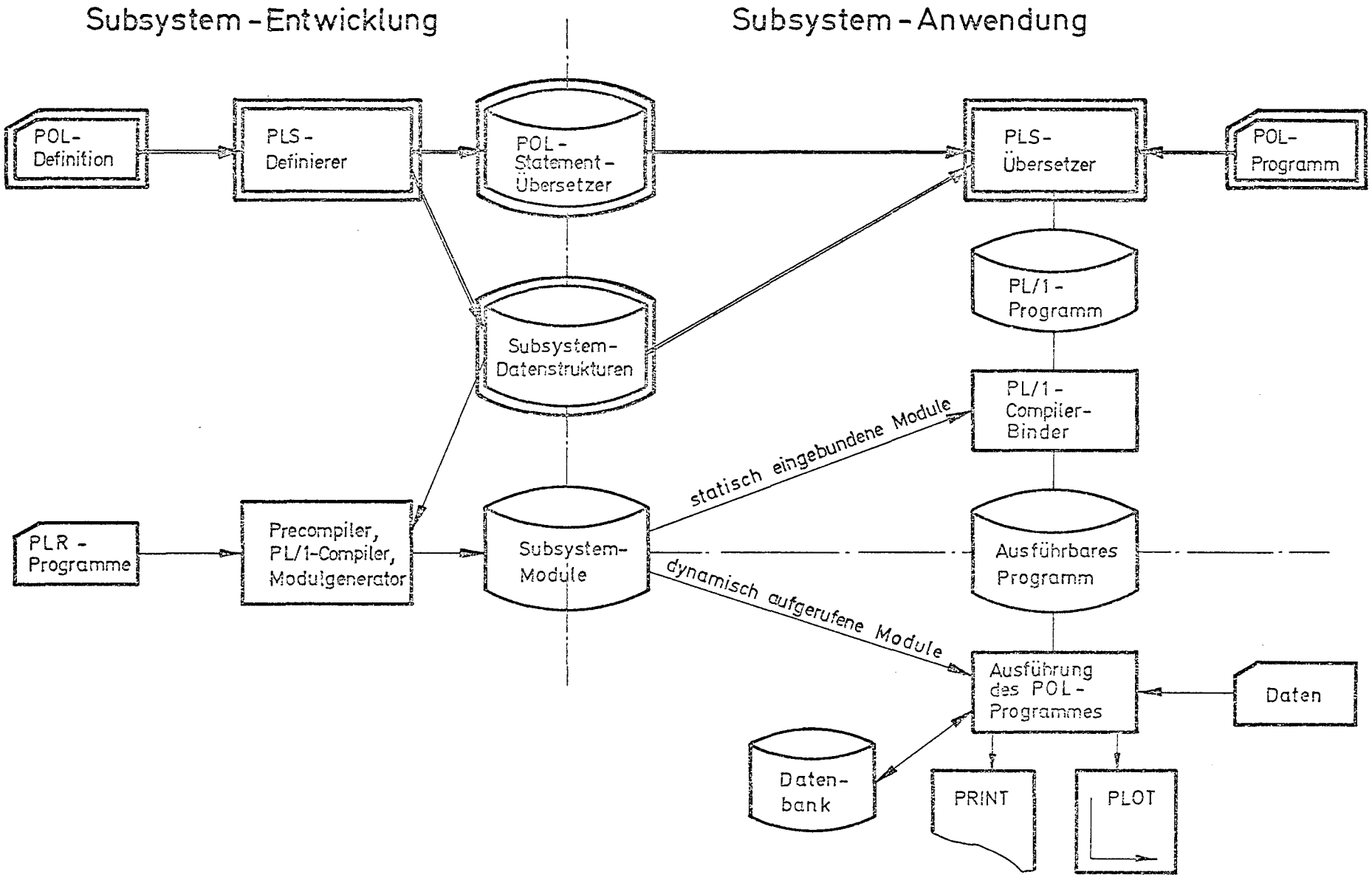


Abb. 1 Gesamtkonfiguration des REGENT - Systems

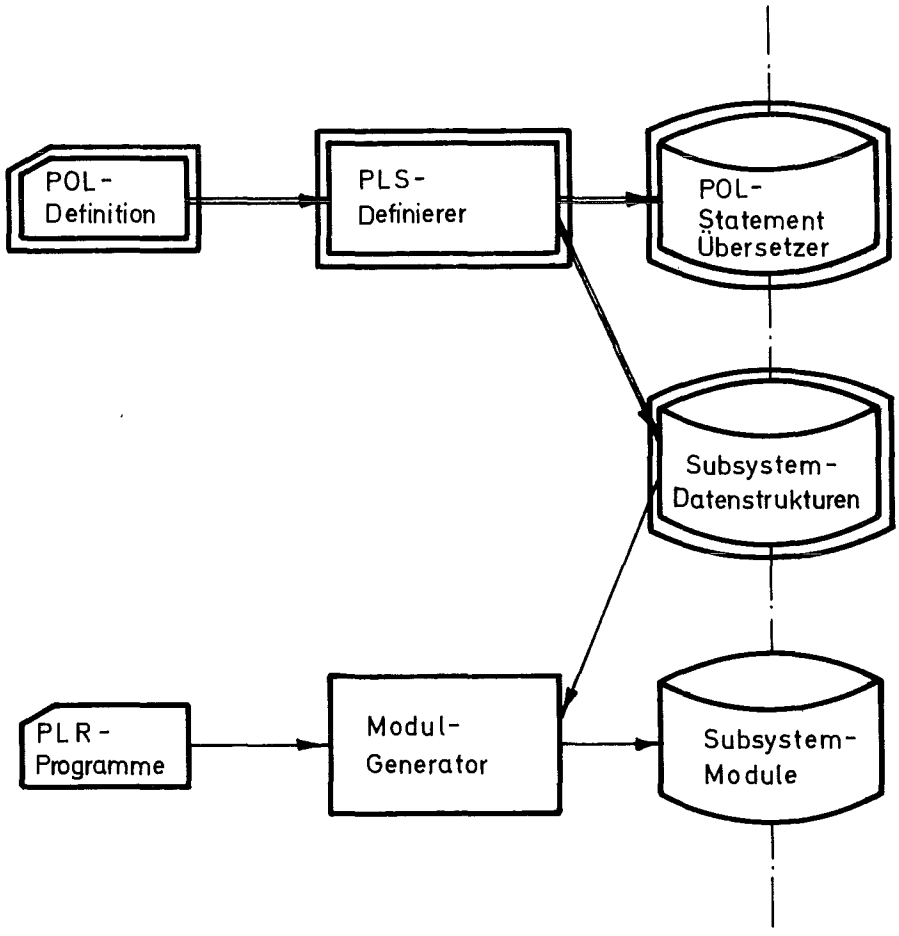


Abb. 2 Subsystem-Entwicklung

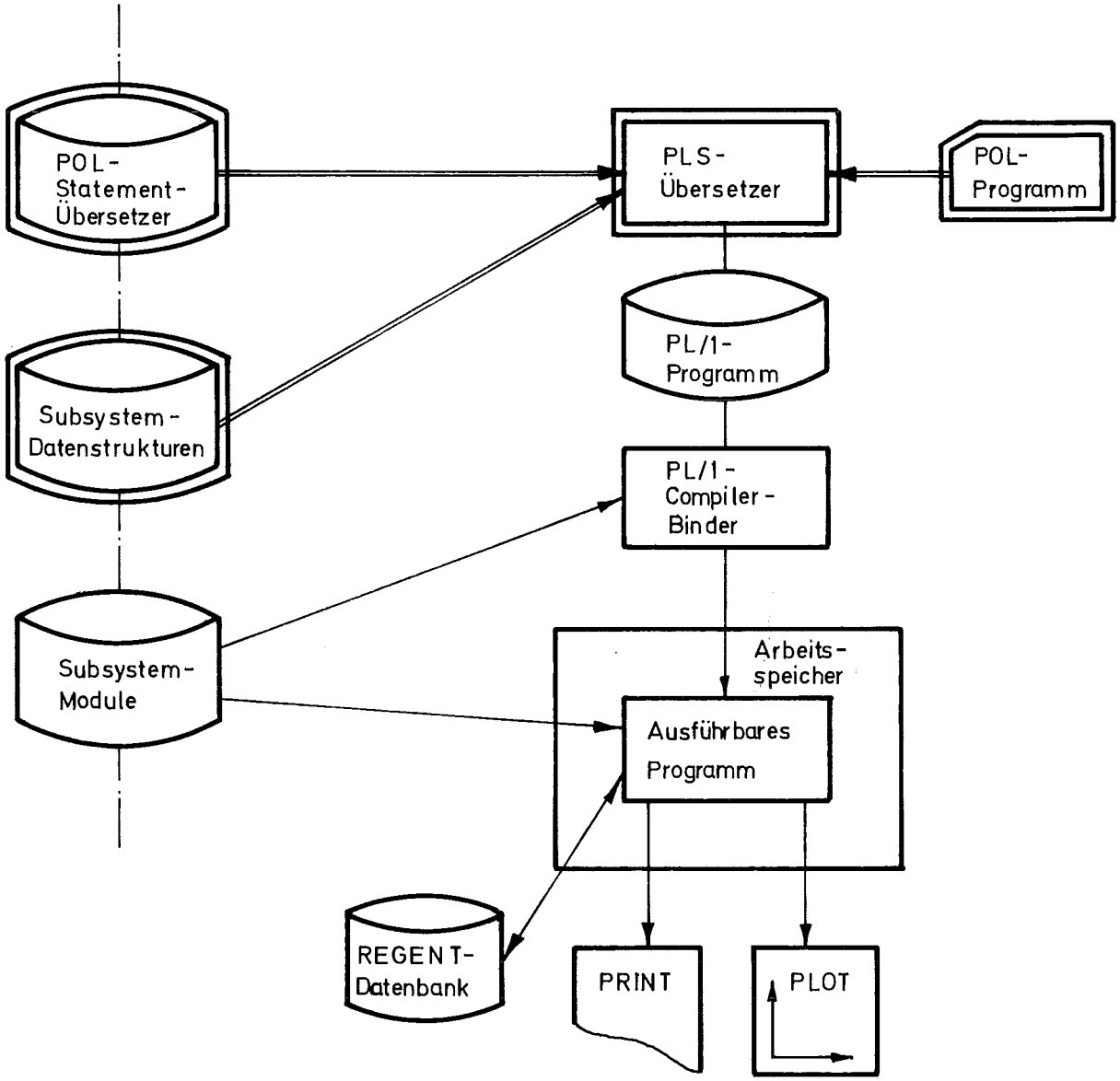


Abb. 3 Subsystem - Anwendung

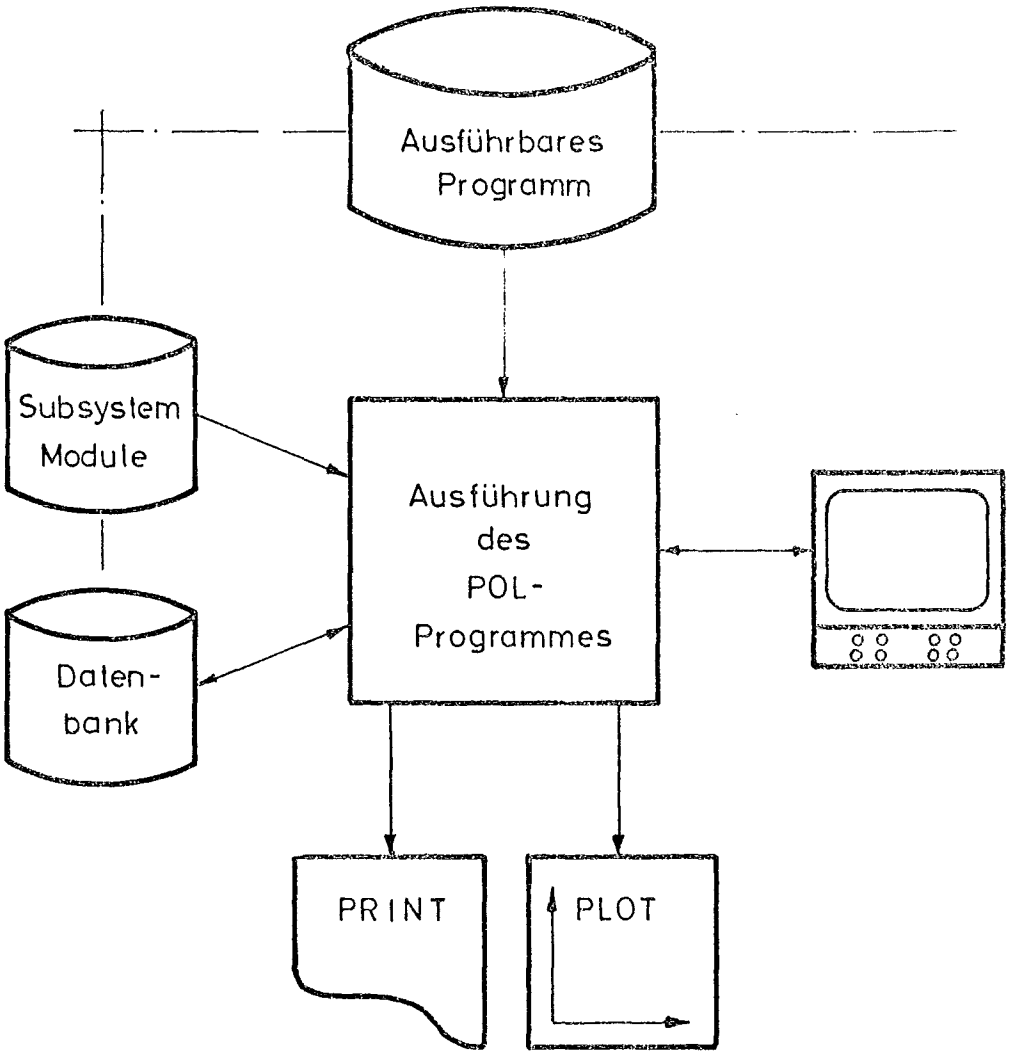


Abb. 4 Interaktive Programmausführung mit REGENT

K A P I T E L 2

Subsystementwicklung

2.1 Allgemeines

Die Entwicklung eines neuen REGENT-Subsystems erfolgt in folgenden Einzelschritten:

A Subsystemspezifikation
B Einrichten des Subsystems
C Definition der Subsystemdatenstrukturen
D Erstellen der Subsystemsprache
E Erstellen der Module des Subsystems
F Testen des Subsystems
G Dokumentation

Natürlich werden diese Schritte nicht nur ein einziges Mal und genau in dieser Reihenfolge durchlaufen. Meist müssen Schritte wiederholt werden, manche Teile können auch parallel ausgeführt werden (z.B. Schritte D, E und G).

2.2 Subsystemspezifikation

Die Subsystemspezifikation sollte ausgehen von möglichst klar formulierten Wunschvorstellungen hinsichtlich der Subsystemfähigkeiten. Sie sollte insbesondere festlegen

- die Syntax und Wirkung der Subsystembefehle
- die Datenstruktur des Subsystems.

2.3 Einrichten des Subsystems

Ein neues Subsystem wird mit der INITIATE-Anweisung des PLS-Subsystems eingerichtet. Die Anweisung DESTROY in PLS löscht ein Subsystem wieder. Auf die Verwendung des Paßwortes in diesen Anweisungen ist zu achten.

2.4 Definition der Subsystemdatenstrukturen

REGENT kennt folgende Subsystemdatenstrukturen:

- den Subsystem-COMMON, bestehend aus der COMMON-Struktur und den Zusatzvereinbarungen

- die Ausführungszeitdatenstruktur
- die Übersetzungszeitdatenstruktur.

Für den Subsystem-COMMON und die Ausführungszeitdatenstruktur muß jeweils festgelegt werden, ob die darin enthaltenen Informationen auch unmittelbar dem Anwender des Subsystems (z.B. in PL/1-Anweisungen) zugänglich sein sollen oder ob sie dem Anwender gegenüber geschützt sein sollen.

2.4.1 Der Subsystem-COMMON

Der Subsystem-COMMON wird definiert durch die zwischen den PLS-Anweisungen

```
DATA COMMON;
---
END DATA;
```

stehenden Vereinbarungen. In diesem COMMON können vereinbart werden:

die COMMON-Struktur
und Zusatzvereinbarungen für den Common in Form von
- PL/1-Deklarationen,
- PL/1-Prozeduren,
- REGENT-Deklarationen
  (DYNAMIC ENTRY, DESCRIPTOR, POOL, BANK).

Die im Subsystem-COMMON definierte COMMON-Struktur und die Zusatzvereinbarungen werden bei der Ausführung des Subsystems stets nur einmal bei der Anweisung ENTER erzeugt. Das bedeutet, daß alle Informationen (Daten und Prozeduren), die während der ganzen Ausführung eines Subsystems genau einmal existieren müssen, in diesem Subsystem-COMMON definiert sein müssen.

Ferner ist zu beachten, daß alle Informationen im Subsystem-COMMON, die nicht Bestandteil der COMMON-Struktur sind, nur in dem Anwendungs-Programm direkt zugänglich sind, in dem die Anweisung ENTER steht, nicht jedoch in den Moduln des Subsystems oder in den Moduln, die mit der Sprache dieses Subsystems erstellt werden. Sollen sie auch in den Moduln zugänglich sein, so muß dies stets auf dem Wege über die COMMON-Struktur sichergestellt werden. Die Zusatzvereinbarungen werden daher im Normalfall nur zur Initialisierung der COMMON-Struktur dienen.

! Beispiel

```
!
!        SUBSYSTEM XXX;
!        DATA COMMON;
!        DCL 1,
!           2  E ENTRY INIT( E),
!           2  PK PTR INIT(ADDR( K)),
!           2 TITEL CHAR(60) VARYING;
!        DCL  K INIT(10);
!
!         E: PROC;
!         ---
!         END  E;
!        END DATA;
```

! Erläuterung

! Der COMMON dieses Subsystems besteht aus

! der COMMON-Struktur und den Zusatzvereinbarungen für
! - die Variable K
! - die Prozedur E.

! Die Variable K und die Prozedur E sind in den
! Moduln dieses Subsystems und in den Moduln, die mit der
! Sprache dieses Subsystems erstellt werden, über die
! COMMON-Struktur zugänglich.

2.4.2 Die COMMON-Struktur

Die COMMON-Struktur wird durch eine der Deklarationen in DATA COMMO vereinbart. Sie unterscheidet sich von den übrigen dort evtl. stehe den Deklarationen, welche Zusatzvereinbarungen darstellen, durch folgende Merkmale:

- sie muß als erste dort stehen,
- es muß eine Datenstruktur sein, deren Name entweder fehlt oder mit dem vollen Namen des Subsystems identisch ist.

Die COMMON-Struktur ist in allen Moduln des Subsystems direkt zugänglich. Sie ist auch in allen Moduln, die mit der Sprache dieses Subsystems erstellt werden, direkt zugänglich. Die Namen der Variablen in der COMMON-Struktur können, wenn die Eindeutigkeit es erfordert, mit dem vollen Namen des Subsystems qualifiziert werden.

! Beispiel
!
! Die Subsysteme XXX und YYY sollen beide in der COMMON-Struktur
! eine Variable BILD_GROESSE haben. Dann sind z.B. folgende Anweisungen möglich:
!

```
!           ENTER XXX;
!           ---
!           ENTER YYY;
!           ---
!             YYY.BILD_GROESSE = XXX.BILD_GROESSE;
!             ---
!           END YYY;
!           END XXX;
```

Da alle Module eines Subsystems von der COMMON-Struktur abhängen, prüft REGENT stets, ob die COMMON-Struktur, die bei Erstellung eines Moduls benutzt wurde, dieselbe ist wie bei der Verwendung des Moduls. Maßgeblich ist das Erstellungsdatum der COMMON-Struktur bzw. des Moduls. Diese Prüfung stellt sicher, daß alle Module die gültige COMMON-Struktur enthalten. Sie erfordert natürlich, daß nach Änderung der COMMON-Struktur alle Module des Subsystems neu erstellt werden müssen.

2.4.3 Die Ausführungszeitdatenstrukturen

Die Ausführungszeitdatenstrukturen werden definiert durch die Daten- und Prozedurvereinbarungen, die zwischen den PLS-Anweisungen

```
DATA name;
---
END DATA;
```

stehen. name ist dabei eine in Hochkomma eingeschlossene Zeichenkette

mit bis zu 32 Zeichen. Es können mehrere derartiger Ausführungszeitdatenstrukturen definiert werden. Sie werden dann von REGENT zusammengefaßt.

Die Ausführungszeitdatenstrukturen werden stets dann bereitgestellt, wenn die Sprache des Subsystems benutzt wird. Das ist einerseits der Fall nach ENTER, andererseits aber auch nach einer Anweisung

```
entryname: PROCEDURE REGENT(SUBS=subsystemname);
```

In die Ausführungszeitdatenstrukturen sind daher solche Informationen aufzunehmen, die für die Ausführung einzelner Anweisungen nötig sind. Typischerweise gehören hierher die Vereinbarungen von DYNAMIC ENTRYs, die nur aus der Übersetzung von Subsystemanweisungen heraus aufgerufen werden, nicht jedoch aus Moduln des Subsystems.

2.4.4 Die Übersetzungszeitdatenstruktur

Die Übersetzungszeitdatenstruktur wird nur in seltenen Fällen benötigt. Sie ist in Kap.5 näher erläutert.

2.4.5 Schutz der Subsystemdatenstrukturen

Die Daten- und Prozedurvereinbarungen

- des Subsystem-COMMON,
- der Zusatzvereinbarungen für den COMMON und
- der Ausführungszeitdatenstrukturen

werden bei Übersetzung eines Anwenderprogramms in das erzeugte PL/1-Programm eingefügt. Sofern (wie meist üblich) die PL/1-Anweisungen Bestandteil der Subsystemsprache sind, so kann der Anwender im Prinzip direkt auf Daten in diesen Datenstrukturen zugreifen. Es muß daher sichergestellt werden, daß der Anwender nicht versehentlich Daten des Subsystems verändert. Wenn der Anwender innerhalb seines Programms neue Blöcke einführt (BEGIN oder PROC), so muß sichergestellt sein, daß der Bezug zwischen der Vereinbarung von Subsystemdaten und ihrer Verwendung nicht beeinflußt wird. Beide Ziele werden auf folgende Weise erreicht:

Regel 1: Alle Daten- und Prozedurnamen in den DATA-Definitionen eines Subsystems, die dem Anwender nicht zugänglich sein sollen, müssen mit dem Zeichen # beginnen. Bei Verwendung dieser Variablen in Anweisungsdefinitionen und PLR-Programmen muß das Zeichen # ebenfalls verwendet werden.

Regel 2: Alle Daten- und Prozedurnamen, die dem Anwender zugänglich sein sollen, müssen in der Benutzungsanleitung des Subsystems beschrieben sein.

Regel 3: Die Verwendung von Namen, die mit dem Zeichen # beginnen, ist in Subsystemsprachen verboten.

Regel 4: Die Verwendung von Namen, die mit QQ beginnen, ist grundsätzlich verboten.

2.5 Erstellen der Subsystemsprache

Die Subsystemsprache wird mittels der STATEMENT- und CLAUSE-Definitionen von PLS definiert. Sofern nicht ausdrücklich durch DESTROY anders festgelegt, enthält jede Subsystemsprache auch alle PL/1-Anweisungen. Siehe Kap. 5. Das PLS-Handbuch ist Kap.11.

2.6 Erstellen der Module des Subsystems

Module für ein Subsystem werden mit dem Modulgenerator erzeugt. Als Programmiersprache für diese Module steht PLR und jede bereits vorhandene andere Subsystemsprache zur Verfügung. Siehe Kap. 3. Das PLR-Handbuch ist Kapitel 10.

K A P I T E L 3

Module und Routinen

3.1 Module und Routinen

Jedes REGENT-Subsystem besteht aus drei wesentlichen Komponenten:

der Subsystemdatenstruktur,
der Subsystemsprache und
den Subsystemmoduln.

Der Modulgenerator übernimmt die Erzeugung der Subsystemmodule. Die Module werden bei der Ausführung des Subsystems dynamisch von dem MAIN-Modul oder aus anderen Moduln in den Arbeitsspeicher geladen, die in ihnen enthaltenen Prozeduren werden ausgeführt, und die Module werden bei Platzmangel nach Benutzung wieder aus dem Arbeitsspeicher beseitigt. (siehe DECLARE ENTRY DYNAMIC)

Außer Moduln kann der Modulgenerator auch sogenannte Routinen erzeugen. Routinen werden im Gegensatz zu Moduln fest in den MAIN-modul oder in andere Module eingebunden. Sie müssen daher als ENTRY EXTERNAL (zusätzlich mit dem Attribut REGENT, falls es sich nicht um eine reine PL/1-Prozedur handelt) dort deklariert sein, wo sie benutzt werden sollen. Das Einbinden in den MAIN-Modul geschieht durch den Lader über einen Automatic Library Call oder über den Linkage Editor mittels einer INCLUDE-Anweisung bzw. ebenfalls Automatic Library Call. Das Einbinden von Routinen in andere Routinen oder Module geschieht mit dem Modulgenerator.

Module werden von REGENT in einer Modulbibliothek verwaltet. Die Routinen sind vom Subsystementwickler selbst zu verwalten. Routinen sollen daher nicht in REGENT-Bibliotheken gespeichert werden.

3.2 Format der Modulgeneratoreingabe

Die Eingabe für den Modulgenerator besteht aus logischen Sätzen von je 80 Zeichen Länge (Lochkartenformat). Diese Sätze bilden entweder Steueranweisungen an den Modulgenerator oder externe Prozeduren.

Steueranweisungen beginnen mit × in Spalte 1 der Lochkarte, gefolgt von einem Kommandowort (,das in Spalte 2 beginnen muß,) und setzen sich fort bis zum nächsten Semikolon. Falls die Steueranweisung nicht vor Spalte 72 mit dem Semikolon endet, ist sie auf der folgenden Lochkarte von Spalte 2 an wiederum bis maximal Spalte 71 fortzusetzen. Auf diese Art darf eine Steueranweisung sich über maximal 5 Karten erstrecken. Falls nach dem Semikolon auf derselben Karte noch Information steht, so wird diese ignoriert.

Alle Karten, die nicht das Zeichen × in Spalte 1 haben, werden bis zur folgenden Steueranweisung bzw. bis Eingabeende als eine externe Prozedur behandelt.

3.3 Modulerzeugung

Der Modulgenerator kann in einem Lauf Module und Routinen für mehrere Subsysteme erstellen. Das jeweilige Subsystem wird durch eine

×SUBSYSTEM-Anweisung

identifiziert. Das betreffende Subsystem muß zuvor mittels PLS initialisiert worden sein.

Die Erzeugung eines Moduls wird durch die

×MODUL-Anweisung

eingeleitet. Jeder Modul muß vom Subsystementwickler einen innerhalb des Subsystems eindeutigen Namen von maximal 6 Zeichen Länge zugewiesen bekommen. Das REGENT-System sorgt selbst dafür, daß keine Konflikte entstehen, wenn in verschiedenen Subsystemen derselbe Modulname benutzt wird. Ein Modul besteht stets aus einer oder mehrerer externen Prozeduren. Einige oder alle dieser externen Prozeduren können aus anderen Moduln heraus aufgerufen werden. Diese werden als RMM-Entries bezeichnet. Alle externen Prozeduren des Moduls können natürlich auch von allen anderen externen Prozeduren desselben Moduls aufgerufen werden.

Sofern ein Modul nur einen RMM-Entry besitzt, der außerdem namentlich mit dem Modulnamen identisch ist, genügt die Angabe des Modulnamens auf der ×MODUL-Anweisung. Andernfalls müssen alle RMM-Entries auf der ×MODUL-Anweisung angegeben werden. Die Namen der RMM-Entries müssen mit Namen von externen Prozeduren des Moduls übereinstimmen und dürfen maximal 6 Zeichen lang sein.

Zur Definition des Moduls, die sich an die ×MODUL-Anweisung anschließt, können folgende Bestandteile dienen:

- externe Prozeduren; diese beginnen (mit evtl. Ausnahme der ersten Prozedur) mit einer ×PROCESS-Anweisung. Die ×PROCESS-Anweisung gibt an, in welcher Sprache die externe Prozedur geschrieben ist und mit welchen Optionen der entsprechende Übersetzer sie übersetzen soll.

- ×INCLUDE-Anweisungen, mit denen zuvor mit dem Modulgenerator erzeugte Routinen, aber auch andere Unterprogramme, die in übersetzter und teilweise gebundener Form auf anderen Bibliotheken liegen, dem Modul beigefügt werden. Diese anderen Unterprogramme können auch mit Programmiersprachen erstellt worden sein, die der Modulgenerator nicht selbst verarbeitet (z.B. FORTRAN, Assembler, COBOL).

- Sonstige Anweisungen zur Modulgestaltung. Diese Anweisungen unterscheiden sich in ihrer Wirkung nicht von den gleichlautenden Anweisungen an den Linkage Editor. Nicht erlaubt sind Anweisungen, die den Linkage Editor Anweisungen NAME und ENTRY entsprechen.

- Ferner können Unterprogramme ohne besondere Steueranweisung durch Automatic Library Call eingebunden werden. Sie müssen auf einer Bibliothek stehen, die über die DD-Karte SYSLIB erreicht werden kann.

Im Sinne des Betriebssystems erzeugt der Modulgenerator auf der Modulbibliothek ein Member, dessen Name aus Subsystemprefix und Modulname zusammengesetzt ist. Bestehende Module können mit der ×DESTROY MODULE-Anweisung wieder gelöscht werden.

3.4 Routinenerzeugung

Der Modulgenerator kann in einem Lauf Module und Routinen für mehrere Subsysteme erstellen. Das jeweilige Subsystem wird durch die

×SUBSYSTEM-Anweisung

identifiziert. Das betreffende Subsystem muß zuvor mittels PLS initialisiert sein.

Die Erzeugung einer Routine wird durch die

×ROUTINE-Anweisung

eingeleitet. Jede Routine muß vom Subsystementwickler einen Namen von maximal 8 Zeichen Länge zugewiesen bekommen. Hinsichtlich der Namensgebung sind einige Bedingungen zu beachten, die davon abhängen, ob die Routine später in einen vom Modulgenerator erzeugten Modul (RMM-Modul) eingebunden wird oder ob sie bei Anwendung des Subsystems Bestandteil des MAIN-Moduls wird.

Sofern die Routine in einen RMM-Modul eingebunden wird, muß ihr Name lediglich innerhalb dieses Moduls eindeutig sein.

Sofern die Routine in den MAIN-Modul eingebunden wird, muß dafür gesorgt werden, daß der Routinenname mit keiner anderen extern deklarierten Variable im MAIN-Modul identisch ist. Da im MAIN-Modul Namensgebungen aus verschiedenen Subsystemen zusammenkommen, ist dies theoretisch nicht zu garantieren. Eine hilfreiche Regel lautet: Namen von Routinen für den MAIN-Modul müssen mit dem Subsystemprefix beginnen. Besser ist die Vermeidung dieser Technik. Das bedeutet: Es wird dringend empfohlen, in den DATA-Anweisungen eines Subsystems (siehe PLS) nicht

DECLARE ... ENTRY EXTERNAL

zu benutzen, sondern stattdessen diese Routinen zu RMM-Moduln zu machen, die als DYNAMIC ENTRY im Subsystemcommon vereinbart werden können.

Zur Definition einer Routine stehen dieselben Möglichkeiten zur Verfügung wie für die Definition eines Moduls. Zusätzlich ist die Anweisung ×ENTRY für den Linkage Editor erlaubt.

Im Sinne des Betriebssystems erzeugt der Modulgenerator auf der Routinenbibliothek ein Member, dessen Name gleich dem Routinennamen ist.

Routinen sollten grundsätzlich in einer Privatbibliothek gespeichert werden. Es wird empfohlen, die ×ROUTINE Anweisung stets mit dem Parameter LIB zu versehen (siehe unten).

Bestehende Routinen können mit der

×DESTROY MEMBER-Anweisung

wieder gelöscht werden.

3.5 PLR-Prozeduren

In den wohl häufigsten Fällen einer Modul- bzw. Routinendefinition wird PLR als Programmiersprache zu benutzen sein. Dazu ist im Anschluß an die ×MODUL- oder ×ROUTINE-Anweisung eine

×PROCESS PLR-Anweisung

einzufügen, auf die unmittelbar das PLR-Programm folgt.

Beispiel:

```
×SUBSYSTEM ABC;
×MODUL XYZ ENTRIES (A,B);
×PROCESS PLR;
    A: PROC;
    -------
    PLR-Anweisungen
    PL/1-Anweisungen
    END A;
×PROCESS PLR M;
    B: PROC;
```

```
    ----
    PLR-Anweisungen
    PL/1-Anweisungen
    PL/1-Makroprozessor-Anweisungen
    -----
END B;
```

Die Option PLR auf der Anweisung darf entfallen. Sofern der PL/1-Übersetzer mit seinen Standardoptionen arbeiten soll, müssen keine weiteren Angaben auf der Anweisung stehen. Andernfalls müssen abweichende Optionen auf der Anweisung angegeben werden.

Der Modulgenerator arbeitet wie folgt: Er prüft, ob die Option MACRO oder INCLUDE für den PL/1-Übersetzer verlangt wird. Ist dies der Fall, so übergibt er die bis zur nächsten ×-Anweisung oder bis Eingabeende folgende Eingabe dem PL/1-Übersetzer mit den Optionen M, MD, NSYN. Es wird also lediglich der PL/1-Makroprozessor benutzt. Der damit erzeugte Quellcode wird dem PLR-Precompiler übergeben. Der von dem PLR-Precompiler erzeugte PL/1-Text wird dem PL/1-Compiler (ohne die Option M) übergeben. Der vom PL/1-Compiler erzeugte Objektcode wird einer Datei zugefügt, die allmählich mit dem gesamten Objektcode und den Linkage-Editor-Anweisungen für den Modul bzw. die Routine gefüllt wird. Sofern weder die Option MACRO noch INCLUDE für den PL/1-Compiler angegeben ist, wird lediglich die Eingabe unmittelbar dem PLR-Precompiler zugeführt.

3.6 Problemsprachen-Prozeduren

In vielen Fällen kann es zweckmäßig sein, für die Erstellung von Moduln oder Routinen eine problemorientierte Sprache eines Subsystems zu benutzen, das zuvor mit REGENT entwickelt worden ist. Es ist sogar möglich, ein Subsystem auf diese Weise mit seinen eigenen Mitteln zu erweitern. Hierfür muß eine

×PROCESS REGENT-Anweisung

benutzt werden, auf die dann die Eingabe in einer Form erfolgt, wie sie aus der Anwendung von Subsystemen bekannt ist.

Beispiel:

```
*SUBSYSTEM ABC;
*MODUL A;
*PROCESS REGENT;
    A: PROC REGENT (SUBSYSTEM = ABC, NOINIT);
       ----
       PL/1-Anweisungen
       REGENT-System-Anweisungen
       Anweisungen des Subsystems ABC
       ENTER UVW;
         ----
       Anweisungen des Subsystems UVW
         ----
       END UVW;
    END A;
```

Erläuterung

Dieser Modul kann aus PLR-Programmen des Subsystems ABC dynamisch aufgerufen werden. In der Prozedur wird das Subsystem UVW eröffnet, benutzt und wieder verlassen.

Auf der *PROCESS REGENT-Anweisung können ebenso wie auf der *PROCESS PLR-Anweisung Optionen für den PL/1-Compiler stehen. Sie haben dieselbe Wirkung wie dort. Der Modulgenerator verarbeitet die Eingabe nach der *PROCESS REGENT-Anweisung ebenso wie bei PLR; es wird lediglich anstelle des PLR-Precompilers der REGENT-Problemsprachenübersetzer benutzt. In der REGENT-Option auf der PROCEDURE-Anweisung müssen die Angaben NOINIT und SUBSYSTEM=name stehen, damit das System die REGENT-Fähigkeiten im Modul bereitstellen kann. "name" muß der Name des Subsystems sein, zu dem der Modul gehört.

3.7 PL/1-Prozeduren

Sollen in Module oder Routinen PL/1-Prozeduren aufgenommen werden, in denen keine REGENT-Fähigkeiten benutzt werden, so können diese im Anschluß an eine

```
*PROCESS PL/1-Anweisung
```

angefügt werden. Auch diese Anweisung kann Optionen für den PL/1-Compiler enthalten.

Beispiel

```
*PROCESS PL/1   M,XREF,NS;
    A: PROC (X,Y,Z) REORDER;
       ----
       PL/1-Anweisungen
       PL/1-Makroprozessor-Anweisungen
       ----
    END A;
```

Bei Benutzung dieser Anweisung werden die REGENT-spezifischen Übersetzer nicht aufgerufen; die Eingabe wird direkt dem PL/1-Compiler übergeben.

Achtung: In diesen Proceduren dürfen keine FILE- oder CONTROLLED-Variable deklariert werden, die nicht als Parameter an die Procedur übergeben werden. Fügt man in eine solche Prozedur zu Testzwecken eine PUT-Anweisung ein, so muß sie als PLR-Procedur erzeugt werden.

3.8 Die *INCLUDE-Anweisung

Sollen in eine Routine oder in einen Modul zuvor erzeugte Routinen oder Unterprogramme aus anderer Quelle in Form von Objekt- oder Lademoduln eingebunden werden, so geschieht dies mit der

```
*INCLUDE-Anweisung.
```

Beispiel

```
*INCLUDE  LIB1(MEMBER1);
```

Erläuterung

Die DD-Karte LIB1 muß einen partitioned dataset definieren, der Objektmodule (von einem Compiler erzeugt) oder Lademodule (vom Linkage Editor erzeugt) als Members enthält. Aus diesem Dataset wird das Member MEMBER1 entnommen und in die Routine bzw. den Modul eingebaut. Der Modulgenerator gibt die Anweisung dazu (nach Entfernen von * und;) an den Linkage Editor weiter.

3.9 Anweisungen an den Linkage Editor

Der Modulgenerator erlaubt es, außer der *INCLUDE-Anweisung auch andere gültige Linkage Editor-Anweisungen an den Linkage Editor weiterzugeben. Dies sind insgesamt

*ALIAS
*CHANGE
*HIARCHY
*INCLUDE
*INSERT
*LIBRARY
*OVERLAY
*REPLACE

Alle diese Anweisungen unterscheiden sich in Form und Wirkung von den Linkage-Editor-Anweisungen nur durch ein * in Spalte 1 und ein ; am Ende der Anweisung.

Dagegen sind die Linkage-Editor-Anweisungen

ENTRY
NAME

im Modulgenerator gesondert zu betrachten. Die Anweisung

*ENTRY

ist nur bei der Erstellung von Routinen, nicht aber für Module erlaubt. Eine *NAME-Anweisung ist überhaupt nicht erlaubt.

3.10 Optionen des PLR-Precompilers, PL/1-Compilers und Linkage Editors

Der PLR-Precompiler, der PL/1-Compiler und der Linkage Editor können durch Angabe von Optionen gesteuert werden. Die sind die gleichen Optionen, die bei Compile-Link-Läufen auf der EXEC-Steuerkarte mit PARM= angegeben werden können. Die Optionen für den PLR-Precompiler werden auf der

×PPARM-Anweisung

angegeben werden.

Der PL/1-Compiler kann durch eine große Anzahl von Optionen gesteuert werden. Diese Möglichkeit steht auch bei Benutzung des Modulgenerators zur Verfügung. Durch Angabe von Optionen auf einer

×CPARM-Anweisung

kann für alle folgenden externen Prozeduren eine andere Liste von Compiler-Optionen definiert werden. Diese gilt dann, sofern nicht auf einer der externen Prozedur vorangehenden ×PROCESS-Anweisung eine neue Liste von Compiler-Optionen angegeben ist. Letztere gilt dann für diese externe Prozedur.

Ähnlich ist es mit den Optionen des Linkage Editors. Hier gelten die bei der Beschreibung der ×LPARM-Anweisung unten festgelegten Standardoptionen. Diese Angaben können jedoch durch eine

×LPARM-Anweisung

geändert werden.

Die Anweisungen ×PPARM, ×CPARM und ×LPARM wirken sich auf die gesamte folgende Eingabe aus.

3.11 Format der Modulgeneratorausgabe

Der Modulgenerator druckt auf der Datei PRINTER eine Ausgabeliste, die entweder nur Nachrichten oder ein Eingabeprotokoll mit Nachrichten enthält. Das Eingabeprotokoll erscheint nur, wenn beim Aufruf des Modulgenerators der Parameter

PARM='LIST'

vorhanden ist.

Das Eingabeprotokoll enthält

- die Modulgeneratoranweisungen (erkennbar an einem * in Spalte 1)
- die Karten zwischen den Modulgeneratoranweisungen, die vom Modulgenerator an die entsprechenden Übersetzerprogramme weitergeleitet werden.

Die Nachrichten sind in Kap.13 erläutert. Zu beachten ist, daß die Nachrichten an der Stelle gedruckt werden, an denen sie festgestellt werden. Es kann vorkommen, daß dies nicht unbedingt an der Stelle in der Liste ist, an der man sie erwarten würde, sondern erst später.

Da der Modulgenerator normalerweise weitere Programme aufruft, die selbst Ausgabe erzeugen, sei auf die Beschreibung dieser Programme hingewiesen. Auf die Datei SYSPRINT schreiben folgende Programme ihre Druckausgabe:

PL/1-Makroprozessor
PLR-Precompiler oder REGENT-Übersetzer
PL/1-Compiler
Linkage-Editor.

3.12 Modulgenerator-Handbuch

Auf der nächsten Seite beginnt das Modulgenerator-Handbuch.

Modulgenerator-Handbuch

Beispiel

```
*PROCESS;
    A: PROC;
    ----
    END A;
*CPARM 'M,IS,S';
    B: PROC;
    ----
    END B;
*PROCESS INCLUDE,NS;
    C: PROC;
    ----
    END C;
```

Erläuterung

Die PL/1-Prozedur A wird mit der Option INCLUDE und den Standardoptionen des PL/1-Compilers übersetzt. Für die Prozedur B gelten die Optionen M, IS und S und für die Prozedur C die Optionen INCLUDE und NS. Der Stern * steht in Spalte 1.

Syntax

```
*CPARM    'options';
    options::= Optionen des PL/1-Compilers
*in Spalte 1
```

Erläuterung

1) Sofern keine *CPARM -Anweisung gegeben ist, gelten die im Compiler standardmäßig angegebenen Optionen.

2) Nach Ausführung der *CPARM-Anweisung gelten die auf der Anweisung angegebenen expliziten Optionen für alle folgenden PL/1-Übersetzungen.

3) Jeweils nur für eine einzelne externe Prozedur können die PL/1-Compiler-Optionen auf der *PROCESS-Anweisung angegeben werden.

4) Im Gegensatz zur PROCESS-Anweisung müssen die Optionen auf der CPARM-Anweisung zwischen Apostrophen stehen.

Beispiel

```
xDESTROY MODUL(MODULA,MODULB) LIB(PRIVLIB);
xDESTROY MODUL(XYZ);
xDESTROY MEMBER(A1, A2) LIB(TESTLIB);
```

Erläuterung

Es werden die Module MODULA und MODULB in der Bibliothek mit dem DD-Namen PRIVLIB gelöscht. Dann der Modul XYZ in der Standard-Modulbibliothek und die Routine A1 und A2 in der Bibliothek mit dem DD-Namen TESTLIB.

Syntax

×DESTROY { MODUL / MEMBER } (m[,m]) [LIB(lib)],

m::= modulname (6 Zeichen) bzw.
Membername = routinename (8 Zeichen)

lib::= ddname der Modul bzw.Routinenbibliothek (8 Zeichen)

× in Spalte 1

Erläuterung

Das × DESTROY - löscht einen bestehenden RMM-Modul bzw. eine RMM-Routine in der Bibliothek. Es kann auch eine Liste von Namen angegeben werden. Fehlt die LIB-angabe, so wird der Modul in der Modulbibliothek mit DD-namen SYSLMOD abgelegt.

Beispiel

```
*SUBSYSTEM  SSS;
*MODUL  ABC;
*PROCESS  PLR;
    ABC:PROC (FILEX);
      DCL FILEX FILE;
      DCL UVW ENTRY (FILE) EXTERNAL REGENT;
      -----
      -----
      CALL UVW (FILEX);
    END ABC;
*PROCESS REGENT; REGENT(NOINIT,SUB=SSS);
    UVW: PROC(F);
      DCL F FILE;
      ENTER DABAL;
      DCL XYZ ENTRY (FILE) EXT;
      CALL XYZ(F);
      -----
      -----
      END DABAL;
    END UVW;
*PROCESS PL/1;
    XYZ: PROC (FILE);
    DCL FILE FILE;
    ----
    ----
    END XYZ;
```

externe
Prozedur

Syntax

Eine Folge von Sätzen im Lochkartenformat, die in Spalte 1 nicht das Zeichen × enthalten.

Erläuterung

1) Der Modulgenerator kann selbst nicht feststellen, ob die vorliegende Folge von Sätzen ohne × in Spalte 1 eine gültige Prozedur darstellen. Er übergibt lediglich diese Satzfolge den aufgrund der ×PROCESS-Anweisung explizit oder implizit festgelegten Sprachprozessoren (PLR, REGENT oder PL/1) zur Verarbeitung.

Beispiel

```
*SUBSYSTEM  SSS;
*ROUTINE  SSR1 LIB(USER);
*PROCESS REGENT;
    SSR1: PROC(X) REGENT(NOINIT,SUB=SSS);
    -----
    END SSR1;
*PROCESS PLR;
    R2: PROC(Y);
    -----
    END R2;
*ALIAS R2;
*INCLUDE  USER(XXSUB);
```

Erläuterung

Es wird eine Routine erzeugt, die in der Routinenbibliothek USER unter dem Namen SSR1 mit Alias R2 abgelegt wird. Die Routine besteht aus

- der Prozedur SSR1, in der problemorientierte Sprachen benutzt werden können
- der PLR-Prozedur R2
- der Routine *XXSUB* aus der benutzereigenen Routinenbibliothek USER.

×link-edit

Syntax

×link-edit;

link-edit::= gültige Anweisungen an den Linkage-Editor

× in Spalte 1

Erläuterung

1) Folgende Anweisungen an den Linkage-Editor sind erlaubt:

ALIAS
CHANGE
HIARCHY
INCLUDE
INSERT
LIBRARY
OVERLAY
REPLACE

2) Nicht erlaubt ist

NAME .

3) Nur für die Erzeugung von Routinen (also nach ×ROUTINE), nicht aber für Module ist erlaubt

ENTRY .

Beispiel

```
*MODUL  A;
    A: PROC;
    -----
    END A;
*MODUL  B;
*LPARM 'SIZE=(108K,26K),DCBS,REUS'
*PROCESS M,OPT(2);
    B: PROC;
    -----
    END B;
*MODUL  C;
    C: PROC ;
    -----
    END C;
```

Erläuterung

Modul A wird mit den Standardoptionen des Linkage Editors erzeugt, Module B und C dagegen mit den auf der LPARM-Anweisung angegebenen Optionen.

Syntax

```
*LPARM   'Options';
```

options::= Optionen des Linkage Editors.
* in Spalte 1

Erläuterung

1) Falls keine *LPARM-Anweisung gegeben wurde, gelten für die Erzeugung von Moduln und Routinen folgende Linkage Editor Optionen:

```
Module:    LIST,MAP,SIZE=(108K,26K),DCBS,RENT,REUS
Routinen:  LIST,MAP,NCAL,SIZE=(108K,26K),DCBS,RENT ,REUS
```

2) Im Anschluß an eine *LPARM-Anweisung gelten für Module und Routinen (bis zur nächsten *LPARM-Anweisung) die auf der Anweisung angegebenen Optionen.

3) Die Optionen DCBS und RENT müssen stets angegeben werden.

Beispiel

```
*SUBSYSTEM   SSS;
*MODUL  M1;
    -----
    -----
*MODUL  M2  ENTRIES (E1,E2) LIB (TESTMOD);
    -----
    E1:PROC(X,Y,Z);
    -----
    END E1;
    E2:PROC(U,V,W);
    -----
    END E2;
  -----
  -----
*MODUL  M3;
*PROCESS  REGENT;
    M3:PROC; REGENT(NOINIT,SUBSYSTEM=SSS);
      ENTER DABAL;
      -----
      END DABAL;
    END M3;
```

Erläuterung

Es werden drei Module erzeugt. M1 und M2 sind PLR-Programme. M3 benutzt die problemorientierte Sprache des Subsystems DABAL. M2 wird auf der Bibliothek mit dem DD-Namen TESTMOD gespeichert. Die Module M1 und M2 in der Standardbibliothek.

Syntax

×MODUL m [ENTRIES (e [,e]×)] [LIB(lib)] ,

m::= modulname (6 Zeichen)
e::= RMM-Entryname (6 Zeichen)
lib::= ddname der Modulbibliothek (8 Zeichen)
× in Spalte 1

Erläuterung

1) Alle folgende Eingabe bis zur nächsten

×SUBSYSTEM-Anweisung,
×MODUL-Anweisung,
×ROUTINE-Anweisung oder bis
Eingabeende

beschreibt den Modul m des in der vorhergehenden ×SUBSYSTEM-Anweisung angegebenen Subsystems.

2) Die RMM-Entrynamen e müssen mit Namen externer Prozeduren übereinstimmen, die in dem Modul enthalten sind.

3) Falls LIB(lib) angegeben ist, wird der Modul auf die Datei mit dem DD-Namen lib geschrieben. Andernfalls wird der Modul auf die Datei mit DD-Namen SYSLMOD geschrieben (das ist normalerweis die Modulbibliothek REGENT.MODS, siehe Kap.8.11)

4) Der Modul m wird in der Modulbibliothek (einem partitioned dataset) als Member mit dem Namen prm eingetragen. Dabei ist pr der Subsystemprefix.

Beispiel

```
*SUBSYSTEM S1;
*MOBUL M1;
  ----
   Prozedur 1
*PPARM 'NOLIST';
*PROCESS;
  ----
   Prozedur 2
  ----
*PPARM 'TEST,LIST';
*MODUL M2;
  ----
   Prozedur 3
  ----
```

Erläuterung:

Prozedur 1 wird mit den Standard-Precompiler-Optionen übersetzt: LIST, NOTEST. Die Programmliste wird gedruckt, aber die Zusatzinformation nicht.
Prozedur 2 erzeugt (außer evtl.Fehlermeldungen) keine Precompiler-Druckausgabe. Für Prozedur 3 wird die Programmliste und Zusatzinformation gedruckt.

Syntax *PPARM

```
*PPARM     'options';
options::= option1 [,option2]
option1::= {LIST   }
           {NOLIST }
option2::= {TEST   }
           {NOTEST }
```

*in Spalte 1

Erläuterung

1) Wenn keine Optionen angegeben werden, gelten: LIST, NOTEST

2) LIST bedeutet, daß der PLR-Precompiler eine Liste der Anweisungen der externen Prozedur, versehen mit Statement-Nummern, auf SYSPRINT ausgibt. Bei NOLIST wird die Programmliste unterdrückt.
Fehlermeldungen erscheinen auf jeden Fall auf SYSPRINT.

3) TEST bedeutet, daß der Precompiler auf SYSPRINT interne Kontrollblöcke und Dateien ausdruckt. Eine Liste der Variablen mit Attributen und Cross-References sowie der Programmtext mit Zusatzinformationen werden ausgedruckt. Bei Precompiler-Fehlern sollte das betreffende Programm noch einmal mit der TEST-Option übersetzt werden, um die Fehlersuche zu erleichtern.

Bei NOTEST wird keine zusätzliche Information über das Programm ausgedruckt.

4) Die Optionen gelten für alle Prozeduren bis zur nächsten *PPARM-Anweisung oder bis zum Ende der Eingabe.

Beispiel

```
*SUBSYSTEM  SSS;
*ROUTINE SUB1 LIB(TESTLIB)
   ----
   Routinendefinition
   ----
```

Erläuterung

Für das Subsystem SSS wird eine Routine SUB1 auf die Routinenbibliothek mit DD-Namen TESTLIB geschrieben.

Syntax

*ROUTINE rout [LIB(lib)];
rout::= Routinenname (8 Zeichen)
lib::= ddname der Routinen-Datei (8 Zeichen)
* in Spalte 1

Erläuterung

1) Alle folgende Eingabe bis zur nächsten

*SUBSYSTEM-Anweisung,
*MODUL-Anweisung
*ROUTINE-Anweisung oder bis
Eingabeende

beschreibt eine Routine, die bei der Erzeugung eines Moduls oder bei der Benutzung des zuvor angegebenen Subsystems in den Main-Modul eingebunden werden muß.

2) Falls LIB(lib) angegeben ist, wird die Routine auf die Datei mit DD-Namen lib geschrieben. Andernfalls wird die Routine auf die Datei mit DD-Namen QQROUTLB geschrieben. Von der Benutzung der Standardbibliothek wird abgeraten.

3) Die Routine wird in der Routinenbibliothek (einem partitioned dataset) als ein Member rout eingetragen.

Beispiel

```
*SUBSYSTEM   SSS;
*MODUL  M;
*PROCESS ;
    M: PROC(Z);
    DCL Z BANK;
        DCL SUB(PTR) ENTRY REGENT,
        -----
    CALL SUB(ADDR(Z));
    END M;
*PROCESS  REGENT;
    SUB: PROC(P)REGENT(NOINIT,SUB=SSS);
      DCL P POINTER;
      ENTER DABAL;
      DCL B BANK BASED(P);
      -----
      END DABAL;
    END SUB;
```

Erläuterung

Im Modul M des Subsystems SSS wird ein Unterprogramm aufgerufen, in dem die Fähigkeiten des REGENT-Subsystems DABAL benutzt werden.

Syntax

×PROCESS [language] [options] ;
language::= PLR [REGENT] PL/1
options::= Optionen des PL/1-Compilers
× in Spalte 1

Erläuterung

1) Falls vor einer externen Prozedur keine ×PROCESS-Anweisung steht, wird angenommen, es sei eine Anweisung der Form

×PROCESS PLR;

gegeben worden.

2) Falls die language-Option fehlt, wird PLR angenommen.

3) Die language-Option bedeutet:
PLR: es folgt eine externe PLR-Prozedur.
REGENT: es folgt eine externe Prozedur, in der die problemorientierte Sprache eines oder mehrerer zuvor definierter Subsysteme benutzt werden.
PL/1: es folgt eine externe PL/1-Prozedur.

Beispiele

```
*SUBSYSTEM  S1;
*MODUL ---;
    -----
      Moduldefinition
    -----
*ROUTINE -----;
    -----
      Routinendefinition
    -----
*SUBSYSTEM  S2;
*ROUTINE -----;
    -----
      Routinendefinition
    -----
*MODUL ---;
    -----
      Moduldefinition
    -----
```

Erläuterung

Für Subsystem S1 wird ein Modul und eine Routine erzeugt.
Für Subsystem S2 eine Routine und ein Modul.

Syntax

```
*SUBSYSTEM  s;
  s::= subsystemname
       * in Spalte 1
```

Erläuterung

1) Alle bis zur nächsten *SUBSYSTEM-Anweisung oder bis Eingabeende zu erzeugenden Module gehören zum angegebenen Subsystem.

K A P I T E L 4

Die Programmiersprache PLR für die Modulentwicklung

4.1 Allgemeines

Die Programmiersprache PLR wird für die Erstellung der Subsystem-Module benutzt. PLR enthält den vollständigen Sprachumfang von PL/1 mit zwei Einschränkungen (siehe Kap.4.7). Zusätzlich stellt PLR jedoch Anweisungen und Datentypen zur Verfügung, die für den Aufbau großer Programmsysteme besonders nützlich sind (Kap.4.2-4.5). Einige weniger häufig benutzte Dienstprogramme kann der Modulhersteller direkt aufrufen, ohne daß ihm dafür eine spezielle PLR-Anweisung zur Verfügung steht.(Kap.4.6)

4.2 Modulverwaltung

4.2.1 Übersicht

Die Modulverwaltung (RMM=REGENT Module Management) erlaubt es, in PLR-Programmen

- Programme, die in anderen Moduln enthalten sind, aufzurufen,
- solche Programme explizit in den Arbeitsspeicher zu laden, aufzurufen und explizit wieder aus dem Arbeitsspeicher zu löschen.

Die Programme, die über die Modulverwaltung aufgerufen werden sollen, müssen vom REGENT-Modulgenerator erzeugt worden sein. Sie müssen bei der Erzeugung des Moduls als RMM-Entries dieses Moduls angegeben worden sein.

```
! Beispiel einer Modulgenerierung:
!
!          *SUBSYSTEM   A;
!          *MODUL   M  ENTRIES (M1,M2);
!             M1: PROC;
!             -----
!             END M1;
!          *PROCESS;
!             M2: PROC;
!             -----
!             END M2;
```

In diesem Fall können aus dem Subsystem A die RMM-Entries (d.h. die Prozeduren) M1 und M2 des Moduls M über die Modulverwaltung aufgerufen werden.
Der Modul M und die RMM-Entries M1 und M2 sind innerhalb des Subsystems A eindeutig. In anderen Subsystemen können dieselben Module und RMM-Entrynamen benutzt werden, ohne daß dadurch Konflikte entstehen.

Damit innerhalb eines PLR-Programms, wenn es zur Ausführung gelangt, die Modulverwaltung wirksam werden kann, muß implizit oder

explizit in der Main-Prozedur-Anweisung der betreffenden Subsystemanwendung die Option REGENT(MOD) enthalten sein, bzw. NOMOD darf nicht enthalten sein.Also z.B.

```
PROG: PROCEDURE OPTIONS(MAIN) REGENT(MOD);
```

Hierauf hat der Subsystemanwender zu achten. Der Subsystementwickler muß dies in der Benutzungsanleitung vorschreiben.

4.2.2 Deklaration von Dynamic-Entry-Konstanten

Es ist zu unterscheiden zwischen Dynamic-Entry-Konstanten und Dynamic-Entry-Variablen. Dynamic-Entry-Konstanten können in PLR-Programmen (abgesehen von zusätzlichen Möglichkeiten) ähnlich benutzt werden wie die in PL/1 üblichen EXTERNAL Entries. Sie können aufgerufen, an Unterprogramme übergeben oder einer Dynamic-Entry-Variablen zugewiesen werden. Sie sind a priori definiert und, sofern im Subsystem ein Modul mit entsprechendem RMM-Entry existiert, so wird dieser beim Aufruf des Dynamic Entry zur Ausführung gebracht. Dynamic-Entry-Variable dagegen müssen (wie alle Variablen) vor ihrer Benutzung durch Initialisierung in der Deklaration oder durch Zuweisung oder durch Parameterzuordnung in einem Unterprogrammaufruf erst definiert werden. Der Aufruf undefinierter Dynamic-Entry-Variablen erzeugt einen undefinierten Systemabbruch. Eine Dynamic-Entry-Konstante hat stets die Speicherklasse AUTOMATIC oder BASED. BASED ist bei Dynamic-Entry-Konstanten nur für Level 1 erlaubt. Eine Variable kann jede Speicherklasse haben.
Es ist sehr einfach, ein Unterprogramm aus einem Modul herauszunehmen und es an der Stelle, wo es bisher als EXTERNAL deklariert und gerufen wurde, als DYNAMIC ENTRY zu benutzen. Dazu muß lediglich das Attribut EXTERNAL in DYNAMIC geändert und in wenigen Fällen die Option MODUL und ENTRYNAME angefügt werden.

! Beispiel

```
!            DCL ENAME CHAR(6) ,Q PTR;
!            ENAME='M1'
!            BEGIN;
!            DECLARE A ENTRY (BIN FIXED(15),POOL) RETURNS(CHAR(8))
!                 MODULE('M') ENTRYNAME (ENAME);
             DECLARE B ENTRY DYNAMIC BASED;
             ALLOCATE B SET(Q);
```

! Erläuterung
!
! - A ist der Name des Dynamic Entry. Hier ist es eine Funktion,
! die eine Zeichenkette der Länge 8 zurückliefert
! (RETURNS(CHAR(8))).
!
! - Das Unterprogramm erwartet zwei Argumente:
! eines davon ist ein PL/1-Datentyp, das zweite ist ein in PLR
! erlaubter Datentyp.
! - Bei B handelt es sich um eine Dynamic-Entry-Variable

In obigem Beispiel werden Modulname und RMM-Entryname der aufzurufenden Prozedur explizit angegeben. Der Aufruf kann nur ausgeführt werden, wenn für das Subsystem ein gleichnamiger Modul mit dem angegebenen RMM-Entrynamen mittels des Modulgenerators fehlerfrei erzeugt worden war. Wegen der im Modulgenerator gegebenen Einschränkungen gelten folgende Maximallängen

MODULE (CHAR(6))
ENTRYNAME (CHAR(6))

Falls ENTRYNAME nicht angegeben ist, wird der Name des Dynamic Entry als RMM-Entryname benutzt (evtl. auf die ersten 6 Zeichen verkürzt). Falls MODULE nicht angegeben ist, wird der RMM-Entryname - sei er nun explizit oder implizit gegeben - als Modulname benutzt.

Die Parameter-Deskriptor-Liste, die auf ENTRY folgt, darf enthalten:

- alle in PL/1 erlaubten Deskriptoren
  (z.B. PTR, FILE, (*)BIN FLOAT)

- die PLR Datentypen
  ENTRY DYNAMIC oder DYNAMIC ENTRY
  DESCRIPTOR
  POOL
  BANK

Im RETURNS-Attribut einer Dynamic-Entry-Deklaration darf stehen:

a) alle in PL/1 erlaubten Deskriptoren (z.B. PTR, BIN FLOAT)
b) die PLR-Datentypen
   ENTRY DYNAMIC oder DYNAMIC ENTRY
   POOL
   BANK
   nicht jedoch DESCRIPTOR.

Ebenso wie bei PL/1-Entry-Deklarationen kann die Parameter-Deskriptor-Liste weggelassen werden. Aus Sicherheitsgründen wird davon aber dringend abgeraten.

4.2.3 Deklaration von Dynamic-Entry-Variablen

Durch das zusätzliche Attribut VARIABLE in der Deklaration eines Dynamic Entry wird dieser als Variable deklariert. Der Zusatz kann in selbstverständlichen Fällen (bei Parametern etc.) entfallen. Eine solche Deklaration darf im Gegensatz zu einer Dynamic-Entry-Konstanten-Deklaration (siehe 4.2.2) die Option INIT, aber nicht die Optionen MODULE und ENTRYNAME enthalten. Im übrigen gelten die Regeln der Deklaration von Dynamic-Entry-Konstanten.
Nach korrekter Initialisierung oder Zuweisung kann eine Dynamic-Entry-Variable wie eine entsprechende Konstante benutzt werden.

Beispiele

A)
```
DCL X ENTRY (BIN FLOAT(21))
  RETURNS(BIT) VARIABLE DYNAMIC;
```

B)
```
SUB1: PROC(SUBPROG);
DCL SUBPROG DYNAMIC ENTRY(CHAR(*));
CALL SUBPROG ('PARAMETER');
-----
```

C)
```
SUB2: PROC;
DCL (A,B) ENTRY DYNAMIC;
DCL C(2) ENTRY DYNAMIC INIT (A,B);
DO J=1 TO 2;
CALL C (J);
END;
-----
```

```
!     D)      DCL 1 STRU,
!                 2 DYNENT ENTRY DYNAMIC  VARIABLE;
!                 2 BIT;
!
!     E)       DCL E BASED(P) DYNAMIC ENTRY   VARIABLE;
```

! Erläuterung

- Dynamic Entry Variable unterscheiden sich von Dynamic Entry Konstanten durch die Option VARIABLE (Beispiel A,D,E)

- Falls es sich um Parameter (Beispiel B) oder Felder (Beispiel C) handelt, so kann VARIABLE entfallen, da es sich dann stets um eine Dynamic Entry Variable handelt.

- Dynamic Entry Variable besitzen wie die Konstanten die ENTRY-Option und evtl. die RETURNS-Option. Sie können mit Dynamic Entry Ausdrücken (Konstante oder Variable) initialisiert werden.

4.2.4 Deklaration von Dynamic Entries im Subsystem-Common

Dynamic Entry-Variable die im Subsystem-Common enthalten sind, brauchen in den PLR-Programmen des Subsystems nicht deklariert zu werden. Sie sind automatisch im ganzen Subsystem bekannt. Siehe hierzu Kap.2.

! Beispiel

! In PLS sei für den Subsystem -Common des Subsystems XXX folgende Vereinbarung definiert:

```
!              DATA COMMON;
!                DCL 1,
!                  -----
!                  -----
!                  2 SAVE_ROUTINE ENTRY (DEC FLOAT(6)) VARIABLE
!                       DYNAMIC INIT (SAVE);
!                  -----;
!                DCL SAVE DYNAMIC ENTRY (DEC FLOAT(6)) MODUL('SAVE')
!                END COMMON;
```

```
!  Außerdem sei mit Hilfe des Modulgenerators folgender Modul er-
!  zeugt worden:
!              *SUBSYSTEM XXX;
!              *MODUL SAVE;
!                 SAVE: PROC(ZEIT);
!                   DCL ZEIT DEC FLOAT;
!                   -----
!                 END SAVE;
!
!  Dann kann in jedem anderen Modul dieses Subsystems XXX ohne be-
!  sondere Vereinbarung die SAVE_ROUTINE dynamisch aufgerufen
!  werden durch z.B.
!               CALL SAVE_ROUTINE (TIME*3600.);
!
```

Ergänzend sei vermerkt, daß diese Routine auch in jeder Statementdefinition (zwischen EXEC; und END EXEC; oder zwischen EXEC und ;;) aufgerufen werden kann. Dort muß allerdings LINK statt CALL stehen, z.B.:

```
              EXEC LINK SAVE_ROUTINE (TIME*3600.);;
```

4.2.5 Aufruf von Dynamic Entries

Der Aufruf von Dynamic Entries, unabhängig davon,ob es sich um Konstante oder Variable handelt, geschieht in gleicher Weise wie der Aufruf von externen Prozeduren in PL/1.

Aufruf von Subroutinen

```
!  Beispiele
!
!     A)      DCL SUB1 ENTRY DYNAMIC;
!               CALL SUB1;
!
!     B)      DCL SUB3 ENTRY (BIT,POOL)
!                     DYNAMIC VARIABLE INIT (SUB2);
!             CALL SUB3 ('1'B,REGENT_POOL);
```

! Erläuterung
!
! Beispiel A ist selbsterklärend. In Beispiel B wird eine Dynamic
! Entry Variable SUB3 aufgerufen. Als Parameter wird ein Bit und
! ein REGENT-Datenpool übergeben. Für SUB2 und REGENT_POOL müssen
! die entsprechenden Deklarationen vorhanden sein.

Aufruf von Funktionen

! Beispiel

```
!              DCL TEXT CHAR(80) VARYING;
!              DCL TEXT_FUN ENTRY (BIN FIXED)
!                    ENTRY DYNAMIC RETURNS (CHAR(20))
!                    ENTRYNAME('TEXTXF'),
!              TEXT = TEXT_FUN(13);
```

! Erläuterung
!
! Die Anwendung unterscheidet sich nicht von der Anwendung eines
! EXTERNAL ENTRY. Da der Funktionsname zu lang ist (über 6 Zeichen)
! wurde im Beispiel der RMM-Entryname (und damit implizit der
! Modulname) explizit angegeben.

Aufrufe von Dynamic Entry Funktionen dürfen stehen

in allen Ausdrücken (arithmetisch, logisch, Zeichenketten etc.) aller PL/1- und PLR-Anweisungen.

4.2.6 LOAD und RELEASE

Wenn ein Dynamic Entry häufig aufgerufen werden soll, so kann die Effektivität der Aufrufe dadurch gesteigert werden, daß der Dynamic Entry vor der Benutzung durch einen LOAD-Befehl geladen wird.
Der Loadbefehl bewirkt folgendes:

- Es wird festgestellt, welche Dynamic-Entry-Konstante durch den Befehl referiert wird. (Auch wenn eine Dynamic-Entry-Variable angegeben ist, wird durch diese letztlich eine Dynamic-Entry-Konstante referiert.)

- Wenn der betreffende Dynamic-Entry noch nicht geladen ist, so wird der zugehörige RMM-Modul in den Arbeitsspeicher gebracht, wo er bis zum zugehörigen RELEASE verbleibt. Gleichzeitig wird veranlaßt, daß alle folgenden Aufrufe unmittelbar (also ohne Umweg über die Modulverwaltung) zu dem betreffenden RMM-Entry gelangen.

- Wenn die referierte Dynamic-Entry-Konstante bereits geladen war, ist die Anweisung ohne Wirkung.

Dieser Effekt muß vom PLR-Programmierer wieder rückgängig gemacht werden, wenn der betreffende Dynamic Entry nicht mehr häufig benutzt wird. Dies geschieht über den RELEASE-Befehl.

! Beispiel

```
!            A: PROC(PROG) REGENT;
!                DECLARE PROG ENTRY (BIN FIXED) DYNAMIC;
!                LOAD PROG;
!                DO J=1 TO 10000;
!                  CALL PROG(J);
!                END;
!                RELEASE PROG;
!            END A;
```

! Erläuterung

!

! - Wegen der häufigen Aufrufe von PROG ist ein LOAD-RELEASE
! sinnvoll. Man beachte, daß es unwesentlich ist, ob der
! RMM-Entry der letztlich durch PROG aufgerufen wird,
! (PROG selbst ist eine Variable) bereits geladen ist oder
! nicht.

4.3 Dynamische Datenstrukturen

4.3.1 Übersicht

Die DDS-(Dynamische Datenstrukturen)-Verwaltung dient der benutzerfreundlichen Bearbeitung flexibler und möglicherweise sehr großer hierarchisch strukturierter Datenmengen, deren Teilmengen geordnet sind. Elementartyp einer solchen Teilmenge ist eine einfache Liste (Vektor, eindimensionales Feld). Diese Vektoren dürfen als Elemente Daten enthalten (DDE- bzw. Dynamische-Daten-Element-Vektoren), oder aber es können selbst wieder Listen (Deskriptor-Vektoren) sein. Damit können mehrdimensionale Datenfelder, aber auch Datenbäume und allgemeinere hierarchische Strukturen aufgebaut werden. Wesentlich für die DDS ist, daß sie dynamisch sind, d.h. daß sie während ihrer Benutzung nicht nur ihren Inhalt ,sondern auch ihren Aufbau ändern können. Diese Änderungen beziehen sich nicht nur auf die Länge der Listen, sondern auch auf die Art ihrer Elemente. Diese Struktur unterliegt nur der Beschränkung, daß alle Elemente eines DDE des gleichen Typs sein müssen. Wohl aber kann eine DDS mehrere DDE unterschiedlichen Typs enthalten.

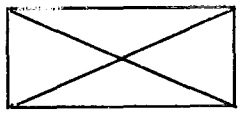
In vielen Bereichen des CAD (z.B. der Strukturmechanik) werden sehr große Datenmengen bearbeitet, deren Größe sowohl herkömmliche Arbeitsspeicher als auch virtuelle Speicherbereiche überschreitet. Die DDS-Verwaltung stellt daher auch einen virtuellen Datenspeicher bereit, der nur durch die Größe des Platten- oder Trommelspeicherbereichs einer Betriebssystemdatei begrenzt ist.

Zur langfristigen Speicherung der DDS kann die REGENT-Datenbank-Verwaltung verwendet werden, die DDS besonders unterstützt.

Die Verwendung von DDS erfordert beim Benutzer das Verständnis des logischen Aufbaus dieser Datenstrukturen und der zu ihrer Beschreibung benutzten Begriffe.

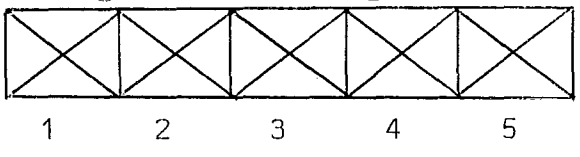
Dynamisches Datenelement

Ein dynamisches Datenelement (DDE) enthält letztlich die Information, die der Benutzer in der hierarchischen Struktur eines DDS speichern kann. Ein DDE kann selbst eine ganze Datenstruktur sein, allerdings nur noch mit der Flexibilität, die durch PL/1-STRUCTURES der Speicherklasse BASED ohne die REFER-Option geboten wird. Alle Dimensionen von Feldern in DDEs müssen daher als Konstante angegeben sein. Ein DDE kann selbst aber durchaus mehrere Felder von Daten, Bits und Zeichenketten enthalten. Für den mit ICES oder IST vertrauten Anwender mag von Interesse sein, daß in diesen beiden Systemen das Gegenstück zu einem DDE jeweils nur eine einzelne Date (Festkommazahl, Gleitkommazahl, Zeichen) sein kann. In graphischer Darstellung wird ein DDE hier stets durch einen durchkreuzten Kasten repräsentiert:



DDE-Vektor

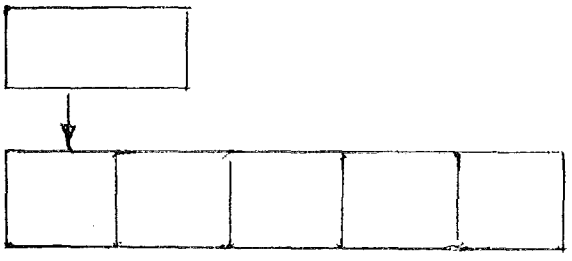
Wenn man eine Menge gleichartiger DDEs zusammenfaßt und den einzelnen Elementen die Zahlenfolge 1, 2, 3 etc. zuordnet, so erhält man einen DDE-Vektor. Die zugehörige Nummer aus der Zahlenfolge wird Index genannt:



Deskriptor, Deskriptorvektor

Ein Deskriptor ist ein REGENT-spezifisches Datenelement, das vom Anwender nicht direkt verändert werden kann, von dessen Existenz er aber wissen muß. Ein Deskriptor (durch einen leeren Kasten dargestellt) bildet für die DDS-Verwaltung den Zugang zu einem DDE-Vektor

oder auch - da auch Deskriptoren zu einer Liste zusammengefaßt werden können - den Zugang zu einem Deskriptorvektor.



Die einzelnen, durch einen Index 1, 2, 3 usw. gekennzeichneten Deskriptoren können jeweils wieder auf einen Deskriptorvektor oder einen DDE-Vektor verweisen.

Basisdeskriptor

Eine ganze dynamische Datenstruktur ist wegen dieser Art des logischen Aufbaus über einen einzigen Deskriptor zugänglich. Dieser wird Basisdeskriptor genannt.

Teildatenstruktur

Die Datenmenge, die von einem Deskriptor, der nicht Basisdeskriptor ist, zugänglich ist, wird Teildatenstruktur (TDS) genannt.

Der Benutzer von REGENT arbeitet entweder mit einzelnen dynamischen Datenelementen (DDE) oder mit Teildatenstrukturen (TDS) oder ganzen dynamischen Datenstrukturen. In den beiden letzteren Fällen handelt es sich stets um spezielle REGENT-Anweisungen oder um Unterprogrammaufrufe, im ersten Fall dagegen besteht - bis auf wenige Ausnahmen - kein Unterschied zwischen einem DDE und einer normalen PL/1-Variablen.

4.3.2 Deklaration, Initialisieren und Rücksetzen von Basisdeskriptoren

DDS sind stets über Basisdeskriptoren zugänglich. Die Benutzung von DDS setzt daher die Deklaration der entsprechenden Basisdeskriptoren voraus.

! Beispiel

```
!                A: PROC(BASE);
!                   DCL (BASE,X) DESCRIPTOR;
!                -----
```

! Erläuterung
!
! Der Parameter BASE und die Variable X werden als Basisdeskriptoren deklariert. Die Deklaration von Basisdeskriptoren kann auch mittels PLS in Subsystemdatenstrukturen (insbesondere im Subsystemcommon) geschehen.

Basisdeskriptoren müssen vor ihrer ersten Verwendung initialisiert werden. Dies geschieht durch Zuweisung des Wertes der PL/1-Builtin-Function NULL.

! Beispiel

```
!           DCL A DESCRIPTOR INIT(NULL());
!           DCL 1 B (N),
!               2 C DESCRIPTOR INIT((N)NULL()),
!               2 D DESCRIPTOR;
!           D = NULL();
```

! Erläuterung
!
! Der Deskriptor A und das Deskriptorfeld C werden durch das INIT-Attribut der Deklaration initialisiert. Das Deskriptorfeld D wird durch Zuweisung initialisiert.

Amerkung: Ab REGENT-Version 1.3 wird die Initialisierung der Basisdeskriptoren vom PLR-Precompiler im Normalfall gewährleistet. Lediglich für Felder mit storage class BASED ist der Subsystemersteller verantwortlich.
Wenn eine dynamische Datenstruktur aufgebaut wurde, muß ihr Basisdeskriptor vor Verlassen des obersten Blocks, in dem er deklariert wurde, wieder ausdrücklich freigegeben werden (RESET).

! Beispiel

```
!                A: PROC(BASE);
!                   DCL (BASE,X) DESCRIPTOR;
!                   -----
!                   -----
!                   RESET X;
!                END A;
```

! Erläuterung
!
! Der Basisdeskriptor X muß vor Verlassen von A freigegeben werden, nicht jedoch BASE, da BASE von einer anderen Prozedur als Parameter übernommen wurde. Im Gegensatz zu ICES, wo ein vergessenes DESTROY in solchen Fällen katastrophale Folgen hat, bedeutet das Weglassen von RESET lediglich eine geringe Einbuße an Speicherplatz.

4.3.3 Deklaration von dynamischen Datenelementen

Bei Verwendung von DDS speichert der Anwender die Informationen in dynamischen Datenelementen (DDE), die über Basisdeskriptoren zugänglich sind. Die Zuordnung (Assoziierung) zwischen einer bestimmten Art von DDE und einem bestimmten Basisdeskriptor kann bei der Deklaration des DDE geschehen.

! Beispiel

```
!          DCL (A,B) DESCRIPTOR;
!          DCL UA DYNAMIC(A);
!          DCL 1 PUNKT DYNAMIC;
!                2 FARBE BIN FIXED(15),
!                2 X(3);
!          DCL (UB, Y(10)) DYNAMIC(B);
```

! Erläuterung
!
! Die DDE UA und UB sind bei jeder Verwendung (sofern sie nicht anders qualifiziert werden) mit den Basisdeskriptoren A bzw. B assoziiert, ebenso das Feld Y stets mit B. Dagegen sind DDEs vom Typ PUNKT mit keinem Basisdeskriptor assoziiert. Sie müssen daher bei jeder Verwendung explizit einem Basisdeskriptor zugeordnet werden. Hier besteht eine Analogie zu PL/1-Based-Variablen.

4.3.4 Aufbau von dynamischen Datenstrukturen

4.3.4.1 DEFINE-Anweisung

Bevor Daten in dynamischen Datenelementen gespeichert werden können, muß die betreffende dynamische Datenstruktur erst aufgebaut ("definiert") werden. Dies geschieht mit der DEFINE-Anweisung. In ihrer einfachsten Form definiert die DEFINE-Anweisung zu (man kann auch sagen:unterhalb) einem Basisdeskriptor oder einem anderen Deskriptor einen neuen Deskriptorvektor oder einen DDE-Vektor.

Beispiel

```
!                DCL D DESCRIPTOR,(X,INTEGER) DYNAMIC(D), N INIT(3);
!       /*1*/    DEFINE D, (2) DESCRIPTOR);
!       /*2*/    DEFINE D (1), (1) AS (INTEGER);
!       /*3*/    INTEGER (1,1)=N;
!       /*4*/    DEFINE D (2), (INTEGER (1,1)) DESCRIPTOR;
!       /*5*/    DO J=1 TO INTEGER (1,1);
!                DEFINE D (2,J), (J) AS(X);
!                END;
```

Erläuterung

Nach Anweisung 1 existiert zu dem Basisdeskriptor D ein Vektor von 2 Deskriptoren, die als D (1) und D (2) bezeichnet werden. Man sagt, D verweist (oder zeigt) auf einen Deskriptor-Vektor der Länge 2. Nach Anweisung 2 zeigt D (1) auf einen DDE-Vektor der Länge 1. Dieses DDE kann nun als

```
!                D (1,1)        INTEGER
```

bzw., da in der Deklaration INTEGER mit D assoziiert wurde, verkürzt als

```
!                INTEGER (1,1)
```

verwendet werden. Siehe hierzu Anweisungen 3,4 und 5. Nach der Schleife 5 verweist der Deskriptor D (2) auf einen Deskriptorvektor, dessen einzelne Deskriptoren nach der Schleife (5, 6, 7) auf unterschiedlich lange DDE-Vektoren zeigen. Diese DDE können nun als

```
!                D (2,I,J)      X oder    X (2,I,J)
```

! verwendet werden, wobei allerdings J nicht größer als I sein
! darf.

Soll ein Deskriptorvektor definiert werden, zu dessen Deskriptoren anschließend jeweils gleichartige Teildatenstrukturen definiert werden sollen, so kann dies verkürzt geschrieben werden. Statt

```
DEFINE D (2), (5) DESCRIPTOR;
DO J=1 TO 5;
DEFINE D (2,J), (7) DESCRIPTOR;
END;
```

kann geschrieben werden:

```
DEFINE D (2), (5,7) DESCRIPTOR;
```

Dasselbe Prinzip gilt auch für mehrere Hierarchieebenen. Sollen auf der untersten Ebene statt Deskriptorvektoren DDE-Vektoren definiert werden, so ist die entsprechende AS-Option anzufügen.

! Beispiel
!
```
DEFINE D, (2,5,7) AS (INTEGER);
```

! Erläuterung
!
! Damit wird eine DDS angelegt, die logisch einem dreidimensionalen
! Feld der Form
!
```
DCL INTEGER (2,5,7)
```
!
! entspricht, jedoch die Speichervorteile einer DDS besitzt.

Sofern ein Deskriptor vor einer entsprechenden DEFINE-Anweisung bereits auf einen Vektor zeigt, so bewirkt das DEFINE zunächst ein DESTROY und anschließend das DEFINE.

```
!  Beispiel
!            DEFINE D, (5,7) AS (INTEGER);
!            DEFINE D (4), (3) DESCRIPTOR;

!  Erläuterung
!
!  Obige Anweisungsfolge bewirkt dasselbe, wie wenn zwischen den
!  beiden Anweisungen
!            DESTROY D (4);
!  stehen würde.
```

In den oben geschilderten Fällen zeigt ein Deskriptor nach einer Definition stets auf einen Vektor fester Länge; der größte Index dieses Vektors ist in der DEFINE-Anweisung angegeben. Wird innerhalb dieses Vektors ein Deskriptor bzw. Datenelement mit einem größeren als dem maximalen benutzt, so führt dies (ebenso wie ein Index von 0 oder weniger) zum Abbruch des Programms mit entsprechender Fehlermeldung. Es besteht jedoch die Möglichkeit, die Benutzung größerer Indizes zu erlauben. Dazu dient die STEP-Option.

```
!  Beispiel
!            DEFINE D (5), (7) AS (INTEGER) STEP (N);
```

Während ohne die Angabe STEP (N) die Zuweisung

```
            INTEGER (5,19) = 3;
```

ein Fehler wäre, bewirkt diese Zuweisung nun, daß der DDE-Vektor INTEGER (5,1) bis INTEGER (5,7) verlängert wird. Dies geschieht in definierten Schritten, deren Größe vom Wert N gemäß folgender Tabelle abhängt:

N	Schrittweite
<1	0
1	1
2 bis 5	5
6 bis 10	10

11 bis 20	20
21 bis 50	50
51 bis 100	100
>100	200

Die Verlängerung des Vektors wird so lange durchgeführt, bis die betreffende Zuweisung ausgeführt werden kann. Im Fall N = 9 wäre demnach die Schrittweite = 10; der Vektor hätte nach der Zuweisung eine Gesamtlänge von 27. Es muß jedoch beachtet werden, daß von den Werten INTEGER (5,1) bis INTEGER (5,27) nach obigen Anweisungen nur der Wert INTEGER (5,19) einen wolhldefinierten Wert hat (nämlich 3), alle anderen sind undefiniert.

Die Verlängerung eines Vektors geschieht bei jedem Zugriff auf Elemente jenseits der aktuellen Länge eines Vektors. Im obigen Beispiel wird daher bei

```
I = INTEGER (5,38);
```

der DDE-Vektor auf die Länge 47 verlängert. Der gelieferte Wert INTEGER (5,38) ist allerdings undefiniert; die weitere Rechnung wird daher fehlerhaft.

Wenn eine DDS durch eine DEFINE-Anweisung um mehr als eine Hierarchie-Ebene erweitert wird, so gilt die STEP-Anweisung für alle diese Ebenen. (Anmerkung: Der mit ICES vertraute Anwender wird sich erinnern, daß dort die STEP-Option nur die letzte dieser Ebenen betraf.)

```
!  Beispiel
!           DEFINE D, (5,5,5) AS (INTEGER) STEP (10);


! Erläuterung

! Die DDS D entspricht einem Feld der Art
!              DCL INTEGER (5,5,5);
! ,wobei jedoch alle drei Dimensionen bei Bedarf in Schritten von
! 10 wachsen können.
```

In der DEFINE-Anweisung kann zusätzlich eine Prioritäts-Option angegeben werden, die für die logische Datenstruktur unerheblich ist, die aber die Effektivität der Datenverwaltung durch das REGENT-System beeinflußt. Diese Option lautet entweder

HIGH oder LOW oder PRIORITY (p)

,wobei p ein Bit-Ausdruck ist, der bei '1'B HIGH entspricht und bei '0'B LOW. Standardwert ist LOW.

Für den Fall, daß mehrere DDS ganz oder teilweise dieselbe logische Struktur haben sollen, ist es nicht erforderlich, die entsprechenden Anweisungen zu wiederholen. Die Definition wird durch die LIKE-Option vereinfacht.

```
! Beispiel
!              DEFINE A, ---,
!              DEFINE A (--), --;
!              etc. Definition der dynamischen Datenstruktur A
!              DEFINE B, LIKE A;
!              DEFINE C, (2) DESCRIPTOR;
!              DO J=1 TO 2;
!              DEFINE C(J), LIKE A (2);
!              END;

! Erläuterung
!
! Die DDS B erhält dieselbe Struktur wie A. Die DDS C erhält zwei
! Teilstrukturen, die beide wie die Teilstruktur von A (2) aufge-
! baut sind.
```

4.3.4.2 REDEFINE-Anweisung

Die REDEFINE-Anweisung dient dazu,

- die Länge eines Deskriptor- oder DDE-Vektors auf die angegebene Länge oder die Länge modulo(aktuelle Länge,Schrittweite) (falls 0, dann Schrittweite) zu reduzieren,

- die Schrittweite für Verlängerung des Vektors neu festzusetzen (diese neue Schrittweite dient auch einer evtl. Längenänderung) oder

- die Prioritätsangaben für die REGENT-Datenverwaltung des Vektors zu ändern.

Diese Funktionen können in einer Anweisung einzeln auftreten oder kombiniert werden. Ist die Schrittweite 0 und keine neue Länge explizit angegeben, so wird die Länge nicht verändert.

```
! Beispiel
!           DEFINE D, (10) AS (INTEGER) STEP (10);
!           INTEGER (90) = 100;
!           REDEFINE D, REDUCE HIGH RELEASED;

! Erläuterung
!
! Der DDE-Vektor INTEGER wird mit der Länge 10 angelegt. Er wird
! durch die Zuweisung auf 90 verlängert. Außerdem wird automatisch
! der Prioritätszustand von LOW RELEASED auf LOW UNRELEASED gesetzt.
! Die REDEFINE-Anweisung reduziert den Vektor auf die Länge 10,

! (da modulo (90,10) = 0) und den Prioritätszustand auf HIGH RELEASED.
! Der eben bei INTEGER (90) eingespeicherte Wert ist damit natürlich
! verloren.
```

4.3.4.3 CONNECT-Anweisung

Es kann in manchen Anwendungen zweckmäßig sein, in verschiedenen DDS die identisch gleichen Teilstrukturen benutzen zu können. Eine Analogie hierzu bietet in PL/1 die Möglichkeit, Strukturen vom Typ BASED als Maske auf andere Strukturen aufzuprägen.

! Beispiel

```
!           DCL 1 STR1,
!                 2 TEIL1, 3 -----,
!                 2 TEIL2, 3 X(1000), 3 -----;
!           DCL 1 STR2 BASED(P),
!                 2 X(1000), 2 -----;
!           P = ADDR(STR1.TEIL2);
```

! Erläuterung

!

! Nach der letzten Anweisung enthalten STR1.TEIL2 und STR2 die identisch gleichen Daten. Eine Zuweisung STR2.X(503)=1.3; ist in ihrer Wirkung identisch mit STR1.TEIL2.X(503)=1.3;

Bei Verwendung von DDS kann derselbe Effekt mittels der CONNECT-Anweisung erreicht werden.

```
DCL (STR1,STR2) DESCRIPTOR;
DCL X DYNAMIC;
```

! Beispiel

```
!           DEFINE STR1, (2,2) DESCRIPTOR;
!           DEFINE STR1 (2,1), (1000) AS (X);
!           CONNECT STR1 (2,1), STR2;
```

! Erläuterung

!

! Nach der letzten Anweisung ist eine Zuweisung STR2(503) X=1.3;
! in ihrer Wirkung identisch mit STR1(2,1,503) X=1.3; Die Teilstruktur, auf die der Basisdeskriptor STR2 zeigt, ist dieselbe
! wie die, auf die der Deskriptor STR1(2,1) zeigt.

Zur Auflösung einer derartigen Beziehung dient die DESTROY-Anweisung.

4.3.4.4 SWITCH-Anweisung

Die SWITCH-Anweisung erlaubt den Austausch von DDS oder TDS.

```
!  Beispiel
!                SWITCH D1(4.3), D2(6,7,8,9);
```

```
!  Erläuterung
!
!  Die Teildatenstrukturen, auf die die Deskriptoren D1(4,3) und
!  D2(6,7,8,9) zeigten, werden durch diese Anweisung vertauscht.
```

4.3.4.5 COPY-Anweisung

Die COPY-Anweisung erzeugt eine in Struktur, Werten und Priorität gleiche Kopie der ersten angegeben DDS oder TDS. Der in der INTO-Option angegebene Deskriptor zeigt dann auf diese Kopie. Falls er zuvor auf eine DDS oder TDS zeigte, wird diese durch ein DESTROY beseitigt.

```
!  Beispiel
!                  COPY D1 INTO (D2(3,4));
```

```
!  Erläuterung
!
!  D2(3,4) zeigt nach der Anweisung auf eine dynamische Datenstruk-
!  tur, die eine Kopie der DDS von D1 darstellt.
```

4.3.4.6 DESTROY-Anweisung

Eine dynamische Datenstruktur kann, nachdem sie aufgebaut wurde, vom Anwender durch die DESTROY-Anweisung ganz oder teilweise wieder abgebaut werden.

```
!  Beispiel
!                  DEFINE D, (2) DESCRIPTOR;
!                  DEFINE D(1), (3,4) AS (INTEGER);
!                  DEFINE D(2), (1) AS (INTEGER);
!                  DESTROY D(1,3);
```

! Erläuterung
!
! Die DESTROY-Anweisung macht das in der zweiten DEFINE-Anweisung
! implizit enthaltene

```
DEFINE D(1,3), (4) AS (INTEGER);
```

! rückgängig. Die Werte INTEGER (1,3,1) bis INTEGER (1,3,4) existie-
! ren anschließend nicht mehr.

! Beispiel

```
DEFINE wie oben
DESTROY D;
```

! Erläuterung
!
! Die gesamte DDS D wird vernichtet.
! Anmerkung: Ein solches DESTROY ersetzt nicht das vor Verlassen
! des Blocks erforderliche RESET. Wohl aber impliziert das RESET
! ein DESTROY.

Falls zwei DDS durch CONNECT dieselbe TDS enthalten, wird bei DESTROY dafür gesorgt, daß die gemeinsame TDS nicht abgebaut wird.

! Beispiel

```
DEFINE D, (2) DESCRIPTOR;
DEFINE D(2), (4,4,4) AS (INTEGER);
CONNECT D(1), D(2,4,3);
DESTROY D(2,4);
```

! Erläuterung
!
! Nach der CONNECT-Anweisung sind die Werte D(2,4,3,1) bis D(2,4,3,4)
! identisch mit den Werten D(1,1) bis D(1,4). Nach der DESTROY-An-
! weisung ist die gesamte TDS, auf die der Deskriptor D(2,4) zeigt,
! nicht mehr vorhanden. Insbesondere existieren auch D(2,4,3,1)
! bis D(2,4,3,4) nicht mehr; dagegen sind D(1,1) bis D(1,4) noch
! vorhanden.

Es sei ausdrücklich nochmals auf den Unterschied zwischen COPY und CONNECT hingewiesen. COPY erzeugt eine Kopie einer DDS oder TDS; während bei CONNECT einunddieselbe DDS oder TDS lediglich unter zwei verschiedenen Bezeichnungen (und mit unterschiedlicher Zugriffseffektivität) angesprochen wird.

4.3.5 Verwendung dynamischer Datenstrukturen

4.3.5.1 Deskriptoren

Deskriptoren können in den in Kap.4.3.4 und Kap. 4.4.10 beschriebenen PLR-Anweisungen auftreten. Ferner können Deskriptoren als formale bzw. aktuelle Parameter einer Prozedur bzw. eines Unterprogrammaufrufs auftreten. An anderen Stellen ist die Verwendung von Deskriptoren nicht erlaubt.

Soll in einer Prozedur ein Deskriptor als Parameter aus der aufrufenden Prozedur übernommen werden, so muß er als solcher deklariert sein.

! Beispiel

```
!            SUB: PROC (N,D) RETURNS (CHAR(8));
!            DCL N BIN FIXED, D DESCRIPTOR;
!            ---
!            ---
!            END SUB;
```

! Erläuterung
!
! In der Prozedur SUB kann D uneingeschränkt als Basisdeskriptor
! behandelt werden.

Wenn eine Prozedur aufgerufen werden soll, so ist zu unterscheiden, ob sie in der rufenden Prozedur als

- interne Prozedur bekannt ist, oder
- als ENTRY EXTERNAL REGENT oder
- als ENTRY DYNAMIC

deklariert ist.

Im ersten Fall ist lediglich beim Aufruf der aktuell zu übergebende Deskriptor beim Aufruf in die Argumentenliste einzutragen.

```
!  Beispiel
!           DCL X DESCRIPTOR;
!           DEFINE ----;
!             SUB: PROC (N,D) RETURNS (CHAR(8));
!                  DCL N BIN FIXED, D DESCRIPTOR;
!                  -----
!             END SUB;
!           PUT SKIP LIST (SUB(1,X));
!           PUT SKIP LIST (SUB(6,X(2,1,7));
```

```
!  Erläuterung
!
!  An die Funktionsprozedur wird zunächst die ganze DDS X übergeben;
!  die zurückgelieferte Zeichenkette wird gedruckt. Dann geschieht
!  dasselbe mit der TDS auf die der Deskriptor X(2,1,7) zeigt.
```

Falls jedoch die zu rufende Prozedur als EXTERNAL REGENT oder DYNAMIC deklariert ist, muß die ENTRY-Deklaration die Parameterliste gültig vereinbaren.

```
!  Beispiel
!           DCL SUB ENTRY EXT REGENT RETURNS (CHAR(8));
!           DCL TEXT CHAR(8);
!      TEXT = SUB (ooooooooooooo11B, X(2,1,7));
```

```
!  Erläuterung
!
!  Das Weglassen der sogenannten parameter-descriptor-list ist wie
!  in PL/1 zwar zulässig, es wird jedoch dringend davon abgeraten,
!  da dann evtl. erforderliche Datenkonversionen beim Aufruf nicht
!  ausgeführt werden und Fehler schlecht zu finden sind.
```

```
!  Beispiel
!           DCL SUB ENTRY (BIN FIXED(15), DESCRIPTOR)
!                     EXTERNAL REGENT RETURNS (CHAR(8));
!           oder
!           DCL SUB ENTRY (BIN FIXED(15), DESCRIPTOR)
!                     DYNAMIC RETURNS (CHAR(8));
!           DCL TEXT CHAR(8);
!           TEXT = SUB (3,X(2,1,7));
```

! Erläuterung
!
! Dies ist die vollständige und sicherste Deklaration.

```
!  Beispiel
!          DCL SUB ENTRY (BIN FIXED(15),)
!                   EXTERNAL REGENT RETURNS (CHAR(8));
!          oder
!          DCL SUB ENTRY (BIN FIXED(15),)
!                   DYNAMIC RETURNS (CHAR(8));
```

! Erläuterung
!
! Auch diese Form ist zulässig, bietet aber weniger Sicherheit.

4.3.5.2 Dynamische Datenelemente

Grundsätzlich wurde bei der Entwicklung des REGENT-Systems darauf geachtet, daß dynamische Datenelemente in derselben Weise und an denselben Stellen verwendet werden können wie PL/1-Daten gleichen Typs. Es gibt jedoch Ausnahmen, die teils aus prinzipiellen Gründen, teils aus Gründen der Effektivität bei der Übersetzung, teils aus Gründen der raschen Fertigstellung des REGENT-Systems gegenwärtig noch beachtet werden müssen. Dynamische Datenelemente sind an einigen Stellen nicht zugelassen (siehe Kap. 10, dynamic element reference). Die einfachste Form der Verwendung von DDE ist die einer Verallgemeinerung von PL/1-Feldern.

```
!  Beispiel
!          Anstelle von
!                DCL A(N,M,K);
!          kann geschrieben werden
!                DCL A_BASE DESCRIPTOR;
!                DCL A DYNAMIC(A_BASE);
!                DEFINE A_BASE,(N,M,K) AS(A) STEP(100);
```

! Erläuterung
!
! Die DDS A hat gegenüber dem PL/1-Feld A die Vorteile, daß nur
! die aktuell benötigten Daten den Arbeitsspeicher belegen und daß
! das "Feld" sich automatisch bei Bedarf vergrößert. In der Anwen-
! dung wird in beiden Fällen geschrieben
! A (I,J,L) .

Es ist daher einfach, in PL/1-Programmen bei einzelnen Datenfeldern, die auf DDS umgestellt werden sollen, lediglich durch Änderung der Deklaration die Umstellung zu bewerkstelligen.

Überhaupt ist die Verwendung von DDE besonders einfach, wenn in der Deklaration des DDE in der DYNAMIC-Option ein Basisdeskriptor angegeben ist ("assoziierter Basisdeskriptor"). Dann gelten folgende Regeln für das Auffinden des DDE:

- Als Basisdeskriptor gilt der assoziierte Basisdeskriptor
- Die erste in dem DDE-Ausdruck auftretende Indexliste gibt an, der wievielte Deskriptor aus den hierarchisch angeordneten Deskriptorvektoren (bzw. beim letzten Index: DDE-Vektor) benutzt werden soll, um das DDE aufzufinden.

```
!  Beispiel
!           DCL 1 FELD(10) DYNAMIC (BASE),
!                 2 WERTE(50),
!                 2 DATEN(3),
!                   3 (X,Y,Z);
```

! Erläuterung
!
! Die Bezeichnungen
! FELD(5,J,2). DATEN(6,2)
! FELD.DATEN(5,J,2) (6,2)
! FELD(5,J,2) (6). DATEN(2)
! DATEN (5,J,2) (6,2)
!
! referieren alle dieselbe aus drei Werten X,Y,Z bestehende Sub-
! struktur, deren ausführliche Bezeichnung lautet:
!
! BASE(5,J,2) -> FELD(6). DATEN(2).

Sofern in der Deklaration eines DDE kein Basisdeskriptor als assoziiert angegeben ist, oder falls ein DDE im Zusammenhang mit einem anderen als dem assoziierten Basisdeskriptor benutzt werden soll, so muß das DDE voll qualifiziert werden. Die Qualifikation besteht aus dem Basisdeskriptor, der Indexliste und dem Zeichen .

! Beispiel

```
!        DCL (B1,B2) DESCRIPTOR;
!        DCL 1 FELD DYNAMIC(B1),
!            2 N, 2 X(10);
!        DCL DATEN(20) DYNAMIC;
!
!          B2 (4,3,2) -> FELD.N
!          B2 (4,3,2) -> X
!          B1 (3,4)  ->  DATEN(2)
!          B2 (3,6)  ->  DATEN
```

! Erläuterung
!
! Dies sind alles gültige und voll qualifizierte DDE-Referenzen.
! Dagegen wäre X(6)(6) identisch mit B1(6)-> X(6) und ebenfalls
! gültig. Ein Fehler wäre jedoch DATEN(3), da kein Basisdeskriptor
! hierfür angegeben oder assoziiert ist.

4.3.6 Automatische Verwaltung von dynamischen Datenstrukturen

Das REGENT-System verwaltet automatisch die dynamischen Datenstrukturen. Es sorgt dafür, daß dynamische Datenelemente, auf die zugegriffen wird, im Arbeitsspeicher verfügbar sind, während sie zu anderen Zeiten von der Datenverwaltung auf externe Speichermedien (z.B.Platten) ausgelagert werden können. Die Datenverwaltung kann aber nicht vorausschauend wissen, welche Daten bald und häufig benötigt werden und welche nicht. Dem Subsystementwickler ist daher die Möglichkeit gegeben, die Verwaltungsstrategie der Datenverwaltung zu beeinflussen.

Die Datenverwaltung richtet sich nach einer jedem Deskriptorvektor und jedem DDE-Vektor zugeordneten Aktualitätskennziffer. Diese wird durch folgende Vorgänge festgelegt bzw. verändert:

- Angabe einer Priorität HIGH oder LOW bei DEFINE oder REDEFINE
- Zugriff auf einen Deskriptor bzw. ein DDE des betreffenden Vektors. Dies versetzt den Vektor in den Status UNRELEASED, und zwar alle Deskriptorvektoren, die beim Zugriff durchlaufen werden.
- Angabe der RELEASED-Option bei REDEFINE. Dies versetzt den Vektor aus dem Status UNRELEASED zurück in den Status RELEASED, und zwar zur genau dem angegebenen Deskriptorvektor.
- Einfrieren des Vektors im Arbeitsspeicher durch die Anweisung FIX (erzeugt Status FIXED)
- Rückgängigmachen des FIXED-Status durch LOOSE.

Die Vorteile der DDS hinsichtlich dynamischer Struktur und virtueller Speicherung müssen durch entsprechenden Aufwand erkauft werden.

Hieraus ergeben sich für den Subsystemersteller folgende Empfehlungen für effektive Programmgestaltung:

- DDS sollten nur dort benutzt werden, wo die PL/1-Datentypen AUTOMATIC und BASED nicht ausreichen.

- Besonders effektiv wird die Ausführung des Programms, wenn DDE-Vektoren, auf die in einem Programmteil sehr häufig zugegriffen wird, mit der Anweisung FIX "eingefroren" werden und wenn auf diese Vektoren mit BASED-Variablen zugegriffen wird. Allerdings müssen die Vektoren möglichst bald durch LOOSE wieder freigegeben werden, um die Gefahr der Arbeitsspeicherüberfüllung zu reduzieren.

- Deskriptor-Vektoren und DDE-Vektoren, die oft benutzt werden, sollten die Priorität HIGH erhalten. Wenn ihre Benutzungshäufigkeit zurückgeht, sollten sie die Priorität LOW erhalten.

- Deskriptor-Vektoren und DDE-Vektoren, die über längere Zeit nicht mehr benötigt werden, sollten durch REDEFINE den Status RELEASED zugewiesen bekommen.

- Es ist dabei zu berücksichtigen, daß ein Zugriff auf einen Deskriptor oder ein DDE auch einen Zugriff auf alle Deskriptovektoren bedeutet, die auf dem Weg zu diesem Deskriptor bzw. DDE durchlaufen werden.

Die Aktualitätskennziffer hat folgenden Wert:

	Priorität	Status	
0	LOW	RELEASED	LOOSE
1	HIGH	RELEASED	LOOSE
2	LOW	UNRELEASED	LOOSE
3	HIGH	UNRELEASED	LOOSE
4	LOW	RELEASED	FIXED
5	HIGH	RELEASED	FIXED
6	LOW	UNRELEASED	FIXED
7	HIGH	UNRELEASED	FIXED

Die Datenverwaltung sorgt dafür, daß Deskriptor- und DDE-Vektor mit einer Aktualitätskennziffer von 4 oder höher (also alle FIXED) weder auf externe Speichermedien ausgelagert noch im Arbeitsspeicher verschoben werden. Im übrigen versucht die Datenverwaltung stets Daten mit höherer Aktualitätskennziffer möglichst lange im Arbeitsspeicher zu halten. Wenn Platz für neue Daten benötigt wird oder wenn Daten, die auf externem Speicher liegen, benutzt und daher in den Arbeitsspeicher gebracht werden müssen, so werden also zunächst - falls der freie Platz nicht reicht - Daten mit der Aktualitätskennziffer 0 ausgelagert usw. Derartige Umspeicherungen (Reorganisationen) werden von REGENT automatisch durchgeführt, können jedoch auch vom Benutzer zusätzlich global beeinflußt werden. Dazu dient die REORG-Anweisung.

Die REORG-Anweisung definiert für die Datenverwaltung folgende Merkmale:

- in der SIZE-Option: diejenige Größe des Arbeitsspeichers in Bytes, die von der Datenverwaltung mit DDS belegt werden darf. Bei Überschreiten dieser Größe werden Reorganisationen ausgelöst.

- in der LEVEL-Option: diejenige Aktualität, die höchstens noch in die Auslagerung auf externe Dateien einbezogen werden darf. Dies ist normalerweise LOW UNRELEASED, jedoch kann auch LOW RELEASED, HIGH RELEASED oder HIGH UNRELEASED angegeben werden. LOOSE ist in allen diesen Fällen selbstverständlich und wird daher nicht angegeben.

Darüberhinaus kann die Reorganisation mit Hilfe des Utility-Programms QQDREO (siehe Kapitel 10) gesteuert werden. Eine Reorganisation kann durch QQDREO auf Blätter beschränkt werden bzw. auch auf die im Einzelfall unbedingt erforderliche Bytezahl (im Gegensatz zur gesamten Prioritätsgruppe).

Die Programmiersprache PLR stellt dem Benutzer die ATTRIBUTE-Anweisung zur Verfügung, mit der er sich über den gegenwärtigen Zustand eines Deskriptorvektors oder DDE-Vektors informieren kann. Mit dieser Anweisung kann der Benutzer feststellen:

- ob es sich um einen Deskriptor-Vektor oder einen DDE-Vektor handelt,
- wie lang der Vektor gegenwärtig ist,
- wie groß die Schrittweite für Verlängerungen ist und
- welche Aktualitätskennziffer er besitzt.

4.3.7 Programmierte Verwaltung von dynamischen Datenelementen

In Programmbereichen, in denen es auf höchste Ausführungseffektivität ankommt, ist es zweckmäßig, DDE-Vektoren im Arbeitsspeicher "einzufrieren" und mit BASED-Variablen auf sie zuzugreifen. Man erspart sich dadurch bei jedem Zugriff alle von der REGENT-Datenverwaltung durchgeführten Überprüfungen, die der Sicherheit und Flexibilität dienen, man bezahlt dafür mit einem größeren Risiko, daß bei falscher Programmierung schwer lokalisierbare Fehler auftreten.
Für diese Art des raschen Zugriffs dient das Anweisungspaar FIX-LOOSE.

```
!  Beispiel
!           DCL A DESCRIPTOR;
!           DCL X DYNAMIC(A);
!           DEFINE A,(2,10) AS(X) STEP(10),
!           REDEFINE A(1), HIGH;
!           ON ENDFILE(SYSIN) GO TO END1;
!           J=0;
!           DO WHILE ('1'B)/*UNTIL ENDFILE(SYSIN)*/;
!           GET LIST(X(1,J+1));
!           J=J+1)
!           END;
!           END1:
!             REDEFINE A(2), HIGH;
!  /*1*/      DO I=1 TO J;
!  /*2*/      X(J-I+1,J) = X(1,J);
!  /*3*/      END;
!  /*4*/      DO I = 1 TO J;
!  /*5*/      X(2,J) = SQRT(ABS(X(2,J)));
!  /*6*/      END;
!             DO I=1 TO 2;
```

```
!            REDEFINE A(I), RELEASED STEP(0), LOW;
!            END;
!            REDEFINE A, RELEASED;
```

! Erläuterung
!
! Es werden Daten nach X(1,J+1) gelesen. Dieser DDE-Vektor hat die
! Priorität HIGH. Vor dem Umspeichern nach X(2,J) in umgekehrter
! Reihenfolge erhält auch dieser Vektor hohe Priorität. Jeweils
! nach der letzten Benutzung der Vektoren in diesem Programmteil
! erhalten sie die niedrigste Aktualität (LOW RELEASED) und außer-
! dem die Schrittweite 0. Da auch auf den Deskriptorvektor, auf den
! A zeigt, zugegriffen wurde, wird auch dieser am Schluß RELEASED.

Im folgenden Beispiel wird gezeigt, wie der im obigen Beispiel auf den Label END1 folgende Teil des Programms durch Verwendung von FIX und LOOSE mit höchster Effektivität programmiert werden kann.

! Beispiel

```
!            DCL Y1(32000) BASED(X1);
!            DCL Y2(32000) BASED(X2);
!            FIX A(1) SET(X1) ON(1) READ;
!            FIX A(2) SET(X2) ON(1);
!  /*1*/     DO I=1 TO J;
!  /*2*/     Y2(J-I+1) = Y1(I);
!  /*3*/     END;
!  /*4*/     DO I=1 TO J;
!  /*5*/     Y2(I) = SQRT(ABS(Y2(I)));
!  /*6*/     END;
!            DO I=1 TO 2;
!            LOOSE A(I);
!            END;
```

! Erläuterung
!
! Dies ist die effektivste Art des Zugriffs auf DDE-Vektoren.

4.4 REGENT-Dateiverwaltung

4.4.1 Allgemeines

Die REGENT-Datei-Verwaltung dient vor allem der langfristigen Verwaltung großer Datenmengen, d.h. der Übergabe von großen Datenmengen von einem REGENT-Job an einen anderen. Es ist eine bekannte Tatsache, daß bei den meisten Anwendungen der EDV im Entwicklungs- und Konstruktionsbereich eine derartige Datenübergabe von ausschlaggebender Bedeutung ist.
Ein besonderes Merkmal der REGENT-Datei-Verwaltung ist es, daß sie es erlaubt, die ganze zu einer bestimmten Organisationseinheit gehörenden Datenmenge baumartig in Teilmengen zu unterteilen und diese einzeln zu verwalten.

! Beispiel
!
! - Gesamtdatenmenge eines Projekts (BANK)
! - Teildatenmenge allgemeine Information (CATALOG)
! - Teildatenmenge Zeichnungen (CATALOG)
! - Prinzipskizze
! - Einzelteilskizze
! - Fertigungszeichnungen (CATALOG)
! - Teil 1
! - Teil 2
! - --
! - Teildatenmenge Festigkeit (CATALOG)
! - Lastfall 1 statisch (CATALOG)
! - Lastannahmen
! - Verformungen
! - Spannung
! - Lastfall 2 dynamisch (CATALOG)
! - Lastverlauf
! - Verformungen (CATALOG)
! - Zeitpunkt 1
! - Zeitpunkt 2
! - ---

Eine derartige Gesamtdatenmenge wird im REGENT-System als Datenbank, kurz: BANK, bezeichnet. Jede Teildatenmenge wird als POOL bezeichnet. Dabei wird unterschieden,

- ob die Teildatenmenge selbst aus Teildatenmengen besteht; dann ist sie ein "Katalog" und wird durch CATALOG gekennzeichnet,
- oder ob die Teildatenmenge nicht mehr aus Teildatenmengen besteht; dann ist sie ein "Segment" und kann (muß nicht) durch DATA besonders gekennzeichnet werden.

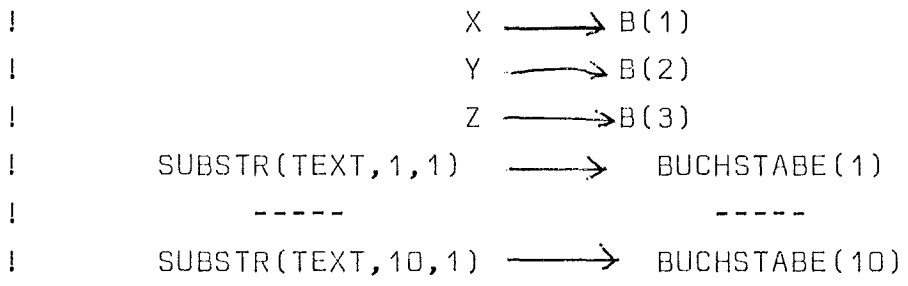
Ein Segment hat selbst keine Unterteilung. Es enthält lediglich diejenigen Daten, die durch STORE POOL-Anweisungen in das Segment geschrieben wurden, genau in der Reihenfolge, in der sie geschrieben wurden. In dieser Reihenfolge werden die Daten normalerweise auch wieder gelesen. Dabei muß jedoch keine Satzstruktur eingehalten werden (wie bei normaler RECORD-Ein/Ausgabe in PL/1), vielmehr ist lediglich auf die Reihenfolge und den jeweils richtigen Datentyp der zu lesenden Daten zu achten (wie bei STREAM-Ein/Ausgabe in PL/1). Ein "Zeiger" gibt bei sequentiellem Lesen oder Schreiben an, an welcher Stelle des Segments die folgende STORE- oder RETRIEVE-Anweisung wirksam wird. Der Abstand dieser Stelle vom Beginn des Segments wird in Bytes gemessen und heißt Offset.

! Beispiel

```
!          DCL 1 A, 2(X,Y,Z), 3 TEXT CHAR(10);
!          SET POOL(POOL)NEXT(0);
!          STORE POOL(POOL) FROM(A);
!          -----
!          DCL B(3), BUCHSTABE(10) CHAR(1);
!          SET POOL(POOL)NEXT(0);
!          DO J=1 TO 3;
!          RETRIEVE POOL(POOL) INTO (B(J));
!          -----
!          END;
!          DO J=1 TO 10;
!          RETRIEVE POOL(POOL) INTO (BUCHSTABE(J));
!          -----
!          END;
```

Erläuterung

Obiges Beispiel bewirkt folgende Zuweisungen:



Durch besondere - in den folgenden Kapiteln beschriebene - Anweisungen kann jedoch auch abweichend von diesem sequentiellen Verfahren geschrieben oder gelesen werden.

Mehrere Segmente können in einem Katalog zusammengefaßt sein; auch mehrere Kataloge oder auch Segmente und Kataloge gemeinsam können in einem Katalog und letztlich in der Datenbank zusammengefaßt sein. In diesem Zusammenhang kann eine BANK auch als Katalog verstanden werden. Das Einrichten neuer POOLs (Kataloge oder Segmente) geschieht durch ein Initialisieren (INITIATE-Anweisung). Wenn bestehende POOLs benutzt werden sollen, müssen sie zunächst eröffnet werden (OPEN POOL). Jeweils nach Benutzung sind die POOLs zu schließen. Gleiches gilt auch für die BANK.

Im Sinne des Betriebssystems entspricht eine BANK einem Dataset,im Sinne von PL/1 entspricht sie einem FILE. Die Zuordnung zwischen BANK und Dataset geschieht über folgende Beziehungen:

Betriebssystemebene:		Dataset
Zuordnung	:	DD-Karte der Job Control Sprache
PL/1-Ebene	:	FILE-Konstante im Main-Modul
		FILE-Variable in PLR-Programmen
Zuordnung	:	BANK-Deklaration mit oder ohne
		FILE-Option
REGENT-Ebene	:	BANK

4.4.2 Datenschutz

Datenschutz ist immer unter den zwei Aspekten zu sehen:

- Schutz gegen Verlust,
- Schutz gegen unbefugte Benutzer.

Innerhalb des REGENT-Systems sind dem Datenschutz Grenzen gesetzt durch die Tatsache, daß das REGENT-System grundsätzlich Mängel des Betriebssystems in dieser Hinsicht nicht beheben kann. Da jede BANK sich dem Betriebssystem als Dataset darstellt, ist prinzipiell die Möglichkeit gegeben, daß ein unbefugter Benutzer diesen Dataset zerstört oder liest, indem er andere Mittel als die der REGENT-Dateiverwaltung benutzt. Gegen beide Arten der unbefugten Benutzung helfen nur administrative Maßnahmen wie z.B. die Anfertigung von Sicherheitskopien auf Band bzw. das Geheimhalten der inneren Struktur der Segmente in der BANK.

Anders ist die Situation, wenn der Benutzer mit den Mitteln der REGENT-Dateiverwaltung eine BANK benutzt. Für diese (normalen) Fälle sieht REGENT eine besondere Datenschutztechnik auf der Basis von Schlüsselworten bzw. Paßworten vor. Jeder BANK und jedem POOL sind zwei Gruppen zu je 4 Schlüsselworten zugeordnet. Jedes Schlüsselwort der ersten Gruppe (Schlüsselwort 1 bis 4) entspricht einer bestimmten Berechtigung, die BANK bzw. den POOL zu benutzen. Jedes Schlüsselwort der zweiten Gruppe (5 bis 8) beduetet, daß die entsprechende Berechtigung nicht nur auf die betreffende BANK bzw.den POOL zutrifft, sondern auch auf alle darin enthaltenen POOLs. Diese zweite Gruppe ist die Gruppe der "privilegierten" Schlüsselworte.

Schlüsselworte-Berechtigungen

nicht privilegiert		privilegiert	
1	I (INPUT)	5	IP (INPUT PRIV)
2	O (OUTPUT)	6	OP (OUTPUT PRIV)
3	U (UPDATE)	7	UP (UPDATE PRIV)
4	C (CHANGE PASSWORD)	8	CP (CHANGE PASSWORD PRIV)

Diese Schlüsselworte können auch alle oder teilweise gleich sein, was bedeutet, daß nicht besonders zwischen verschiedenen Berechtigungen unterschieden wird. Außerdem schließen bestimmte Berechtigungen automatisch andere ein. So berechtigt zusätzlich

```
R   zu nichts als Lesen
W   zu I
U   zu O,I
C   zu U,O,I
RP  zu I
WP  zu IP,O,I
UP  zu OP,IP,U,O,I
CP  zu UP,OP,IP,C,U,O,I
```

Schließlich gibt es die Möglichkeit, bei der Neu-Eröffnung einer BANK oder eines POOL festzulegen, daß alle oder einige Berechtigungen jedem Benutzer ohne Datenschutz gegeben sein sollen. Dies wird durch NOPASSWORD festgelegt.

```
!  Beispiel
!              INITIATE BANK(BANK1) NOPASS;
!
!  Erläuterung
!
!  Es wird eine BANK eingerichtet, zu deren Benutzung keinerlei
!  Berechtigung nachgewiesen werden muß.

!  Beispiel
!              INITIATE POOL(P) PASSWORD (ALL='MYNAME',
!                       U,O = 'PUT',I = 'GET')
!
!  Erläuterung
!
!  Es wird zunächst für alle Schlüsselworte 'MYNAME' eingetragen,
!  dann wird dies für U und O geändert in 'PUT' bzw. für I in 'GET'.
!  Wenn bei späterer Eröffnung das Paßwort 'MYNAME' angegeben ist,
!  so ist alles außer U,O,I erlaubt. Wird jedoch als Paßwort nur
!  'GET' angegeben, so wird nur die Berechtigung I erworben, bei ei-
!  nem Paßwort 'PUT' die Berechtigung U und O (und damit auch I).
!  Der "Eigentümer" des POOL(P) kann also Anderen die lesende Be-
```

! nutzung des POOL erlauben, indem er ihnen das Paßwort 'GET'
! mitteilt, er kann ihnen auch ein Ändern der Daten erlauben, in-
! dem er ihnen das Paßwort 'PUT' mitteilt.

Jede bereits existierende BANK und jeder existierende POOL muß vor einer Benutzung mit einer OPEN BANK- bzw. OPEN POOL-Anweisung eröffnet werden. Dabei kann jeweils ein Paßwort angegeben werden. Die REGENT-Dateiverwaltung vergleicht das angegebene Paßwort mit den Schlüsselworten der BANK bzw. des POOL (sofern diese nicht mit NOPASSWORD angelegt sind). Aufgrund dieses Vergleichs wird festgestellt, welche Berechtigungen für diese Benutzung von BANK bzw.POOL bestehen, d.h. welche Anweisungen anschließend durchgeführt werden dürfen. Wird nach dem Öffnen eine Anweisung gegeben, zu der keine Berechtigung besteht, so ist dies ein Fehler. Sofern der Benutzer als Paßwort ein privilegiertes Schlüsselwort angegeben hat, überträgt sich die Berechtigung automatisch auf alle untergeordneten POOLs.

Der Datenschutz wird dadurch wirksam, daß eine Operation (OPEN,STORE, RETRIEVE), für die der Benutzer seine Berechtigung nicht nachgewiesen hat, nicht ausgeführt wird, sondern eine Fehlermeldung erzeugt. Man beachte: Das Öffnen einer BANK ist stets möglich, auch wenn keines der gültigen Schlüsselworte für die BANK angegeben wurde. Der Fehler tritt erst ein, wenn dann in der BANK ein POOL eröffnet werden soll.

Abgesehen von den Passworterfordernissen sind die Operationen INITIATE POOL, OPEN POOL, STORE und RETRIEVE in einem Pool nur dann erlaubt, wenn der betreffende Pool mit dem richtigen Mode (INPUT, OUTPUT, UPDATE) eröffnet wurde. Es gelten folgende Regeln:

In einem Pool bzw.einer Bank, die eröffnet wurden mit	sind erlaubt die Operationen
INPUT	RETRIEVE, RETRIEVE FROM SET OPEN OLD INPUT KEEP RELEASE
OUTPUT	RETRIEVE, RETRIEVE FROM STORE OPEN OLD INPUT KEEP RELEASE OPEN MOD OUTPUT KEEP RELEASE INITIATE DATA CATALOG OUTPUT
UPDATE	RETRIEVE, RETRIEVE FROM STORE, STORE TO SET OPEN OLD INPUT OUTPUT UPDATE KEEP RELEASE DELETE OPEN MOD OUTPUT KEEP RELEASE DELETE INITIATE DATA CATALOG OUTPUT UPDATE

Durch diese Regeln wird sichergestellt, daß eine eingeschränkte Berechtigung, die ein Benutzer für einen Pool erworben hat, in Pools, die in diesem Pool enthalten sind, mindestens gleichermaßen eingeschränkt bleibt.

Generell ist festzustellen:

INPUT	erlaubt die Benutzung bestehender Information und schließt jede Veränderung und Zerstörung ebenso aus wie das Hinzufügen neuer Information.
OUTPUT	erlaubt die Benutzung bestehender Information sowie das Hinzufügen neuer Information, schließt jedoch die Veränderung und Zerstörung bestehender oder neu hinzugefügter Information aus.
UPDATE	erlaubt die Benutzung bestehender Information, das Hinzufügen neuer Information und das Verändern und Zerstören von Information.

4.4.3 Deklaration von BANK und POOL

In PLR-Programmen, die eine BANK benutzen, muß diese deklariert sein. Eine BANK darf in PLR-Programmen nur als Parameter oder Variable deklariert werden.

! Beispiel

```
!             DCL  B1  BANK;
```

! Erläuterung

! Die Bank B1 wird deklariert; sie muß in diesem Fall als Parameter übergeben werden.

! Beispiel

```
!             DCL ... B(2) BANK VARIABLE INIT(B1,B1);
```

! Erläuterung

! Die Variablen B(1) und B(2), die Teil einer STRUCTURE sein können, werden deklariert und mit B1 initialisiert.

Desgleichen muß auch jeder POOL, der im PLR-Programm benutzt wird, vereinbart sein.

```
!  Beispiel
!                DCL  P(3) POOL;
!
!  Erläuterung
!
!  Es wird ein Feld von 3 Pools deklariert.
```

4.4.4 INITIATE BANK

Das Einrichten einer neuen Bank geschieht mit der INITIATE BANK Anweisung. Mit dieser Anweisung wird festgelegt:

- welcher Datenschutz für die BANK gelten soll,
- wieviel Pools in der Bank angelegt werden dürfen,
- welche Verwaltungslisten für die BANK geführt werden sollen.

4.4.4.1 Datenschutz

Sie Kapitel 4.4.2.

4.4.4.2 Platzangabe

Die Anweisung kann eine Option

SPACE(n)

enthalten. Der Wert von n gibt an, wieviele Pools in der Bank angelegt werden können. Dabei handelt es sich allerdings nur um die Maximalzahl von Pools auf der obersten Hierarchiestufe der Bank. Innerhalb eines jeden dieser Pools kann eine beliebige Anzahl von Pools angelegt werden. Der Standardwert für n ist 16.

Die Anweisung enthält keine Option, die den maximal für die Datenspeicherung verfügbaren Platz auf dem Datenträger angibt. Das REGENT-System holt sich diese Information aus dem Betriebssystem der Rechenanlage.

4.4.4.3 Verwaltungslisten

Die REGENT-Dateiverwaltung führt zwei Verwaltungslisten, in denen

- der in der BANK verfügbare Platz (FREELIST) und
- der früher belegte, aber wieder freigegebene Platz (DELETELIST)

vermerkt wird. Die Funktion der FREELIST ist unmittelbar einleuchtend. Die DELETELIST bedarf einer Erläuterung:

Wenn ein POOL belegt und wieder freigegeben wird (letzteres geschieht beim CLOSE nach einem OPEN DELETE), so wird der dadurch wieder verfügbar werdende Platz normalerweise nicht sofort wieder für Neubelegung eingesetzt, sondern zur Vermeidung von Speicherfragmentierung zunächst in der FREELIST reserviert gehalten, bis entweder diese Liste voll ist oder ohne Verwendung der in dieser Liste vermerkten Platzbereiche eine Platzanforderung (beim INITIATE POOL) nicht befriedigt werden könnte. Die Maximalzahl der Einträge in die FREELIST wird durch die Option

DELETELIST (Anzahl)

in der INITIATE-BANK-Anwiesung festgelegt. Die Option

NODELETELIST

bedeutet dagegen, daß freigegebener Platz sofort in die FREELIST eingetragen wird.

Entsprechend gibt die Option

FREELIST (Anzahl)

an, wieviele Einträge in der FREELIST gemacht werden können. Bei Überschreiten dieser Zahl führt die REGENT-Dateiverwaltung eine Speicherreorganisation durch.

Beide Anzahl-Angaben sind als BINARY FIXED(31) Ausdrücke aufzufassen und haben - falls die Option fehlt - die Standardwerte 16 für DELETELIST und 64 für FREELIST.

In der Option LISTPAGE wird angegeben, wieviel Platz(in Zahl der Einträge) jeweils in einer Seite jedes Katalogs in der Bank vorgesehen werden soll. Sofern nicht sehr lange Kataloge (also viele Einträge pro Katalog) zu erwarten sind, wird der Standardwert von 16 empfohlen.

! Beispiel

```
!           INITIATE BANK(B(2)) PASSWORD (ALL='MYPASS')
!                DELETELIST(64);
```

! Erläuterung

! Es wird auf dem Datenträger, der durch die BANK B(2) referiert
! wird, eine Bank initialisiert. Für alle Operationen mit dieser
! Bank wird später das Passwort 'MYPASS' benötigt. Die DELETELIST
! kann 64 Einträge aufnehmen, ebenso die FREELIST (Standardwert 64).

4.4.5 OPEN BANK

Eine BANK muß, nachdem sie initialisiert und evtl. bereits benutzt worden ist, für jede Benutzung durch eine OPEN-BANK-Anweisung eröffnet werden. Diese Anweisung kann folgende Optionen enthalten:

- PASSWORD siehe Kap.4.4.2
- BUFNO (Anzahl der Puffer)
- mode (INPUT oder OUTPUT oder UPDATE)

Für die Anzahl der Puffer ist ein Standardwert von 2 vorgesehen. Bei einer BANK, in der sehr stark random zugegriffen wird, empfiehlt sich die Angabe einer größeren Pufferzahl, sofern dadurch nicht zuviel Arbeitsspeicher belegt wird. Jeder Puffer ist so lang wie die BLKSIZE-Angabe in den DCB-Parametern der Betriebssystemdatei, welche die BANK enthält.
Für mode gilt INPUT als Standardwert.

! Beispiel

```
!           DCL (B1,B2) BANK;
!           OPEN BANK(B1) BUFNO(4);
!           OPEN BANK(B2) PASSWORD('MYNAME') OUTPUT;
```

! Erläuterung

! Die Bank B1 wird ohne Paßwortangabe eröffnet (sie sei zuvor
! mit NOPASSWORD initialisiert worden). Ihr sind 4 Puffer zuge-
! ordnet. Es sind in ihr nur INPUT-Operationen möglich (s.4.4.2).

```
!  Die Bank B2 wird mit OUTPUT und der Standardpufferzahl eröffnet.
!  In ihr sind OUTPUT und INPUT-Operationen möglich, allerdings
!  nur, soweit das Paßwort 'MYNAME' die Berechtigung erteilt.
!  Wenn z.B. die Initialisierung lautete:
!
!       INITIATE BANK(B2) PASSWORDS (ALL='YOURNAME'
!    INPUT='MYNAME');
!  so sind trotz der Eröffnung von B2 mit OUTPUT nur INPUT-Opera-
!  tionen möglich.
```

4.4.6 INITIATE POOL

Diese Anweisung erlaubt das Anlegen eines neuen Pools in einer bestehenden Bank bzw. einem bestehenden Pool, die zu diesem Zweck mit OUTPUT oder UPDATE und entsprechendem Passwort eröffnet sein müssen. Es kann angegeben werden

- die Schlüsselworte für den Datenschutz,
- der maximale Platz
- der Pooltyp DATA oder CATALOG .

Die Schlüsselwortliste für den Datenschutz entspricht genau der Angabe in der INITIATE BANK-Anweisung (siehe 4.4.2).

Der maximale Platz ebenso wie bei INITIATE BANK (s.4.4.4.2) wird angegeben durch die Option

SPACE(n)

,wobei n bei CATALOG die Maximalzahl der Einträge angibt, bei DATA dagegen die Maximalzahl der zu speichernden Bytes. Der Standardwert ist 16.
Der Pooltyp gibt an, ob es sich um einen Katalog handelt (CATALOG), in dem weitere Pools eröffnet werden können, oder ob in dem Pool Daten (DATA) gespeichert werden sollen. Bei CATALOG können in dem Pool die Operationen OPEN und CLOSE angewandt werden,bei DATA dagegen die Operationen STORE, RETRIEVE und SET. Standardwert ist DATA.

! Beispiel

```
!           DCL B BANK;
!           DCL (P1,P2,P3) POOL;
!           INITIATE POOL(P1) IN (B) CATALOG;
!           OPEN POOL(P1) IN (B) UPDATE;
!           INITIATE POOL(P2) IN (P1) SPACE(10000);
!           INITIATE POOL(P3) IN (P1) CATALOG;
```

! Erläuterung

! In Bank B (die bereits eröffnet sei) wird ein Pool P1 als Katalog angelegt. Dieser Katalog wird anschließend eröffnet, damit darin ein neues Segment P2 von der Länge 10000 Bytes und ein neuer Katalog P3 angelegt werden können.

4.4.7 OPEN POOL OLD bzw. OPEN POOL MOD

Diese Anweisung dient zum Öffnen eines bereits bestehenden Pools. Bei OLD kann der Pool mit INPUT, OUTPUT oder UPDATE eröffnet werden. Bei MOD dagegen nur mit OUTPUT, da MOD bedeutet, daß anschliessend an das Ende der im Pool bereits enthaltenen Daten weitergeschrieben wird. MOD ist auch nur für Pools vom Typ DATA, nicht dagegen für CATALOG sinnvoll und erlaubt. Die Angabe von CATALOG oder DATA entfällt jedoch, da die entsprechende Information bereits der REGENT-Dateiverwaltung bekannt ist. Insgesamt sind bei OPEN OLD bzw. MOD anzugeben

- das Paßwort für die Zugriffsberechtigung
- die Close-Optionen KEEP, RELEASE, DELETE.

Das Paßwort muß angegeben werden, sofern in dem Pool eine Operation vorgenommen werden soll, für welche beim Anlegen des Pools (d.h. beim INITIATE) ein Schlüsselwort festgelegt wurde. Das angegebene Paßwort wird mit allen im Pool eingetragenen Schlüsselworten verglichen. Eine Übereinstimmung überträgt dem Benutzer die jeweilige Benutzungsberechtigung.

Die Close-Option entscheidet, ob beim Schließen des Pools (CLOSE POOL)

- der POOL erhalten bleiben soll (KEEP),
- der POOL in dem Teil, in dem Information eingefügt wurde, erhalten bleiben soll, jedoch etwa noch verfügbarer Platz im Speicherbereich (bei DATA) oder Katalog (bei CATALOG) gelöscht werden soll (RELEASE),
- der POOL und alle in ihm etwa noch angelegten Pools gelöscht werden sollen (DELETE).

Standardwert is KEEP.

! Beispiel

```
!           DCL B BANK, (P1,P2,P3,P4) POOL;
!           OPEN POOL(P1) IN(B) UPDATE;
!           OPEN POOL(P2) IN(P1) OUTPUT;
!           OPEN POOL(P3) IN(P1) UPDATE DELETE;
!           OPEN POOL(P4) IN(P3) INPUT;
!              -----
!              -----
!           CLOSE POOL(P1);
```

! Erläuterung

! In Bank B wird ein Pool P1 mit UPDATE eröffnet. Darin wird ein ! Pool P2 mit OUTPUT eröffnet und ein Pool P3 mit UPDATE, in letzterem ein Pool P4 mit INPUT. Beispielsweise könnte nun der Inhalt von P4 nach P2 übertragen werden. Beim Schließen von P3 ! würden P3 und P4 gelöscht.

4.4.8 CLOSE BANK

Die Anweisung schließt eine Bank und alle darin evtl. noch eröffneten Pools. Die Pools werden je nach Angabe der bei ihrer Eröffnung festgelegten Close-Option erhalten, gelöscht oder in ihrer Länge reduziert.

4.4.9 CLOSE POOL

Die Anweisung schließt einen Pool und alle darin evtl. noch eröffneten Pools. Die Pools werden je nach Angabe der bei ihrer Eröffnung festgelegten Close-Option erhalten, gelöscht oder in ihrer Länge reduziert.

4.4.10 Schreiben und Lesen in einem REGENT-Pool

Ein REGENT-Pool vom Typ DATA ist als eine sequentielle Datei zu verstehen, in der ein Zeiger eine vom Anfang der Datei an gerechnete aktuelle Lese- bzw. Schreibposition angibt. Diese Positionsangabe entspricht der Anzahl von Bytes zwischen Dateianfang und der aktuellen Position, doch sollte von dieser Tatsache möglichst kein Gebrauch gemacht werden; es genügt meist, wenn man diese Position als eine Kennziffer behandelt. Bei jedem sequentiellen Lese- bzw. Schreibbefehl wird der Positionszeiger um den entsprechenden Betrag erhöht. Bei direktem Lesen und Schreiben geschieht dies nicht. Beim Eröffnen eines Pools zeigt der Zeiger auf den Anfang der Datei (bei OPEN OLD) bzw. auf die Position, die unmittelbar auf die zuletzt beschriebene folgt (bei OPEN MOD). Um abweichend vom rein sequentiellen Lesen und Schreiben an beliebiger Position einer Datei einen Lese- bzw. Schreibvorgang beginnen zu können, kann der Positionszeiger durch die SET-Anweisung auf eine beliebige Position in der Datei eingestellt werden. Folgende Datentypen können in Pools gespeichert werden:

- alle primitiven PL/1-Daten (CHAR,BIT,ALIGNED,BIN FLOAT etc., jedoch nicht FILE, ENTRY etc. und BIT UNALIGNED)
- alle PL/1-Felder und Strukturen,
- Datenelemente (DDE) dynamischer Datenstrukturen, die obigen Bedingungen genügen,
- dynamische Datenstrukturen (DDS) und Teildatenstrukturen (TDS).

4.4.10.1 Sequentielles Lesen und Schreiben von Daten

Die Anweisungen

RETRIEVE POOL(pool) TO(variable) [SET(key)];

und

STORE POOL(pool) FROM(variable) [SET(key)];

dienen zum Lesen und Schreiben von Daten (PL/1-Variablen, Feldern bzw. Strukturen von PL/1-Variablen oder dynamischen Datenelementen). Sie sind nur erlaubt, wenn der Pool vom Typ DATA und mit INPUT, OUTPUT oder UPDATE (bei RETRIEVE) bzw. OUTPUT oder UPDATE (bei STORE) sowie mit der erforderlichen Zugriffsberechtigung eröffnet ist.

Dieses sequentielle Lesen und Schreiben unterscheidet sich syntaktisch vom direkten Lesen und Schreiben dadurch, daß bei RETRIEVE keine FROM-Option und bei STORE keine TO-Option angegeben wird. Die SET-Option bewirkt, daß die aktuelle Zeigerposition zu Beginn dieses Vorgangs in der Variablen key gespeichert wird. Diese Variable sollte als BIN FIXED(31) deklariert sein. Nach Beendigung der Operation steht der Zeiger im POOL auf der Position, die auf das letzte übertragene Byte folgt.

! Beispiel

```
!          BEGIN;
!          DCL 1 STRUCT(N),
!                2 POINT,
!                  3 (X,Y,Z),
!                  3 TEXT CHAR(10) VARYING;
!          DO J=1 TO N;
!          RETRIEVE POOL(P(1)) TO (POINT(J));
!          STORE POOL(P(2)) FROM (POINT(J)) SET(K);
!          STORE POOL(P(3)) FROM(K);
!               ------
!               ------
!          END;
```

! Erläuterung

! Aus Pool P(1) werden Strukturen der Art POINT gelesen, in STRUCT
! gespeichert und nach Pool P(2) geschrieben. Die Positionen dieser Strukturen in P(2) werden durch SET(K) gemeldet und in Pool
! P(3) gespeichert.

! Beispiel

```
!           DCL STRUCT DESCRIPTOR;
!           DCL 1 POINT DYNAMIC (STRUCT),
!                   2 (X,Y,Z),
!                   2 TEXT CHAR(10) VARYING;
!           DEFINE STRUCT,(100) AS (POINT) STEP(100);
!           DO J=1 TO N;
!           RETRIEVE POOL(P(1)) TO (POINT(J));
!           STORE POOL(P(2)) FROM (POINT(J)) SET(K);
!           STORE POOL(P(3)) FROM(K);
```

! Erläuterung

! Dieses Beispiel unterscheidet sich vom vorhergehenden nur dadurch, daß anstelle des PL/1-Feldes von POINT-Strukturen eine dynamische Datenstruktur benutzt wird.

4.4.10.2 Direktes Lesen und Schreiben von Daten

Die Anweisungen

RETRIEVE POOL(pool) TO (variable)
FROM(key_1) [NEXT(key_2)] ;

und

STORE POOL(pool) FROM (variable)
TO(key_1) [NEXT(key_2)] ;

dienen zum Lesen und Schreiben in einem Pool an anderer Stelle als an der aktuellen Zeigerposition. Die aktuelle Zeigerposition wird durch diese Operationen nicht verändert. Die SET-Option gibt die Position an, an der der Übertragungsvorgang beginnt; durch die NEXT-Option kann die auf das letzte übertragene Byte folgende Position gemeldet werden. key_1 und key_2 sollten BIN FIXED(31)-Variable sein.

! Beispiel

```
!           DCL 1 POINT,
!                 2 (X,Y,Z),
!                 2 TEXT CHAR(10) VARYING;
!           ATTRIBUTE P(1) ACT(KEYEND);
!           KEY=1;
!           DO WHILE (KEY < KEYEND);
!           RETRIEVE POOL(P(1)) TO (POINT) FROM (KEY) NEXT (KEY);
!           STORE POOL(P(1)) FROM (POINT);
!           END;
```

! Erläuterung
!
! Der Inhalt des im Beispiel von Kap. 4.4.10.1 beschriebenen Pools
! P(1) wird als Kopie nochmals im Anschluß an den bestehenden In-
! halt geschrieben. Hinsichtlich ATTRIBUTE siehe Kap.4.5.2. Die
! Verwendung von KEY sowohl in der FROM- wie auch in der NEXT-
! Option bewirkt praktisch ein sequentielles Lesen.

! Beispiel

```
!          DO J=1 TO 10;
!          RETRIEVE POOL(P(3)) TO(K);
!          END;
!          STORE POOL(P(1)) FROM (POINT) TO(K);
```

! Erläuterung
!
! Es wird wieder das Beispiel aus Kapitel 4.4.10.1 benutzt, unter
! der Annahme, daß Pool P(3) inzwischen geschlossen und neu eröff-
! net wurde. Nach der Schleife enthält K die Position der 10ten
! in Pool(P(2)) stehenden POINT-Struktur. Diese wird mit dem gegen-
! wertigen Wert von POINT überschrieben.

Bei der Benutzung des direkten Schreibens ist Vorsicht geboten. Es muß darauf geachtet werden, daß die neu in den Pool geschriebenen Daten sich von ihrer Struktur her mit den dort bereits stehenden decken.

! Beispiel

```
!           STORE POOL(P(1)) TO (K+1);
!   oder
!           DCL A BIT(50);
!           STORE POOL(P(1)) FROM(A) TO(K);
```

! Erläuterung
!
! In dem in Kap.4.4.10.1 benutzten Pool würde hierdurch Information
! in einer Weise überschrieben, die wahrscheinlich nicht den Ab-
! sichten des Programmierers entspricht.

Ähnliches gilt auch für das Lesen.

! Beispiel

```
!           DCL A CHAR(300);
!           RETRIEVE POOL(P(1)) TO(A) FROM(15);
```

! Erläuterung

! Es wird der Inhalt mehrerer POINT-Strukturen (die erste und
! letzte unvollständig) in die Zeichenkette A gelesen. Der Inhalt
! von A entspricht dann sicher nicht den Absichten des Programmie-
! rers.

4.4.10.3 Lesen und Schreiben dynamischer Datenstrukturen

Die Anweisungen

```
STORE POOL(pool) FROM (d-reference) [DIRECT] [SET(key)] ;
```

und

```
RETRIEVE POOL(pool) TO (d-reference);
```

übertragen ganze dynamische Datenstrukturen (DDS) oder Teildatenstrukturen (TDS). Direktes Lesen und Schreiben ist bei DDS und TDS nicht möglich (kann jedoch durch die SET-Anweisung bei aller gebotenen Vorsicht praktisch erreicht werden). Die Option DIRECT bei STORE ermöglicht späteren indexsequentiellen Zugriff.

! Beispiel

```
!           DCL BASE DESCRIPTOR;
!           DEFINE BASE,(3);
!           DO J=1 TO 3;
!           RETRIEVE POOL(P1) TO (BASE(J));
!           END;
!           STORE POOL(P2) FROM (BASE) DIRECT;
```

! Erläuterung

! Drei DDS werden aus Pool P1 gelesen und als TDS einer DDS
! mit Basisdeskriptor BASE gespeichert. Anschließend wird die
! ganze DDS, die zu BASE gehört, in Pool P2 geschrieben.

4.4.10.4 Direkter Zugriff auf dynamische Datenelemente im Pool

Die Anweisungen

```
STORE POOL(pool) FROM (variable)
     TO (key,indexlist);
```

und

```
RETRIEVE POOL(pool) TO (variable)
     FROM (key,indexlist);
```

erlauben den direkten Zugriff auf Datenelemente von dynamischen Datenstrukturen, die in einem Pool gespeichert sind. Der Positionszeiger im Pool wird dadurch nicht verändert. Hinsichtlich der gebotenen Vorsicht sei auf Kap. 4.4.10.2 verwiesen.

Beispiel

```
DCL BASE DESCRIPTOR;
DCL 1 POINT DYNAMIC(BASE),
    2 (X,Y,Z);
DCL PX LIKE(POINT) AUTOMATIC;
DEFINE BASE, (10,10,10) AS(POINT);
-----
-----
STORE POOL(P) FROM(BASE) DIRECT SET(K);
-----
RETRIEVE POOL(P) FROM(K,4,3,5) TO(PX);
PX=2.*PX;
STORE POOL(P) FROM(PX) TO(K,4,3,5);
```

Erläuterung

Eine DDS bestehend aus einem (logisch) dreidimensionalen Feld von Wertetripeln (x,y,z) wird im Pool P gespeichert. Anschließend wird ein solches Wertetripel wieder gelesen, verändert und zurückgeschrieben.

4.4.5 Verwaltungsbefehle

4.4.5.1 Setzen des Positionszeigers

Die Anweisung

```
SET POOL(pool) NEXT(key);
```

setzt den aktuellen Positionszeiger in einem Pool auf den Wert key. Damit kann an beliebiger Position innerhalb des Pools mit sequentiellem Lesen und Schreiben neu begonnen werden.

```
!  Beispiel
!           DO J=1 TO 10;
!           RETRIEVE POOL(P(3)) TO(K);
!           END;
!           SET POOL(P(2)) NEXT(K);
!           DO J=1 TO 3;
!              RETRIEVE POOL(P(2)) TO(POINT);
!                -----
!                -----
!           END;
!
!  Erläuterung
!
!  Es wird die Position der zehnten POINT-Struktur in P(2) aus P(3)
!  entnommen. Dann werden nacheinander die  zehnte, elfte und zwölfte
!  POINT-Struktur aus P(2) gelesen.
```

Die Vorsicht, die bei direktem Lesen und Schreiben geboten ist, gilt gleichermaßen auch für das Verwenden des sequentiellen Lesens und Schreibens nach einer Neudefinition des Positionszeigers durch SET.

4.4.5.2 Feststellen von Poolmerkmalen

Die ATTRIBUTE-Anweisung erlaubt es, wesentliche Merkmale eines Pools dem Anwendungsprogramm mitzuteilen.

```
TYPE(n)
MAX(n)
ACT(n)
NEXT(n)
SPACE(n1,n2,n3,n4,n5)
```

4.5 Nachrichtenverwaltung

4.5.1 Übersicht

Die Nachrichtenverwaltung erlaubt es, in PLR-Programmen (in gleicher Form wie in jeder problemorientierten Sprache und in Anweisungsdefinitionen von PLS zwischen EXEC und END EXEC)

- standardisierte Ablauf- und Fehlernachrichten zu erzeugen,
- die Ausgabe der Nachrichten in bestimmten Nachrichtenklassen zu unterdrücken oder zuzulassen,
- für Nachrichten jeder einzelnen Klasse im Anschluß an die ausgegebene Nachricht einen mehr oder weniger ausführlichen PLIDUMP (Testhilfe, die von PL/1 zur Verfügung gestellt wird) auszudrucken oder diesen Ausdruck zu unterdrücken,
- die Nachrichten seit Eintritt in das REGENT-System in den einzelnen Klassen zu zählen, die Gesamtzahl oder ein auf Null wieder zurücksetzbares Zählwerk abzufragen und aufgrund der eingetretenen Nachrichtenzahl programmierte Reaktionen auszulösen.

Folgende Nachrichtenklassen, die durch sogenannte Level unterschieden werden, sind in REGENT vorgesehen:

Level	Zweck, Bedeutung
D oder DEBUG	Nachrichten für Testzwecke bei der Subsystementwicklung
I oder INFORMATIVE	Hinweis auf Programmablauf, Modelleinschränkungen, allgemeine Erläuterungen. Dieser Level wird angenommen, wenn keiner spezifiziert ist.
W oder WARNING	Hinweise auf unvollständige oder nicht ganz korrekte Aufgabenstellung, wobei aber mit großer Wahrscheinlichkeit die beabsichtigte Reaktion richtig vermutet werden kann.

E oder ERROR	Fehler. Durch Änderung der Aufgabenstellung kann das System oder das Subsystem jedoch möglicherweise ein noch brauchbares Ergebnis erzeugen.
S oder SEVERE	schwerer Fehler. Das System oder Subsystem kann versuchen, einen Programmabbruch zu verhindern. Ergebnisse sind sicher falsch.
T oder TERMINAL	Irreparabler Fehler. Programmabbruch unmittelbar nach der Erzeugung der Fehlermeldung ist sehr wahrscheinlich.

Ob eine erzeugte Nachricht tatsächlich ausgegeben wird, hängt vom sogenannten Aktivierungsgrad der betreffenden Nachrichtenklasse ab. Der Aktivierungsgrad kann durch die Anweisungen

```
MESSAGE ACTIVE   bzw.
MESSAGE INACTIVE
```

jeweils um 1 erhöht bzw. erniedrigt werden. Wenn eine Nachrichtenklasse einen Aktivierungsgrad von 1 oder mehr besitzt, so werden die Nachrichten dieser Klasse ausgegeben, andernfalls werden sie unterdrückt. Der Ausgangsaktivierungsgrad ist 0 für die Level D und I, bzw. 1 für die übrigen Level.
Die Nachrichtenzählung wird durch den Aktivierungsgrad nicht beeinflußt, d.h. auch Nachrichten, die nicht ausgegeben werden, werden gezählt.

4.5.2 Erzeugung und Ausgabe von Nachrichten

Für die Erzeugung von Nachrichten gibt es ein einheitliches Format mit folgender Grundform

```
MESSAGE level TEXT (text);
```

Auch alle anderen Anweisungen, die zur Nachrichtenverwaltung gehören, fangen mit MESSAGE an. Folgendes Beispiel soll die Benutzung des Aktivierungsgrades erläutern:

! Beispiel
!
! In einem PLR-Programm des Moduls A des Subsystems SSS stehe
! folgende Nachricht:
!

```
!            *SUBSYSTEM   SSS
!            *MODUL  A
!              A: PROC;
!                 ---
!                 ---
!                 MESSAGE ERROR
!                   TEXT ('UNGETESTETE VERSION VOM 15.04.75');
!              -----
!              END A;
```

Es sei angenommen, daß dieses Programm innerhalb des Subsystems SSS an zwei Stellen benutzt werden soll: einmal wird es während der Ausführung einer Anweisung S1 unmittelbar vom Hauptprogramm aufgerufen, zum anderen geschieht während der Ausführung einer Anweisung S2 aus einem Modul B heraus. Im Laufe der Subsystementwicklung sei sichergestellt, daß die Verwendung bei S1 als gesichert angesehen werden kann, während die Anwendung bei S2 noch zu falschen Resultaten führt. Dieser Situation kann folgendermaßen Rechnung getragen werden:
Mit PLS werden die Anweisungen wie folgt definiert:

```
ENTER  PLS;
SUBSYSTEM 'SSS';
STA  'S1';
  -----
  EXECUTE LINK B;;
END STA;
STA  'S2';
  -----
  EXECUTE LINK A;
END STA;
END PLS;
```

Der Modul B wird folgendermaßen erzeugt:

```
*SUBSYSTEM  SSS;
*MODUL  B;
  B: PROC;
     ---
     DCL A ENTRY DYNAMIC;
     MESSAGE INACTIVE ERROR;
     CALL   A;
     MESSAGE  ACTIVE  E;
  END B;
```

Die Wirkung der Aktivierungsanweisungen zeigt sich bei der Verwendung des Subsystems.

```
ENTER   SSS;
```

S1; Modul B würde aufgerufen; Fehler vom Level E würde inaktiviert; Modul A wird von B aufgerufen; die in A erzeugte Nachricht wird nicht gedruckt; danach wird Level E wieder aktiviert.

S2; Modul A wird aufgerufen; die Nachricht in A wird gedruckt.

! Beispiel

```
!           MESSAGE INACTIVE;
!           DO J=1 TO 1000;
!             S2;
!             END;
!           MESSAGE ACTIVE;
```

! Erläuterung
!
! Für Testzwecke soll die Anweisung S2 1000 mal benutzt werden. Um
! unnötige Nachrichten zu unterdrücken, werden zuvor alle Nachrichtenlevel inaktiviert und nach der Schleife wieder aktiviert.

4.5.3 Nachrichtenzählung

Zu jeder Nachrichtenklasse werden von REGENT zwei Zählwerke verwaltet, die beide bei Eintritt in das System auf 0 gesetzt sind. Im einen Zählwerk wird die Gesamtzahl aller erzeugten Nachrichten dieser Klasse addiert. Auf diese Zählung hat der Anwender keinen Einfluß. Das zweite Zählwerk kann vom Anwender auf 0 gesetzt werden.

Beispiel

```
*SUBSYSTEM   SSS;
*MODUL  C;
   C: PROC;
   ---
   DCL  A ENTRY DYNAMIC;
   ---
   MESSAGE INACTIVE W;
   MESSAGE RESET W;
   K=0;
   DO WHILE (----);
   ---
   K=K+1;
   CALL  A;
   END;
   MESSAGE COUNT(K1) W;
   IF K1*2 K THEN
   MESSAGE ERROR
      TEXT ('MODULE A ZU MEHR ALS
      '50 PROZENT NICHT FEHLERFREI');
   MESSAGE ACTIVE W;
```

Erläuterung

Im Modul C wird der Modul A in einer Schleife mehrfach aufgerufen, so lange die unter WHILE(---) angegebene Bedingung erfüllt ist. Die Zahl der Aufrufe von A wird in der Variablen K gezählt.
Der Subsystemersteller erwartet, daß im Modul A gelegentlich oder häufig Nachrichten vom Level W erzeugt werden. Da er sie nicht zum Ausdrucken bringen will, inaktiviert er diesen Level vor der Schleife. Ebenfalls vor der Schleife wird das Nachrichtenzählwerk für W auf 0 gesetzt.
Nach Beendigung der Schleife kann nun der Stand des Zählwerks für W-Nachrichten abgefragt werden. In diesem Fall wird es in K1 übertragen. Anhand dieser Information kann der Subsystemersteller Entscheidungen über den Programmablauf planen. In diesem Fall will er eine Nachricht vom Level E erzeugen, falls bei mehr als der Hälfte der Aufrufe von A eine Nachricht der Klasse W erzeugt worden war. Abschließend wird der ursprüngliche Aktivierungsgrad der Nachrichten vom Level W wiederhergestellt.

4.5.4 Testhilfe durch PLIDUMP bei Nachrichtenausgabe

In der Entwicklungsphase eines Subsystems ist es oft sehr wichtig, den Programmablauf im Detail zu verfolgen. Dazu dient vor allem die Angabe eines TRACE, das heißt die Angabe der gerade aktiven Programme und des Ortes ihres Aufrufs aus anderen Programmen. Diese Information wie auch andere wertvolle Testhilfen können von dem PL/1 verfügbaren Programm PLIDUMP geliefert werden.

Der Subsystemersteller und der Subsystemanwender können festlegen, daß bei Ausgabe einer Nachricht automatisch auch diese Testhilfen ausgegeben werden bzw. unterdrückt werden. Dazu dienen die Anweisungen

```
MESSAGE DUMP ON   bzw.   DUMP OFF
```

(Anmerkung: Im Gegensatz zu ACTIVE und INACTIVE wirken diese Anweisungen unmittelbar, also nicht über einen Aktivierungsgrad. Die Folge

```
MESSAGE DUMP ON;
MESSAGE DUMP OFF;
```

hat also dieselbe Wirkung wie ein einfaches MESSAGE DUMP OFF;)

Die Art der als Dump erzeugten Testhilfeausgabe kann durch Optionen gesteuert werden. Diese Optionen sind eingehend im PL/1-Reference-Manual erläutert.

! Beispiel
!
! Der Subsystemanwender spezifiziere
!

```
!          ENTER   SSS;
!          MESSAGE ACTIVE D W;
!          MESSAGE DUMP OFF;
!          MESSAGE DUMP ON D;
!          ----
!          END;
```

!
! In den PLR-Programmen des Subsystems SSS finden sich Anweisungen
! der Art

```
!          MESSAGE DEBUG;
!          MESSAGE WARNING ---;
!          etc.
```

! Erläuterung
!
! Bei obiger Anwendung des Subsystems SSS werden alle Nachrichten-
! level aktiviert; D und W explizit, die übrigen sind implizit
! aktiv. Dann wird für alle Nachrichtenklassen die Dump-Ausgabe
! ausgeschaltet, was eigentlich nicht nötig wäre, da dies der Aus-
! gangszustand ist. Anschließend wird nur für Level D die Ausgabe
! eines Trace eingeschaltet. An allen Stellen im Subsystem, an de-
! nen MESSAGE DEBUG angegeben ist, wird also ein Trace erzeugt.

4.6 Spezielle Unterprogramme

Dem Subsystemersteller stehen neben den Anweisungen noch folgende Unterprogramme zur Verfügung, die in Kapitel 10 näher erläutert werden:

QDREO :	Wahl der Reorganisationsstrategie
QQDBDD:	Basisdeskriptor-Dump
QQDPOL:	DDS-Poolstatistik
QQDSTX:	DDS-Dump(komplett)
QQDSTR:	DDS-Dump(Teil-Dump)
QQDPSZ:	Ändern des DDS-Poolsize
QQMMAP:	RMM-Statistik
QQMREO:	RMM-Reorganisationsroutine
QQTTIME:	Zeitmessungsroutine (TTIMER-Makro-Anwendung)
QQSTIME:	Zeitmessungsroutine (STIMER-Makro-Anwendung)

4.7 Einschränkungen

4.7.1 CONTROLLED-Variable

CONTROLLED-Variable darf in RMM-Moduln nur benutzt werden, wenn sie als Parameter übergeben wurden.

4.7.2 FILE-Konstanten

FILE-Konstanten dürfen in RMM-Moduln nicht vereinbart werden. Besonders zu beachten ist dies bei impliziter Vereinbarung von SYSIN oder SYSPRINT in PL/1-Prozeduren.

KAPITEL 5

Die Programmiersprache PLS für die Entwicklung problemorientierter Sprachen

5.1 Einleitung

Das Problemsprachen-Übersetzungs- und Definitionssystem PLS (Problem Language System)dient dazu, problemorientierte Sprachen zu definieren und zu übersetzen. Diese Sprachen sind in REGENT Erweiterungen der Basissprache PL/1, so daß Variable, Datenaggregate, Kontrollanweisungen und Prozeduren in allen Anwendersprachen verwendet werden können. Der REGENT-Übersetzer übersetzt POL-Programme in PL/1-Programme, die anschließend von den PL/1-Übersetzern der Rechenanlage kompiliert werden.

Das POL-Definitionssystem von PLS ist selbst ein REGENT-Subsystem. Es erlaubt das Initieren und Löschen von Subsystemen, die Definition und das Löschen von POL-Anweisungen für bestimmte Subsysteme und die Spezifikation von Subsystem-Datenstrukturen. Spracherweiterungen und Datenstrukturen werden in Bibliotheken gespeichert, die später vom REGENT-Übersetzer bei der Subsystem-Anwendung benutzt werden (siehe Abb.5.1).

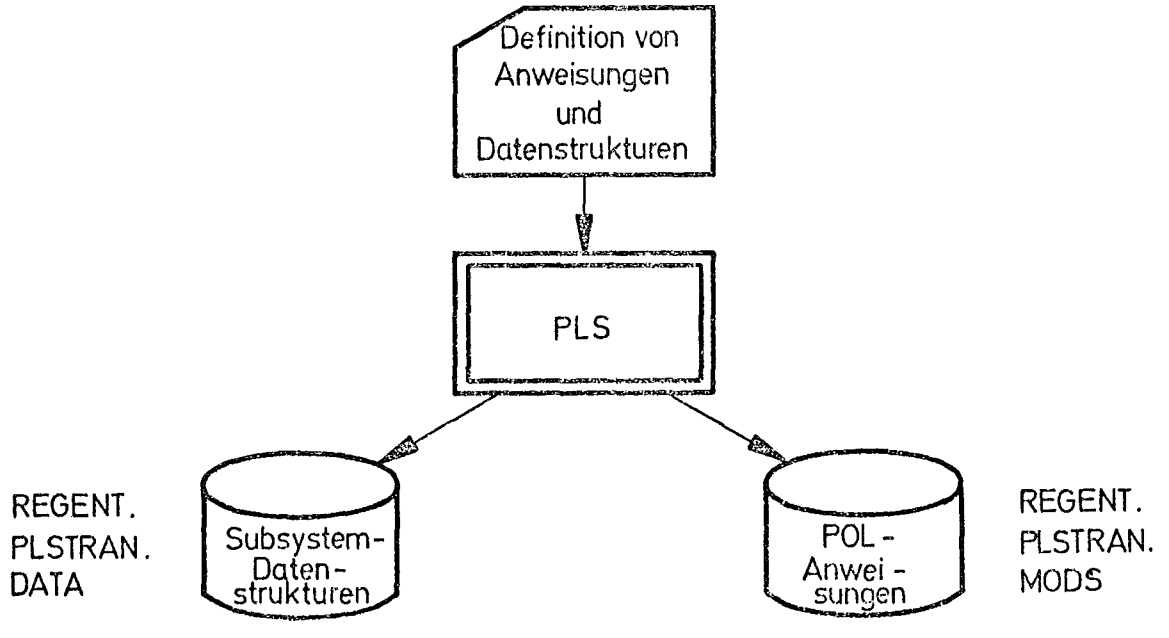


Abb. 5.1: Definition von POL-Anweisungen

Die Anwendung von PLS geschieht in gleicher Weise wie bei anderen REGENT-Subsystemen auch. Es wird aufgerufen durch

```
ENTER PLS;
```

und abgeschlossen durch

```
END PLS;
```

Da PLS keine Dynamic Arrays benötigt, kann zur Einsparung von Speicherplatz auf der PROCEDURE-Anweisung des Hauptprogramms die REGENT-Option NODA verwendet werden.

```
! Beispiel:
!
! DEFINE: PROC OPTIONS(MAIN) REGENT(NODA);
!   ENTER PLS;
!     POL-Definitionen
!   END PLS;
! END DEFINE;
```

In den folgenden Abschnitten werden alle PLS-Anweisungen im Zusammenhang beschrieben, eine alphabetische Beschreibung aller Anweisungen befindet sich im PLS-Handbuch (Kapitel 11).

5.2 Handhaben von Subsystemen

Mittels PLS-Anweisungen können Subsysteme initialisiert und gelöscht werden, Anweisungs- und Datendefinitionen können zu existierenden Subsystemen hinzugefügt oder gelöscht werden, die Anweisungen, Datendeklarationen und Module eines Subsystems können aufgelistet werden. Dieser Abschnitt befaßt sich mit PLS-Anweisungen, die ganze Subsysteme betreffen.

Initialisierung von Subsystemen

Wenn in das REGENT-System ein neues Subsystem eingebracht werden soll, muß der Name dieses Subsystems dem System mitgeteilt werden. Dazu dient die INITIATE-Anweisung. Es wird der Name des Subsystems und

ein Schlüsselwort angegeben. Dieses Schlüsselwort muß stets angegeben werden, wenn das Subsystem erweitert, geändert oder gelöscht werden soll. Die Anweisung hat die Form:

INITIATE SUBSYSTEM name KEY key ;

name ist eine Zeichenkette mit maximal 32 Zeichen, die den Namen des Subsystems darstellt. name darf kein Leerzeichen enthalten. Soll der Name des Subsystems abkürzbar sein, ist nach den signifikanten ersten Buchstaben ein . in den Namen einzufügen. Soll ein Punkt Bestandteil des Namens sein, müssen 2 Punkte geschrieben werden. Beispiel: 'NAM.E', das Subsystem heißt NAME, es kann zu NAM abgekürzt werden. 'S..1.22', das Subsystem heißt S.122 und kann S.1 abgekürzt werden.

key ist eine Zeichenkette von maximal 32 Zeichen und bezeichnet den Schlüssel für das Subsystem. key darf keine Leerzeichen enthalten. Defaultwert für key, falls nicht angegeben, ist der Nullstring, ''.

```
! Beispiel
!
! INITIATE SUBSYSTEM 'GRA.PHIC' KEY 'GRAKEY';
```

Ist ein Subsystem mit gleichem Namen und gleichem Schlüssel schon vorhanden, so wird zuerst das alte Subsystem zerstört und anschließend neu initialisiert. Stimmt dagegen nur der alte Name und nicht der Schlüssel, erfolgt eine Fehlermeldung und keine Initialisierung.

Zuordnung von Definitionen zu einem Subsystem

Sollen zu einem Subsystem neue Anweisungen definiert, Datenstrukturen deklariert oder Anweisungen oder Datendeklarationen gelöscht werden, so muß der Subsystemname und das Schlüsselwort des betreffenden Subsystems angegeben werden. Dazu dient die SUBSYSTEM-Anweisung. Alle POL- und Datenstruktur-Definitionen bis zur nächsten SUBSYSTEM-Anweisung beziehen sich auf das angegebene Subsystem.

```
SUBSYSTEM  name  KEY  key  ;
```

Dabei ist name der Subsystemname, ausgeschrieben oder abgekürzt. key ist der Schlüssel für das Subsystem.

```
! Beispiel
!
! SUBSYSTEM 'SUB1' KEY 'KEY1';
```

Löschen von Subsystemen

Subsysteme werden durch die DESTROY-SUBSYSTEM-Anweisung gelöscht.

```
DESTROY SUBSYSTEM name KEY key;
```

Das Subsystem mit dem Namen name wird nur dann gelöscht, wenn der richtige Schlüssel key angegeben wird.

```
! Beispiel
!
! DESTROY SUBSYSTEM 'SUB1' KEY 'KEY1';
```

Listen von Subsystemen

Mit der LIST-Anweisung lassen sich die REGENT-Subsysteme, die gültigen Anweisungen und Datendeklarationen eines Subsystems und die auf Bibliotheken befindlichen Members eines Subsystems alphabetisch auflisten.

Die Anweisung

```
LIST  SUBSYSTEMS;
```

druckt alle im System befindlichen Subsysteme aus. Durch die Anweisung

```
LIST STATEMENTS OF SUBSYSTEM name;
```

werden alle gültigen Anweisungen und Datendeklarationen eines Subsystems ausgedruckt. Das Listen aller auf den Systembibliotheken befindlichen Teilen eines Subsystems wird durch die Anweisung

```
LIST MEMBERS OF SUBSYSTEM name;
```

erreicht. Die Membernamen der Subsystemmodule und die Membernamen der Anweisungsdefinitionen und Datendeklarationen werden alphabetisch aufgeführt.

```
! Beispiel
!
!    ENTER PLS;
!      LIST SUBSYSTEMS;
!      LIST STATEMENTS OF SUBSYSTEM 'SUB1';
!      LIST MEMBERS OF SUBSYSTEM 'SUB1';
!    END PLS;
```

5.3 Anweisungen

5.3.1 Anweisungsdefinitionen

Damit ein Subsystem angewandt werden kann, werden problemorientierte Anweisungen (POL-Anweisungen) für das Subsystem definiert.

Der erste Schritt der Entwicklung einer Subsystemsprache ist die Spezifikation der Anweisungen, die erforderlich sind, um die Eingabeparameter für das Subsystem zu beschreiben und den Ablauf der Rechnung zu steuern. Während die Algorithmen in einer für Arithmetik geeigneten Sprache programmiert wurden (z.B. FORTRAN oder PL/1), muß die Sprache zur Anwendung dieser Algorithmen die geometrischen und physikalischen Eigenschaften des Modells beschreiben können, das die Problemstellung wiedergibt. Die Anweisungen sollen problemnah und einprägsam sein.

```
! Beispiel: Die POL-Anweisung habe folgende Syntax:
!
!                    { UEBERSCHRIFT 'text' }
!          DRUCKE    { E_MODUL             },
!                    { ALLE [WERTE]        }
!
! Die Anweisungsdefinition steht zwischen
!          STATEMENT  'DRU.CKE';
! und
!          END STATEMENT;
```

```
! Nach STATEMENT wird der Name der Anweisung angegeben. Soll der
! Name abkürzbar sein, wird dies durch einen Punkt nach den signi-
! fikanten Buchstaben des Anweisungsnamens angezeigt. Die DRUCKE-
! Anweisung kann also zu "DRU" abgekürzt werden.
```

POL-Anweisungen werden durch den REGENT-Übersetzer in PL/1-Anweisungen umgesetzt. In einer Anweisungsdefinition muß also nicht nur die Syntax der POL-Anweisung festgelegt werden, sondern es muß auch angegeben werden, in welcher Weise die betreffende POL-Anweisung nach PL/1 übersetzt wird. Dazu hat er außer allen PL/1-Anweisungen spezielle PLS-Anweisungen und PLS-Funktionen zur Verfügung. Diese gestatten ihm, nacheinander die Grundelemente der POL-Anweisung zu gewinnen und danach die Übersetzung in PL/1-Anweisungen zu steuern. Entsprechend den zur Verfügung stehenden Subsystemmoduln und deren Steuerparameter muß die Umsetzung von Sprachanweisungen in Aufrufe dieser Module geplant werden.

```
! Für das Beispiel der DRUCKE-Anweisung sollen folgende PL/1-
! Anweisungen erzeugt werden:
!
! Für  DRUCKE UEBERSCHRIFT:
!      PUT LIST('text') SKIP;
!
! Für  DRUCKE E_MODUL:
!      PUT LIST ('E_MODULE=',emodul) SKIP;
!
! Für  DRUCKE ALLE WERTE:
!      Aufruf des Moduls PUTALL
!
! In der Anweisungsdefinition muß zuerst festgestellt werden, ob
! 'UEB', 'E-M' oder 'ALLE' als nächstes in der POL-Anweisung vor-
! handen ist. Im Falle 'UEB' muß der Text 'text' gewonnen werden,
! im Falle 'ALLE' muß das Füllwort 'WERTE' übergangen werden und
! anschließend müssen die Anweisungen zur Realisierung der Bedeu-
! tung der POL-Anweisung (also hier Anweisungen zum Drucken von
! Werten) erzeugt werden. Das könnte wie folgt aussehen:
!
```

```
!       STAT 'DRU.CKE';
!       IF BIDENTIFIER('UEB') THEN DO;
!          EXEC PUT LIST (NEXT_STRING) SKIP ;;
!          END;
!       ELSE IF BWORD('E-M') THEN DO;
!          EXECUTE PUT LIST('E-MODULE=', E_MODULE) SKIP;;
!          END;
!       ELSE IF IDENTIFIER('ALLE') THEN DO;
!          SKIP ID('WER');
!          EXEC LINK PUTALL;;
!          END;
!       ELSE CALL ERROR(1100);
!       END STATEMENT;
```

Diese Anweisungsdefinition enthält neben PL/1-Anweisungen folgende PLS-Anweisungen:

EXECUTE	Generiere PL/1-Anweisung(en),
SKIPB ID('WER')	Übergehe den Identifier, der mit den Buchstaben 'WER' beginnt (daher SKIPB), falls er als nächstes in der Eingabe steht,
LINK	rufe ein Modul dynamisch auf.

Außerdem werden PLS-Funktionen verwendet:

BIDENTIFIER	Stelle fest, ob als nächstes in der Eingabe ein Identifier vorhanden ist, der mit der angegebenen Zeichenkette beginnt (B in BIDENTIFIER steht für BEGIN),
BWORD	stelle fest, ob die nächsten in der Eingabe stehenden Zeichen (außer Leerzeichen) mit den angegebenen Zeichen übereinstimmen,
IDENTIFIER	stelle fest, ob in der Eingabe der angegebene Identifier steht.

Alle diese Funktionen liefern als Funktionswert in einer BIT(1)-Variablen eine Ja/nein-Antwort zurück und setzen außerdem im Falle von "ja" den Eingabezeiger um das gefundene Anweisungs-Element weiter.

NEXT_STRING	liefert die nächste Zeichenkette.

Im Normalfall beginnt eine POL-Anweisung mit einem Schlüsselwort, so wie auch z.B. alle PL/1-Anweisungen außer der Zuweisung und der Nullanweisung mit einem Schlüsselwort beginnen.
Es ist auch möglich, Anweisungen zu definieren, die nicht mit einem Schlüsselwort, sondern mit einem bestimmten Datentyp beginnen.

Syntax

STATEMENT DATATYPE type ;

Die möglichen Datentypen sind:

Bitstring	- "type" = BITSTRING	Bitkettenkonstante
Identifier	- "type" = IDENTIFIER	Dies sind Benennungen (Namen)
Integerzahl	- "type" = INTEGER	Alle BINARY oder DECIMAL FIXED-Konstanten
Operator	- "type" = OPERATOR	Alle gültigen PL/1-Operatoren +,-,/,*,||,|,&,>, usw.
Realzahl	- "type" = REAL	Alle BINARY oder DECIMAL FLOAT-Konstanten
Characterstring	- "type" = STRING	Alle Zeichenketten-Konstanten

```
! Beispiel
!
!      STATEMENT DATATYPE REAL;
!         /* Verarbeite Real-Zahlen */
!      END STATEMENT;
!      STA D STRING;
!         /* Verarbeite Zeichenketten */
!      END STA;
!
! Die so definierten Anweisungen könnten z.B. folgende POL-State-
! ments verarbeiten:
!
!      1.2;  3.0+4.1-3.7;
!      'STRING 1';  'STRING 2';
```

Den Ablauf des Erkennens von Anweisungen zeigt Abb. 5.2.

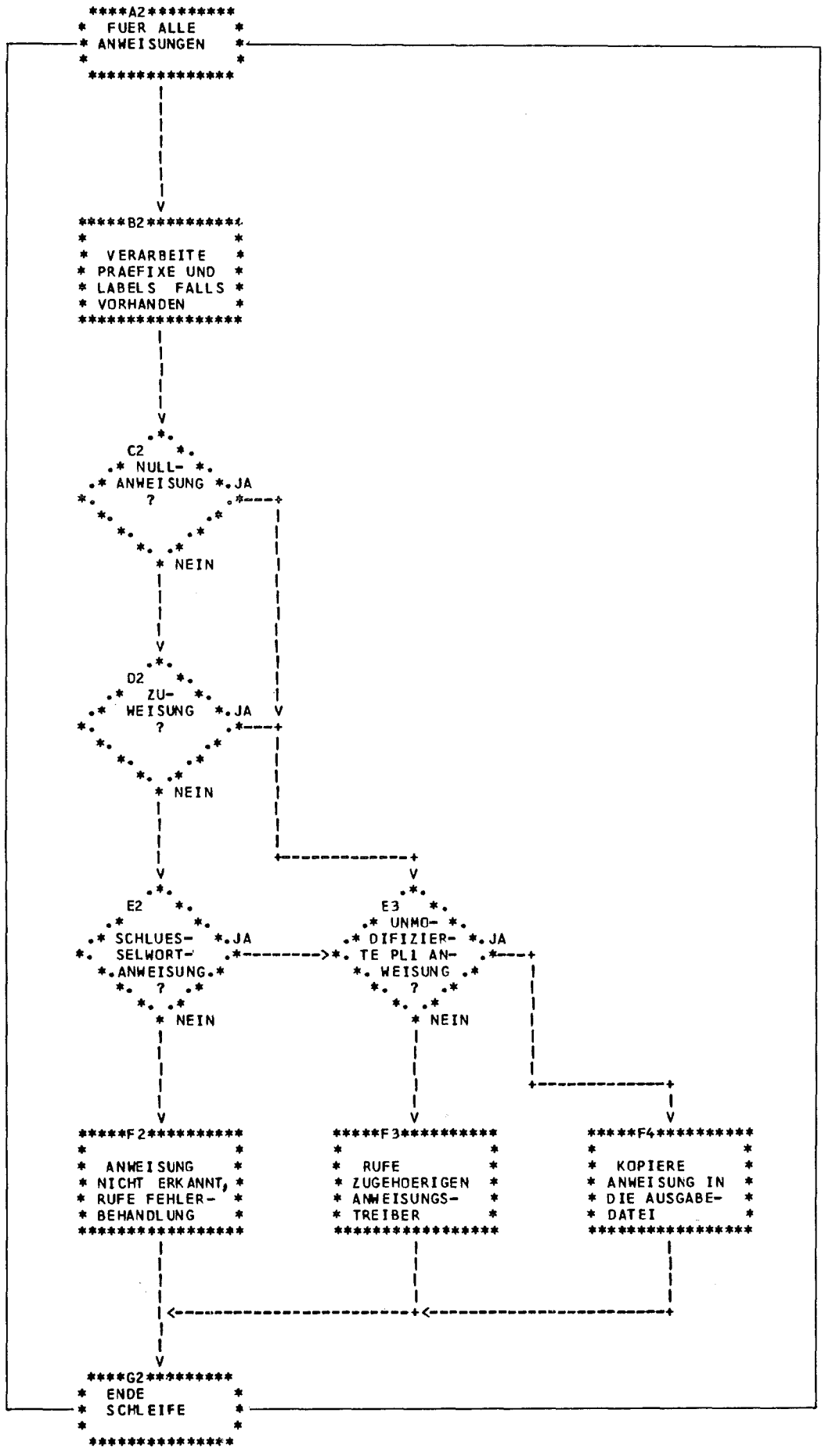


Abb. 5.2: Erkennen von POL-Anweisungen

5.3.2 Clauses

Enthalten mehrere POL-Anweisungen gleiche Teilgruppen, so kann zur Abarbeitung dieser Teile ein (externes) Unterprogramm aufgerufen werden. Diese Unterprogramme heißen Clauses, sie werden durch die CLAUSE-Anweisung definiert. Die Makrozeitanweisung "LINK clausename;" dient zum Aufruf von Clauses, "clausename" ist der bei der CLAUSE-Definition angegebene Name.

```
! Beispiel:
!
! Es sollen die drei Anweisungen PRINT, PUNCH und PLOT definiert
! werden mit der folgenden Syntax:
!
!       PRINT gruppe;
!       PUNCH gruppe;
!       PLOT gruppe;
!       gruppe ::= identifier [, identifier] *
!
! Die Definition lautet:
!
!       STA 'PRINT';
!       EXEC ...;
!       LINK GRUPPE;
!       END STA;
!       STA 'PUNCH';
!       EXEC ...;
!       LINK GRUPPE;
!       END STA;
!       STA 'PLOT';
!       EXEC ...;
!       LINK GRUPPE;
!       END STA;
!       CLAUSE'GRUPPE';
!              Verarbeite Gruppe von Benennungen
!       END CLAUSE;
```

Durch die Anweisung "CLAUSE INITIAL;" wird eine Clause definiert, die beim Subsystemstart (ENTER subsystem;) aufgerufen wird.

Sie kann zum Initialisieren von Datenstrukturen und Dateien dienen und z.B. die Ausgabeliste des Subsystems mit einer Überschrift versehen.

5.3.3 PLS-Funktionen

TYPE RETURNS (BIN FIXED(15)), liefert den Typ des nächsten in der Eingabe stehenden Elements zurück.

1 Integerzahl, Binary oder Decimal Fixed-Konstante ohne Vorzeichen

2 Realzahl, Binary oder Decimal-Float-Konstante ohne Vorzeichen

3 Zeichenkette

4 Bitkette

5 Benennung (Identifier)

6 Imaginärkonstante

7 Operator: +, -, *, /, &, |, ¬, ||, =, ¬=, <, <=, ¬<, >, >=, ¬>, **

8 Klammer auf: (

9 Klammer zu:)

10 Semikolon: ;

11 Wie 1, jedoch mit Vorzeichen + oder -

12 Wie 2, jedoch mit Vorzeichen + oder -

13 Anderes PL/1-Zeichen: _, %, ?

14 Nicht - PL/1 - Zeichen, z.B. !

15 End of File, Ende der Eingabe (EOF)

NEXT_ITEM (typ [, { SKIP / NOSKIP }])

RETURNS(CHAR(250) VARYING), liefert das nächste Element, das in der Eingabe vorhanden ist, der Typ des Elements wird in der BIN FIXED(15)-Variablen "typ" zurückgeliefert, die Werte haben die gleiche Bedeutung wie bei der PLS-Funktion"TYPE". SKIP bedeutet, daß das Element in der Eingabe übergangen wird. Bei NOSKIP wird der Eingabezeiger nicht verändert. Der Defaultwert ist SKIP.

NEXT_REAL
NEXT_STRING
NEXT_BITSTRING
NEXT_OPERATOR
NEXT_IDENTIFIER
NEXT_WORD
NEXT_INTEGER

[{ SKIP | NOSKIP }]

RETURNS (CHAR(250) VARYING), liefert ein Element aus der Eingabe,
NEXT_REAL liefert eine Decimal Float oder Binary Float-Konstante mit oder ohne Vorzeichen,
NEXT_INTEGER liefert eine Decimal oder Binary Fixed-Konstante mit oder ohne Vorzeichen,
NEXT_STRING liefert eine Zeichenketten-Konstante einschließlich der Apostrophe,
NEXT_BITSTRING liefert eine Bitketten-Kosntante einschließlich der Apostrophe und des "B",
NEXT_OPERATOR liefert einen PL/1-Operator,
NEXT_IDENTIFIER liefert eine Benennung,
NEXT_WORD liefert alle Zeichen bis zum nächsten Leerzeichen in der Eingabe oder bis zum nächsten Semikolon.

Falls das gewünschte Element nicht vorhanden ist, d.h. wenn der Typ des nächsten Elements in der Eingabe nicht gleich dem angeforderten Typ ist, erfolgt eine Fehlermeldung. Es wird dann Null oder der Nullstring zurückgegeben.

SKIP: Das gelieferte Element in der Eingabe wird übergangen.
NOSKIP: Der Eingabezeiger bleibt auf dem gelieferten Element stehen.

Ist bei NEXT_WORD das nächste Zeichen in der Eingabe ein Semikolon, so wird ein Semikolon zurückgeliefert, gleichzeitig erfolgt eine Fehlermeldung.

Beispiele:
```
I  = NEXT_INTEGER;
C  = NEXT_STRING;
C1 = NEXT_WORD(NOSKIP);
C2 = NEXT_ITEM(I,SKIP);
```

NEXT_N(n [{, SKIP / NOSKIP}])

RETURNS(CHAR(250)VARYING), liefert die nächsten z Zeichen die in der Eingabe stehen. "n" ist BIN FIXED(15) und muß 250 sein. Sind bis zum Ende der Eingabedatei weniger als n Zeichen vorhanden, erfolgt eine Fehlermeldung, es werden dann nur die vorhandenen Zeichen geliefert.

SKIP-NOSKIP haben die Bedeutung wie bei den anderen PLS-Funktionen.

NEXT_EXPRESSION [({SKIP / NOSKIP})]

RETURNS(CHAR(250)VARYING), liefert den nächsten arithmetischen, logischen oder Zeichenketten-Ausdruck, der in der Eingabe steht.

THIS_STATEMENT [({SKIP / NOSKIP})]

RETURNS(CHAR(3500)VARYING), liefert die aktuelle Anweisung vom Anfang bis einschließlich Semikolon, ohne Labels und Prefixes.

WORD(xyz [{, SKIP / NOSKIP}])

BWORD(xyz [{, SKIP / NOSKIP}])

RETURNS (BIT(1)), stellt fest, ob die Zeichen "xyz" gleich dem nächsten Wort in der Eingabe sind (bei WORD) oder ob "xyz" der Anfang des nächsten Wortes in der Eingabe ist (bei BWORD, B steht für Begin). Ein Wort ist in diesem Sinne eine Folge von Zeichen außer Leerzeichen und Semikolon, abgeschlossen durch Leerzeichen oder Semikolon. Wird das Wort (oder der Wortanfang) gefunden, wird '1'B zurückgegeben, sonst '0'B.

IDENTIFIER(xyz [,{SKIP / NOSKIP}])

BIDENTIFIER(xyz [,{SKIP / NOSKIP}])

RETURNS(BIT(1)), stellt fest, ob die Zeichen "xyz" die nächste Benennung (Identifier) bzw. der Anfang der nächsten Benennung in der Eingabe sind. Eine Benennung ist eine Folge von Zeichen, bestehend aus Buchstaben, Ziffern und den Zeichen $, #, @, _ . Die Zeichenfolge muß mit einem Buchstaben oder #, $, @ beginnen. Benennungen oder Identifier sind also gültige PL/1-Variablennamen. Wird die Benennung gefunden, wird '1'B zurückgeliefert und, falls "SKIP" angegeben wurde, übergangen, Ist die Benennung (oder deren Anfang) nicht in der Eingabe vorhanden, wird '0'B zurückgegeben, der Eingabezeiger wird nicht verändert.

ISEXPRESSION

RETURNS(BIT(1)), liefert '1'B, falls als nächstes in der Eingabe ein arithmetischer, logischer oder Zeichenketten-Ausdruck steht. Sonst ist das Ergebnis '0'B.

FIND(xyz)

RETURNS(BIT(1)), liefert '1'B, falls in der behandelten POL-Anweisung bis zum Semikolon die Zeichen "xyz" der Beginn eines Elements sind. Die Zeichenfolge xyz wird also nur am Beginn einer Benennung einer Konstanten, eines Operators oder eines Begrenzers wie , ,), (,; gesucht. Wird die Zeichenfolge "xyz" in der Anweisung nicht gefunden, wird '0'B zurückgeliefert.

Beispiele:

```
B1 = WORD('DIE');
B2 = BWORD('FOL',NOSKIP);
B3 = WORD('???');
B4 = BWORD('-.-',NOSKIP);
B5 = ID('NAME');
B6 = BID('BEN');
IF ISEX | FIND('HOLE') | WORD('xyz') THEN ...
IF ¬FIND('STACK') THEN GOTO ENDE;
DO WHILE( ¬IDENTIFIER('STACK'));
       SKIP;
       END;
```

5.3.4 PLS-Anweisungen, die innerhalb von STATEMENT- und CLAUSE-Definitionen stehen können

SKIP; SKIPB;	Diese Anweisungen dienen zum Übergehen eines Elements in der Eingabe, sie setzen den Eingabezeiger um ein Element nach rechts. Ob der Eingabezeiger verändert wird, kann davon abhängig gemacht werden, ob ein bestimmtes Element oder eine Elementart als nächstes in der Eingabe steht.
SKIP REAL;	Übergehe das nächste Element, falls es eine Realkonstante ist.
SKIP INTEGER;	Übergehe das nächste Element, falls es eine Integerkonstante ist.
SKIP OPERATOR;	Übergehe das nächste Element, falls es ein Operator ist.
SKIP STRING;	Übergehe das nächste Element, falls es eine Zeichenkette ist.
SKIP BITSTRING;	Übergehe das nächste Element, falls es eine Bitkette ist.
SKIP EXPRESSION;	Übergehe einen arithmetischen, logischen oder Zeichenkettenausdruck, falls ein solcher als nächstes in der Eingabe vorhanden ist.
SKIP WORD;	Übergehe das nächste Wort in der Eingabe (alle Zeichen bis zum nächsten Semikolon oder Leerzeichen).
SKIP n;	"n" ist eine Integerkonstante. Übergehe die nächsten n Zeichen.
SKIP IDENTIFIER;	Übergehe das nächste Element in der Eingabe, falls es eine Benennung ist.

```
SKIP IDENTIFIER(xyz);   Übergehe das nächste Element in der Eingabe,
                        falls es genau die Benennung "xyz" ist.

SKIPB IDENTIFIER(xyz);  Übergehe das nächste Element, falls es eine
                        Benennung ist, die mit "xyz" beginnt.

SKIP(xyz);              Übergehe das nächste Wort in der Eingabe,
                        falls es genau aus den Zeichen "xyz" besteht.

SKIPB(xyz);             Übergehe das nächste Wort, falls es mit den
                        Zeichen "xyz" beginnt.

SKIP;                   Übergehe das nächste Element in der Eingabe.
```

```
! Beispiele:
!           SKIP REAL; SKIP INT; SKIP OPERATOR;
!           SKIP STRING; SKIP BIT; SKIP EX;
!           SKIP WORD; SKIP 12; SKIP ID;
!           SKIP IDENTIFIER('DURCH');
!           SKIP ID('DREI');
!           SKIP ('DER');
!           SKIPB ('ZEI');
!           SKIP;
```

```
PLI;                    Diese Anweisung teilt dem PLS-Übersetzer mit,
                        daß er die Anweisung als System- oder PL/1-
                        Anweisung behandeln soll. Man kann PL/1-An-
                        weisungen auf bestimmte Eigenschaften und
                        Optionen untersuchen, indem man eine POL-
                        Anweisung gleichen Namens definiert. Will
                        man dann die Anweisung unverändert lassen,
                        benutzt man dazu die PLI-Anweisung. PLI
                        impliziert ein RETURN-Statement.
EXECUTE;                Die Anweisungen EXEC und END EXEC umschließen
    .                   die bei der Abarbeitung der POL-Anweisung zu
    .                   generierenden PL/1-Anweisungen. Während der
    .                   Übersetzungsphase des PLS-Übersetzers wird
END EXECUTE;            eine solche "EXEC-Gruppe" nicht ausgeführt,
                        sondern in das erzeugte PL/1-Programm kopiert.
```

Erst zur Ausführungszeit des generierten Programms werden die zwischen EXEC und END EXEC stehenden Anweisungen aktiv. Soll nur eine Anweisung erzeugt werden, kann sie auch zwischen "EXEC" und ";" stehen:

```
EXEC PUT LIST('BEISPIEL')SKIP;;
```

Dabei ist zu beachten, daß sowohl das Semikolon für die EXEC-Anweisung als auch das für die zu generierende Anweisung stehen muß. Mit Hilfe der Kurzform der EXEC-Anweisung lassen sich auch Teile von Anweisungen generieren: "EXEC MULT (A,B); " erzeugt den Teil einer PL/1-Anweisung "MULT(A,B)" (ohne Semikolon). In einer EXEC-Gruppe können beliebige PL/1-Anweisungen stehen. Es ist darauf zu achten, daß das erzeugte Programm auf jeden Fall syntaktisch richtig wird. Labels vor Anweisungen sind problematisch, da bei mehrmaliger Anwendung der gleichen POL-Anweisung das Label mehrfach deklariert wäre. Man kann entweder jedesmal ein neues Label mit Hilfe einer Makrozeit-Variablen generieren oder die Gruppe von Anweisungen, die das Label enthält, zwischen BEGIN; oder END; setzen.

Werden durch eine POL-Anweisung mehrere PL/1-Anweisungen erzeugt, so ist es erforderlich, sie als DO-Gruppe zu generieren.

! Beispiel:
!
! Die POL-Anweisung "DRUCKE ALLES;" soll die Anweisungsfolge "PUT EDIT (HEADLINE) (SKIP,A); CALL PRINT(3); erzeugen. Wenn der POL-Programmierer nun schreibt: "IF X > 0 THEN DRUCKE ALLES;" und die erzeugten PL/1-Anweisungen stehen nicht zwischen DO und END, ergeben sich Resultate, die der POL-Programmierer nicht erwartet (im Beispiel würde also die zweite Anweisung CALL PRINT(3); nicht in der THEN-Clause stehen. Falls erforderlich, kann auch "BEGIN;" und "END;" die erzeugten Anweisungen umschließen. Dies ist immer dann nötig, wenn DECLARE-Anweisungen generiert werden, um doppelte Deklarationen zu vermeiden.

! Beispiel
!
! Eine Liste von Werten und deren Anzahl soll an eine
! Routine übergeben werden. POL-Anweisung:
! RECHNE x_1, y_1 [,x_i, y_i]*; Realzahlen
! Generierte Anweisungen:

```
!       BEGIN;
!       X(1)  = x1;
!       Y(1)  = y1;
!       X(2)  =
!          .
!          .
!       DCL (X(N), Y(N)) DECIMAL FLOAT;
!       CALL RECHNE (X,Y,N);
!       END;
```

! Statementdefinition:

```
!       STATEMENT 'RECHNE'
!       DCL I BIN FIXED(15);
!       EXEC BEGIN;;
!       I = 0;
!       DO WHILE (TYP¬ = 10 & TYP ¬= 15);
!       I = I+1;
!       EXEC X(I) = NEXT_REAL;;
!       SKIP(',');
!       EXEC Y(I) = NEXT_REAL;;
!       SKIP(',');
!       END;
!       EXEC;
!       DCL (X(I), Y(I)) DECIMAL FLOAT;
!       CALL RECHNE (X,Y,I);
!       END;
!       END EXEC;
!       END STATEMENT;
```

! Erläuterung
!
! Dieses Beispiel zeigt auch, daß die generierten PL/1-
! Anweisungen in bestimmter Weise variiert werden kön-
! nen. In dem Beispiel wird im generierten Text der
! Name "I" durch den Wert der Variablen I ersetzt, eben-
! so wird der Name "NEXT_REAL" durch den Wert der PLS-
! Funktion NEXT_REAL ersetzt. Variable, deren Namen in

! der EXEC-Gruppe durch ihren Wert ersetzt werden,
! heißen "Ersetzungsvariable", ihr Wert heißt
! "Ersetzungswert". ("replacement variable", "replace-
! ment value").
! Eine Ersetzung findet nur für aktive Variable statt.
! Aktiv sind neben den PLS-Funktionen alle arithmeti-
! schen oder Zeichenkettenvariablen (mit den Attribu-
! ten BINARY, DECIMAL, PICTURE oder CHARACTER) inner-
! halb des Blockes, in dem die durch eine DECLARE-
! Anweisung deklariert sind. Im obigen Beispiel sind
! also die Variablen I (da als BINARY deklariert) und
! NEXT_REAL (da PLS-Funktion) aktiv. Mit Hilfe von
! ACTIVE und UNACTIVE-Anweisungen können unaktive
! Variable aktiviert und aktive desaktiviert werden.

ACTIVE actname [,actname]* ;

Die ACTIVE-Anweisung ist keine ausführbare Anweisung, sondern eine Deklaration, die innerhalb des PL/1-Blocks gültig ist, in dem sie steht. Die Variablennamen "actname" werden, wenn sie in einer EXEC-Gruppe angetroffen werden, durch den Wert der Variablen dieses Namens ersetzt. Mit der ACTIVE-Anweisung können Variable aktiviert werden, die nicht default-mäßig aktiv sind, wie etwa Bitketten. Auch nicht explizit deklarierte Variable oder solche, denen durch eine DEFAULT-Anweisung Attribute zugewiesen werden, können nur durch die ACTIVE-Anweisung zu Ersetzungsvariablen werden.

UNACTIVE uname [,uname]*;

Ebenso wie ACTIVE ist UNACTIVE keine ausführbare Anweisung, sondern eine Deklaration. Die Variablen "uname" sind in dem PL/1-Block, in dem die Anweisung steht, nicht aktiv. Es können standardmäßig aktive Variable somit von der Ersetzung in EXEC-Gruppen ausgeschlossen werden. Im äußersten Block einer Statement- oder Clause-Definition können auch PLS-Funktionen desaktiviert werden.

EXECUTE LINK routine (argumente) ;;

Innerhalb einer EXEC-Gruppe sind zwei Nicht-PL/1-Anweisungen zulässig, nämlich LINK und MESSAGE. Die Link-Anweisungen dienen dazu, zur Ausführungszeit des POL-Programms Entries in PLR-Moduln dynamisch aufzurufen. "routine" muß ein Name einer DYNAMIC ENTRY-Variablen sein. Das dynamische Laden und Ausführen des Moduls wird von der REGENT-Modulverwaltung RMM vorgenommen. Der Modul muß daher ein vom REGENT-Modulgenerator erzeugter Modul sein. "argumente" sind die an die Routine zu übergebenden Argumente. Ein Entry, der in einem Subsystem durch EXEC LINK aufgerufen wird, muß in der Subsystem-Datenstruktur (z.B. im Subsystem-Common) als DYNAMIC ENTRY mit allen Argumenten (falls vorhanden) deklariert sein.

EXECUTE MESSAGE

Durch eine EXEC-MESSAGE-Anweisung wird im generierten PL/1-Text eine Fehlernachricht erzeugt. Die Syntax entspricht der Syntax der PLR-MESSAGE-Anweisung und der Systemanweisung MESSAGE.

5.3.5 Die FILE-Anweisung

Die PLS-Anweisungen STATEMENT und CLAUSE werden zu PL/1-Programmen erweitert, vom PL/1-Compiler übersetzt und zu ausführbaren Moduln gebunden. Dies erfolgt normalerweise ohne Eingreifen des Anwenders automatisch. Das Übersetzen geschieht zeitlich jeweils bei END STATEMENT bzw. bei END CLAUSE. Das Binden aller Treibermodule erfolgt gemeinsam bei END PLS. Für fast alle Fälle reicht diese Art der Verarbeitung aus. Für einige Sonderfälle kann die FILE-Anweisung dazu dienen, das Binden der Treibermodule zu steuern:

- Das Binden kann an beliebiger Stelle im PLS-Programm erfolgen (natürlich erst nach der STATEMENT- oder CLAUSE-Definition).

- Mehrere STATEMENT- oder CLAUSE-Treiberroutinen können in einen Modul zusammengebunden werden.

Die einfachste und normalerweise angewandte Form der Anweisung ist: FILE;. Es werden alle bis dahin seit der letzten FILE-Anweisung definierten STATEMENT- und CLAUSE-Routinen, jede für sich, zu einem Modul gebunden und in der Datei REGENT.PLSTRAN.MODS abgelegt. Bei der Erzeugung sehr vieler Statements und Clauses werden so die schon übersetzten Treiberroutinen geladen, im Falle eines vorzeitigen Job-Abbruchs sind dann nicht alle Definitionen noch einmal zu wiederholen.

Die FILE-Anweisung ermöglicht es aber auch, verschiedene Statement- und Clause-Treiberroutinen und schon fertig gebundene Module zu einem einzigen Modul zusammenzubinden. Dies ist dann effektiver, wenn die Routinen meist zusammen benutzt werden, da dann nur einmal der Modul von der Platte geladen werden muß.

Normalform:

```
!           FILE;
```

Zusammenbinden der Anweisungsdefinitionen STA1, STA2, STA3 in einen Modul:

```
!           FILE  ('STA1', 'STA2', 'STA3');
```

5.3.6 Das Verarbeiten mehrerer POL-Anweisungen in einer STATEMENT-Definition

Es ist in manchen Fällen wünschenswert, wenn eine STATEMENT-Definition mehrere hintereinanderstehende POL-Anweisungen nacheinander verarbeiten kann. Im REGENT-Subsystem PLS geschieht dies z.B. bei DATA ... END DATA; STAT ... END STAT; use. In diesen Fällen muß der Subsystemprogrammierer zusätzliche Dinge beachten:

- Er muß Prefixe und Labels vor den Anweisungen behandeln.
- Er muß den Typ einer Anweisung feststellen können, u.a. ob eine Zuweisung vorliegt.
- Er muß dafür sorgen, daß PLS ihm die nächste Anweisung zur Verfügung stellt.

Zu diesen Zwecken stellt PLS einige Prozeduren, die normalerweise von PLS intern benutzt werden, auch dem Subsystemprogrammierer zur Verfügung. Die Verarbeitung mehrerer POL-Anweisungen geht in folgenden Schritten vor sich:

1. Verarbeite die Anweisung bis zum Semikolon (einschließlich).

2. Rufe die Prozedur QQNEW_STATEMENT auf, um die fertig bearbeitete Anweisung zu listen und eine neue Anweisung im Eingabepuffer von PLS zur Verfügung zu haben. QQNEW_STATEMENT hat keine Argumente.

3. Rufe die Pozedur PL_PREFIXES auf mit einem CHAR(x)VARYING-Argument, um die Prefixe der neuen Anweisung zu erhalten. Die Prefixe einschließlich aller Klammern und Doppelpunkte werden in dem Argument zurückgegeben. Sind keine Prefixe vorhanden, wird der Nullstring zurückgegeben.

4. Rufe die Prozedur PL_LABELS auf, um die Labels der Anweisung zu erhalten. Alle Labels einschließlich aller Doppelpunkte werden in dem einen CHAR(x)VARYING Argument der Prozedur zurückgegeben, sind keine Labels vorhanden, wird der Nullstring geliefert.

5. Benutze die Funktionsprozedur PL_ASSTEST um festzustellen, ob die Anweisung eine Zuweisung ist. PL_ASSTEST bestitz keine Argumente und liefert als Funktionswert einen BIT(1)-Wert. Ist die Anweisung eine Zuweisung, ist der Funktionswert '1'B, im anderen Fall '0'B.

6. Falls die Anweisung keine Zuweisung ist, betrachte den Anfang der Anweisung (z.B. mit NEXT_IDENTIFIER) und bestimme den Typ der Anweisung.

7. Verarbeite die Prefixe, die Labels und die Anweisung selbst (z.B. indem sie unverändert oder modifiziert mittels EXEC zu dem generierten Text geschrieben wird).

8. Stelle fest, ob dies die letzte Anweisung der zu verarbeitenden Anweisungsgruppe war. Wenn ja, höre auf (RETURN), wenn nein, gehe zum Punkt 2.

```
!  Beispiel
!
!  Nach der POL-Anweisung LISTE; sollen alle folgenden Anweisungen
!  bis LISTEND; durch ein Unterprogramm LOOKLIST bearbeitet werden.
!  Prefixe und Labels sollen unverändert bleiben. Zuweisungen sollen
!  ebenfalls unverändert ausgegeben werden.
!
!  Definition:
!
!  STA 'LISTE';
!  SKIP (';');                          /* übergehe Semikolon */
!  ANW:
!  CALL QQNEW_STATEMENT;                /* hole neue Anweisung */
!  DCL (P,L) CHAR(250) VARYING;         /* Platz für Prefixe u.Labels */
!  CALL PL_PREFIXES(P);                 /* hole Prefixe */
!  CALL PL_LABELS(L);                   /* hole Labels */
!  IF P  = ''THEN EXEC P;               /* Prefixe in den Output */
!  IF L  = ''THEN EXEC L;               /* Labels in den Output */
!  IF PL_ASSTEST THEN EXEC THIS_STATEMENT; /* Zuweisung unverändert
                                                 ausgeben */
!  ELSE DO;                             /* Verarbeite Anweisung */
!       IF IDENTIFIER('LISTEND') THEN GOTO ENDE;  /* letzte Anweisung? */
!       CALL LOOKLIST;
!       END;
!  GOTO ANW;                            /* nächste Anweisung */
!  ENDE :
!  END STATEMENT;
```

5.4 Datenstruktur-Definition

Datenstrukturen in Form von PL/1-Datendeklarationen, die in einem Subsystem während dessen Ausführung benötigt werden, können bei der Subsystemerstellung einmal definiert werden. Sie können danach bei jeder Subsystemanwendung benutzt werden. Datenstrukturen, die zur Übersetzungszeit der Subsystemsprache (zur Makrozeit) zum Zweck der Sprachübersetzung (z.B. zur Kommunikation zwischen Anweisungstreibern) verwendet werden, heißen Makrozeit-Datenstrukturen. Datenstrukturen, die zur Ausführungszeit des Subsystems benutzt werden, werden hier Subsystem-Datenstrukturen genannt.

5.4.1 Ausführungszeit-Datenstrukturen

Subsystem-Datenstrukturen (Ausführungszeit-Datenstrukturen, siehe Kap.2) werden zwischen den PLS-Anweisungen "DATA STRUCTURE dname;" "END DATASTRUCTURE;" definiert als Folge von PL/1-DECLARE-Anweisungen. Diese Deklarationen werden beim Übersetzen von POL-Anweisungen nach dem Eröffnen eines Subsystems (ENTER subname;) in das erzeugte PL/1-Programm eingeschoben und stehen somit für die Dauer der Subsystem-Anwendung zur Verfügung. Beliebig viele Subsystem-Datendeklarationen können angegeben werden. Jede Folge von Deklarationen wird durch einen Namen identifiziert. Durch Angabe dieses Namens in einer "DESTROY DATASTRUCTURE"-Anweisung kann die Datenstruktur wieder gelöscht werden. Die Subsystem-Datenstruktur dient zur Aufbewahrung von Subsystem-Daten während der Ausführungszeit und zur Kommunikation zwischen Subsystem-Routinen. Dem POL-Programmierer kann die Möglichkeit gegeben werden, in der Subsystem-Datenstruktur von den Subsystem-Moduln abgelegte Werte zu verwenden. Eine ausgezeichnete Datenstruktur für jedes Subsystem ist der Subsystem-Common (s.Kap.2). Er wird mittels der Anweisung "DATA STRUCTURE COMMON;" definiert. Eine Deklaration in Form einer einzigen PL/1-"Structure" wird benutzt, um alle diejenigen Subsystem-Daten aufzunehmen, die nicht nur in der POL, sondern auch in allen Subsystem-Moduln ansprechbar sein sollen. Auch alle externen subsystemspezifischen POL-Routinen können auf den Subsystem-Common zugreifen.

Die Subsystem-Datenstrukturen müssen deklariert werden, bevor sie in einem Modul benutzt werden können. Für die Datendeklarationen sind außer PL/1-Datenattributen auch die REGENT-Attribute DYNAMIC ENTRY, BANK und BASEDESCRIPTOR zulässig. Für jeden zu dem Subsystem gehörenden auf der PLS-Ebene dynamisch aufrufbaren Modul (also für jeden Modul, dessen Name in einer EXEC LINK-Anweisung in einer STATEMENT- oder CLAUSE-Definition erscheint) ist im Subsystem-Common oder einer anderen Datenstruktur eine "DECLARE...ENTRY(.....) DYNAMIC"-Deklaration erforderlich.

Der Subsystem-Common wird beim Subsystem-Start angelegt und initialisiert. Die Initialisierung ist möglich durch Angabe von INIT-Optionen in der Deklaration oder durch Anweisungen, die in einer "INITIAL CLAUSE" stehen. Ein Subsystem kann zur Ausführungszeit statt auf externe Moduln auf interne PL/1-Routinen zugreifen. Diese müssen auch in der Common-Definition enthalten sein. Aufgerufen werden sie zweckmäßigerweise über ENTRY-Variable, die im Subsystem-Common stehen und mit den internen Prozeduren initialisiert sind.

! Beispiel
!
```
!            DATA COMMON;
!               DCL 1, . . .
!                   ;
!                   ;
!                   2 #RV1 ENTRY VARIABLE INIT (#R1),
!                   2 #RV2 ENTRY VARIABLE INIT (#R2);
!              #R1:  PROC;
!
!                    END #R1;
!               #R2:  PROC;
!
!                    END #R2;
!            END DATA;
```
! Über die Namen #RV1 und #RV2 kann so im gesamten Subsystem
! (in der POL, in allen Moduln und in allen externen subsystem-
! spezifischen POL-Routinen) auf die Prozeduren #R1 und #R2 zugegriffen werden.Wenn #R1 und #R2 externe Prozeduren sind, muß
! die Deklaration der ENTRIES #R1 und #R2 in obigem Beispiel
! lauten:
```
!          DCL (#R1, #R2) ENTRY EXTERNAL;
```

Die Variablennamen #R1, #R2, #RV1 und #RV2 sind durch das Schutzzeichen "#" vor dem direkten Zugriff des Subsystem-Anwenders geschützt. Sie können nur in Moduln und in EXEC-Gruppen in STATEMENT- und CLAUSE-Definitionen verwendet werden.

5.4.2 Übersetzungszeit-Datenstrukturen

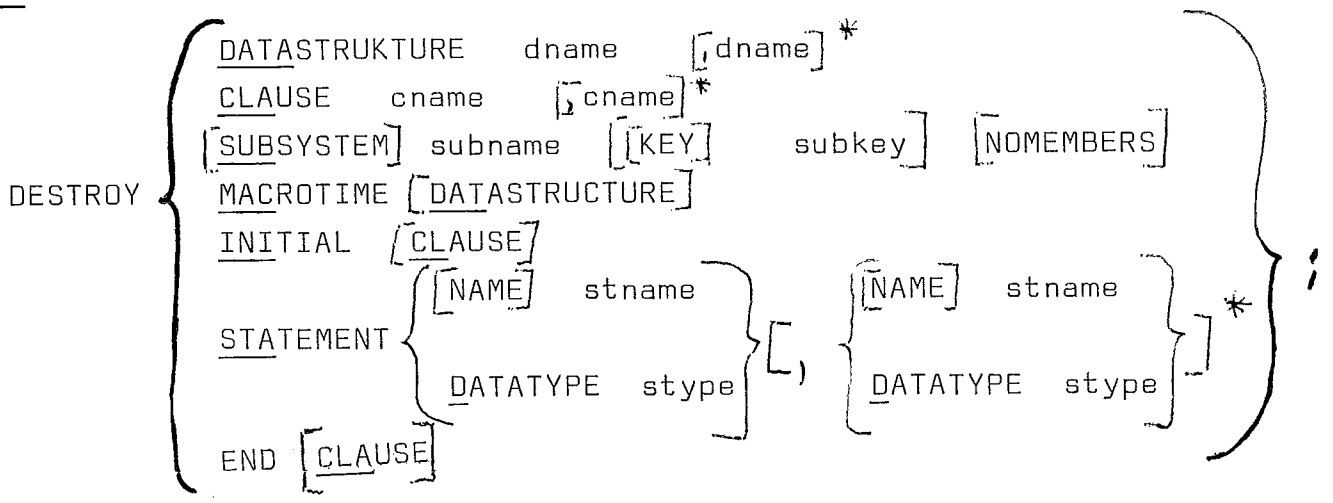
Sie werden nur in seltenen Fällen benötigt. Die Subsystem-Datenstrukturen existieren während einer Subsystem-Anwendung nur zur Ausführungszeit des kompilierten POL-Programms, während der Übersetzungsphase sind sie nicht vorhanden. Die Anweisungs-Treiber-Routinen, die dazu dienen, eine bestimmte Anweisung zu expandieren, sind als unabhängige Module in einer Bibliothek gespeichert und werden bei Bedarf dynamisch in den Arbeitsspeicher geladen. Sie können daher untereinander nicht über globale interne oder auch externe Variable kommunizieren. Mit der Anweisung "MACROTIME DATASTRUCTURE" wird eine Übersetzungszeit-Datenstruktur deklariert, die den Treiberroutinen eine Kommunikation untereinander ermöglicht. Die Übersetzungszeit-Datenstruktur ist eine BASED PL/1-Struktur. Sie wird beim Subsystemstart angelegt. Ein Zeiger auf die Struktur wird an alle Treiberroutinen übergeben, so daß auf die globalen Übersetzungszeitvariablen in allen Routinen zugegriffen werden kann. Zur Übersetzungszeit können zum Abspeichern von Werten verkettete Listen (linked lists) verwendet werden. Der Listenkopf muß dann in der Übersetzungszeit-Datenstruktur gespeichert sein, damit alle Treiberroutinen auf die Listen zugreifen können. Mit Hilfe von verketteten Listen können so auch zur Übersetzungszeit Daten mit variablem Speicherplatzbedarf verwendet werden.

Wie die Subsystem-Datenstrukturen muß auch die Übersetzungszeit-Datenstruktur deklariert werden, bevor sie in einer STATEMENT- oder CLAUSE-Definition benutzt wird. Erweiterungen am Ende der Struktur können vorgenommen werden, ohne daß alle Treiberroutinen neu übersetzt werden müssen. Wird jedoch die Übersetzungszeit-Datenstruktur abgeändert, so müssen alle STATEMENT- und CLAUSE-Definitionen wiederholt werden, die auf die Struktur zugreifen.

5.5 Löschen von Anweisungen und Datenstrukturen

Mit der DESTROY-Anweisung werden Teile eines Subsystems oder ein ganzes Subsystem gelöscht.

Syntax



Erläuterung

DESTROY DATASTRUCTURE löscht die aufgeführten Subsystem-Datenstrukturen mit dem Namen "dname".

DESTROY CLAUSE zerstört die aufgeführten Clauses mit dem Namen "cname".

DESTROY SUBSYSTEM löscht ein Subsystem mit dem Namen "subname", falls der richtige Schlüssel "subkey" angegeben wird. Die Option NOMEMBERS verhindert das Löschen der Subsystemmdoule.

DESTROY MACROTIME DATASTRUCTURE löscht die Makrozeit-Datenstruktur des gerade behandelten Subsystems.

DESTROY INITIAL CLAUSE löscht die Initialisierungsroutine des gerade behandelten Subsystems.

DESTROY STATEMENT löscht die angeführten POL-Anweisungen. "stname" ist das Schlüsselwort der POL-Anweisungen, die mit einem festen Schlüsselwort beginnen. "stype" ist der Typ der Datentyp-Anweisungen, die mit einem bestimmten Datentyp beginnen. "stype" kann ASSIGNMENT, PASSIGNMENT, REAL, INTEGER, IDENTIFIER, STRING, BITSTRING oder OPERATOR sein.

DESTROY END CLAUSE löscht die Ende-Clause des Subsystems.

5.6 Sonstige PLS-Anweisungen

COMPRESS, NOCOMPRESS

Die PLS-Systembibliotheken für die Anweisungsdefinitionen und die Tabellen sind Partitioned Data Sets (PDS) des OS/360. Sie müssen von Zeit zu Zeit komprimiert werden, um die Lücken zu schließen und ein Überlaufen zu verhindern. PLS komprimiert die Bibliotheken dann automatisch am Subsystemende, wenn der freie Platz eine installationsabhängige Grenze erreicht. Durch die COMPRESS-Anweisung kann unabhängig davon ein Komprimieren erzwungen werden:

```
COMPRESS;
```

Soll dagegen nicht komprimiert werden, kann dies erreicht werden durch:

```
NOCOMPRESS;
```

Anmerkung: Falls das Betriebssystem den dynamischen Aufruf von IEBCOPY nicht erlaubt, muß die Datei in einem eigenen Step komprimiert werden.

CPARM, LPARM

Die Statement- und Clause-Definitionsprogramme benutzen zur Erzeugung eines Objektmoduls einer Treiberroutine den PL/1-Optimizing Compiler und den Linkage Editor des OS/360. Mit der CPARM-Anweisung können die Compiler-Parameter geändert werden.
Default-Parameter für den Compiler:

```
'INCLUDE,NA,NAG,NOP,NSTG,SMSG,NX'.

! Ändern der Compiler-Parameter:
!
!          CPARM 'MACRO,NIS,LIST,XREF,A,AG';
```

Default-Parameter für den Linkage-Editor:

```
'DCBS,SIZE=(106K,24K),REUS,RENT,LIST'

! Ändern der Linkage-Editor-Parameter:
!
!       LPARM 'MAP,XREF,SIZE=(240K,100K)';
```

5.7 Private Bibliotheken für Subsysteme

Außer subsystemspezifischen Dateien und Datenbanken sind während der Subsystementwicklung und zur Subsystemanwendung drei Dateien wesentlich:

- Die Bibliothek der Subsystem-Module.
- Die Bibliothek der Subsystem-Routinen.
- Die Bibliothek der Anweisungs-Treibermodule. Sie muß im Übersetzungsstep (P-Step) einer REGENT-Anwendung und im REGENT-Go-Step, wenn das Subsystem PLS verwendet wird, ansprechbar sein. Der Standardname der Datei ist REGENT.PLSTRAN.MODS. Der DD-Name ist im P-Step QQCLIB und im Go-Step PLSLIB.

- Die Bibliothek der Übersetzertabellen und Datenstrukturen. Sie muß im P-Step und, falls das Subsystem PLS benutzt wird, auch im Go-Step einer REGENT-Anwendung über den DD-Namen SUBLIB ansprechbar sein. Ihr Standardname ist REGENT.PLSTRAN.DATA.

Für ein neu zu entwickelndes Subsystem können die Dateien benutzt werden, die auch die fertigen Subsysteme für REGENT-Anwendungen enthalten. Dies ist jedoch in zweierlei Hinsicht von Nachteil:
1. Durch das häufige Ändern und Komprimieren der Bibliotheken besteht die erhöhte Gefahr einer Zerstörung der Daten durch Software- oder Hardwarefehler. Es wären dann alle REGENT-Anwendungen bis zur Restaurierung der Dateien unmöglich. 2. Die REGENT-Dateien werden bei Änderungen für die Dauer der Änderung für alle anderen Benutzer gesperrt. Dies kann die Verweilzeit von REGENT-Anwendungsläufen im Rechner erhöhen. Man sollte daher die Dateien zumindest während der Subsystementwicklung getrennt anlegen. Das Verfahren wird hier beschrieben

PLR-Modul-Bibliothek:

> Eine neue Load-Bibliothek ist zu allokieren. Sie wird im Modulgenerator-lauf als SYSLMOD und bei Tests des neuen Subsystems im REGENT-Go-Step als STEPLIB angegeben.

Routinen-Bibliothek, Treiberbibliothek:

Auch hierfür ist je eine neue Load-Datei (PDS, Blocklänge 6447) zu allokieren.

Die Datei PLSTRAN.DATA mit den Subsystem-Tabellen darf für jedes REGENT-System nur einmal existieren.

Die drei benutzereigenen Bibliotheken müssen folgende DS-Namen haben:

plib.MODS für die Modulbibliothek
plib.LOAD " " Routinenbibliothek
plib.PLSTRAN.MODS für die Treiberbibliothek

Die Dateien sind auf einer Einheit "punit" mit dem Namen "pvol" zu allokieren.

Dann wird beim Definieren der Anweisungen mit PLS, beim Erzeugen von Modulen und Routinen mit PLR und beim Anwenden des Subsystems stets auf die privaten Bibliotheken zugegriffen, wenn sowohl in der Prozedur REGENT, als auch bei QQPCL die Parameter

PLIB=plib,PUNIT=punit, PVOL=pvol

verwendet werden.

Beispiel:

POL-Definition:

```
// EXEC REGENT,PLIB=SUB1,PUNIT=3330,PVOL=REGICE
//P.SYSIN DD *
      T: PROC OPTIONS(MAIN) REGENT(NODA);
       ENTER PLS;
       SUBSYSTEM SUB1;
         :
```

Modul-Erzeugung:

```
// EXEC QQPCL,PLIB=SUB1,PUNIT=3330,PVOL=REGICE
//P.SYSIN DD *
*SUBSYSTEM SUB1;
*MODULE ...  ;
   :
*ROUTINE ...;
   :
```

Subsystem-Ausführung:

```
// EXEC REGENT,PLIB=SUB1,PUNIT=3330,PVOL=REGICE
//P.SYSIN DD *
  S:PROC OPTIONS(MAIN)REGENT;
   ENTER SUB1;
     :
```

5.8 Die Druckausgabe bei Verwendung von PLS

Entsprechend den Job-Steps: POL-Übersetzung, PL/1-Kompilation und Ausführung gliedert sich die gedruckte Ausgabe eines PLS-Laufes in drei Teile.

POL-Übersetzungszeit

Zur Übersetzungszeit gibt der REGENT-Übersetzer eine Liste der Eingabe aus. Diese Liste wird nicht nur bei Anwendung von PLS, sondern bei allen REGENT-Anwendungen erzeugt. Mit der REGENT-Option LIST/NOLIST kann diese Ausgabe gesteuert werden. Die Anweisungen werden vom REGENT-Übersetzer durchnumeriert. Übersetzungszeit-Fehlermeldungen erscheinen vor der fehlerhaften Anweisung. Fehlermeldungen können vom REGENT-Übersetzer stammen, sie beginnen dann mit den Buchstaben POL, oder die Meldungen werden durch PLS erzeugt und sind durch die Buchstaben PLS kenntlich gemacht.

PL/1-Kompilation

Das vorübersetzte PL/1-Programm wird durch den PL/1-Compiler weiterverarbeitet. Dieser erzeugt dabei die üblichen Listen, die durch die Compiler-Parameter gesteuert werden können. Insbesondere wird das generierte PL/1-Programm gelistet (falls nicht NOSOURCE angegeben wurde). In dieser Liste sind die Anweisungsnummern des REGENT-Übersetzers als Kommentare eingefügt. Falls der PL/1-Compiler Fehlermeldungen erzeugt, kann so leichter zurückverfolgt werden, welche POL-Anweisung die Ursache der fehlerhaften PL/1-Anweisung ist.

Ausführungszeit

Zur Ausführungszeit werden ausgegeben:

- Vollzugs- und Fehlermeldungen
- Compilerausgabe
- Linkage-Editor-Ausgabe.

Vollzugsmeldungen geben (falls MESSAGE ACTIVE INFORMATIVE) Nachrichten über ausgeführte Arbeiten, z.B.: 'STATEMENT name HAS BEEN PROCESSED'. Fehlermeldungen informieren über aufgetretene Fehler. In Kapitel 13, Abschnitt "PLS-Fehlermeldungen" sind die Nachrichten aufgeführt und erläutert. Fehlermeldungen in Statementdefinitionen beinhalten die Anweisungsnummer des REGENT-Übersetzers.

Jede STATEMENT- und CLAUSE-Definition wird von PLS in ein PL/1-Programm und danach in einen Modul umgesetzt. Zur Kompilation wird der PL/1-Compiler, zum Binden des Moduls der Linkage Editor verwendet. Daher erscheint für jedes STATEMENT und für jede CLAUSE eine komplette Compilerdruckausgabe zur Ausführungszeit. Die Ausgabe kann über die Compiler-Parameter mit Hilfe der PLS-Anweisung CPARM gesteuert werden. Bei END PLS oder FILE wird das Binden der Definitionsroutinen zu Moduln veranlaßt. Für jeden Modul erfolgt eine Linkage-Editor-Druckausgabe. Sie kann durch die Linkage-Editor-Parameter gesteuert werden. Dies geschieht mit Hilfe der PLS-Anweisung LPARM.

! Beispiel

```
!           LPARM 'LIST,MAP,XREF';
!           CPARM 'SOURCE,A,AG,XREF,STG';
!           STA 'ABC';
!           END STA;        Hierfür erscheint eine Compilerliste

!           END PLS;        Jetzt erscheint eine Linkage-Editor-
                            Druckausgabe.
```

KAPITEL 6

Anwendung von Subsystemen

6.1 Ablauf eines REGENT-Programms

Ein REGENT-Programm, bestehend aus Subsystem-Aufrufen, subsystemspezifischen Problemsprachen-Anweisungen und PL/1-Anweisungen, wird durch den REGENT-Übersetzer in ein PL/1-Programm übersetzt, das anschließend kompiliert und ausgeführt wird. Abb. 6.1 zeigt den prinzipiellen Ablauf. Der Main-Modul, der schließlich abläuft, enthält den erforderlichen Teil der REGENT-Kernroutinen für die REGENT-Verwaltungsbereiche (Module, Dynamische Datenstrukturen, Datenbank, Nachrichten und Plotausgabe). Diese Teile werden durch den Linkage Editor oder Loader durch automatischen Bibliotheksaufruf in den Main-Modul eingefügt. Nicht ständig erforderliche Teile des REGENT-Kernes werden bei Bedarf dynamisch aus der REGENT-Bibliothek nachgeladen. Die subsystemspezifischen Module werden ebenfalls aus der Modulbibliothek geladen. Aus dem REGENT-Mainprogramm können außerdem vorübersetzte externe POL-Prozeduren aufgerufen werden. Diese werden durch den Linkage Editor oder Loader mit in den Main-Modul eingebunden.

Die Subsysteme können ihre Daten langfristig mit Hilfe der Datenbankverwaltung auf externen Datenträgern speichern. Eine oder mehrere Datenbänke können einem Subsystem zugeordnet sein. Eine Datenbank kann auch Daten aus Anwendung mehrerer Subsysteme enthalten. Außerdem können die Subsysteme natürlich alle PL/1-Möglichkeiten der Datenspeicherung nutzen. Der beschriebene Ablauf eines REGENT-Programms wird durch die Job-Control-Language-Prozedur REGENT realisiert. Ein Standard-REGENT-Job hat somit folgendes Aussehen:

```
//      Jobkarte
// EXEC REGENT,␣PLIB=libname
//P.SYSIN DD *
   MAIN: PROC OPTIONS(MAIN)REGENT(parms);
       REGENT-Programm
/*
//                              //
```

PLIB ist dann erforderlich, wenn die Subsystembibliothek auf privaten Dateien liegen (siehe Kapitel 5, S.5-31).

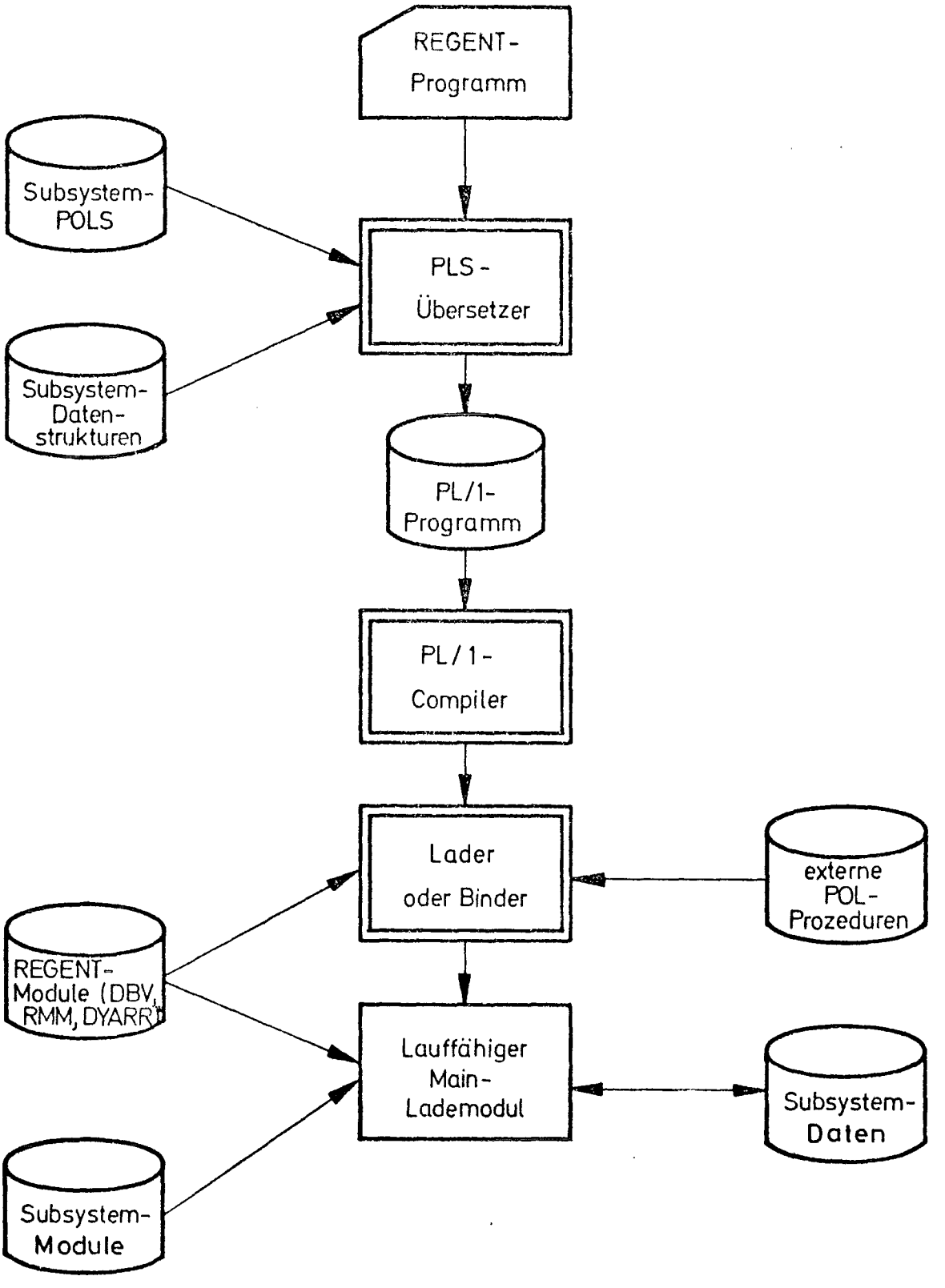


Abb.6.1: Ablauf eines REGENT-Programms

6.2 Die REGENT-Option auf der PROCEDURE-Anweisung

Ein REGENT-Programm wird vom PLS-Übersetzer an der REGENT-Option auf der PROC-Anweisung der externen Prozedur erkannt.

Syntax

```
REGENT | REGENT (option[,option]*)

option::= { INIT   } | { SUBSYSTEM    = subname } |
          { NOINIT }   { NOSUBSYSTEM            }

          { MOD    } | { DARRAYS } | { BANK   }
          { NOMOD  }   { NODA    }   { NOBANK }

          { LIST [= ddname] } |
          { NOLIST          }

            DPOOL = dpoolsize | { PLOT[= CALCOMP|STATOS  } |
                                {        |XYNETICS|TEKTRONIX]
                                { NOPLOT                 }
            MPOOL = mpoolsize
```

Standardwerte

REGENT(INIT,MOD,DA,LIST,NOBANK,DPOOL=100000,MPOOL=100000,NOPLOT)

Fehlt eine gültige Procedure-Anweisung als erste Anweisung eines Programms, wird die Anweisung

REGENT: PROC OPTIONS(MAIN) REORDER;

eingefügt.

Bedeutung

INIT: Die REGENT-Datenstruktur wird initialisiert, die REGENT-Kernroutinen werden in den Modul integriert, die Modul-Verwaltung und Dynamic-Array-Verwaltung (jeweils falls benutzt) werden initialisiert. Die INIT-Option muß also für den Main-Modul angegeben werden.

NOINIT: Es werden keine Initialisierungen durchgeführt. Die REGENT-Kernroutinen werden nicht in den Modul integriert. NOINIT muß für externe POL-Routinen benutzt werden, die aus einem Main-Modul gerufen werden sollen. Dasselbe gilt für POL-Module.

SUBSYSTEM = subname:

Es werden neben den fehlenden REGENT-Initialisierungen auch keine Subsystem-Initialisierungen vorgenommen. Das Subsystem-Environment des Subsystems "subname" wird der externen POL-Prozedur vom rufenden Modul übergeben. Diese Option impliziert NOINIT.

NOSUBSYSTEM: Die POL-Prozedur dieser Routine bzw. dieses Moduls ist nicht an eine spezielles Subsystem-Environment gebunden, das vom rufenden Modul übergeben wird. Diese Option impliziert NOINIT.

MOD: MOD gibt an, daß die Modulverwaltung benutzt wird. Dies ist Voraussetzung für die Benutzung der Verwaltung dynamischer Datenstrukturen und der Datenbankverwaltung.

NOMOD: Es wird keine dynamische Modulverwaltung benutzt, die entsprechenden Kernroutinen werden nicht in den Modul integriert, die Initialisierung für die Modulverwaltung entfällt.

DARRAYS: Es wird die REGENT-Dynamische-Datenstruktur-Verwaltung benutzt.

NODA: Die Verwaltung dynamischer Datenstrukturen wird nicht benötigt. Die entsprechenden REGENT-Kernroutinen fehlen, die entsprechende Initialisierung entfällt.

BANK: Die REGENT-Dateiverwaltung wird benötigt.

NOBANK: Die REGENT-Dateiverwaltung wird bei dieser Anwendung nicht benötigt.

LIST: PLS druckt auf SYSPRINT eine mit Statement-Nummern und Fehlermeldungen versehene Liste des POL-Programms aus. Ist "ddname" angegeben, erfolgt die Ausgabe der Programmliste auf die Datei dieses DD-Namens. Standardausgabedatei ist SYSPRINT.

NOLIST: Es werden nur die PROC-Anweisungen mit der REGENT-Option und die Fehlermeldungen gedruckt.

DPOOL: Gibt die Größe des Dynamic-Array-Bereiches an. Dies ist der Arbeitsspeicherbereich, in dem dynamische Datenstrukturen verwaltet werden.

NPOOL: Bereichsgröße für die dynamische Programmverwaltung.

PLOT: Es soll Zeichenausgabe erzeugt werden. Entweder die Plot-Routinen der verschiedenen Plotter stehen zur Verfügung (bei der GfK Calcomp, Statos, Cynetics, Tektronix, Standardwert: PLOT=CALCOMP).

NOPLOT: Es kann in diesem REGENT-Lauf keine Zeichenausgabe erzeugt werden.

6.3 Die FINISH-Anweisung

SYNTAX

FINISH,

Bedeutung

FINISH schließt das REGENT-System ab. FINISH muß die letzte ausführbare Anweisung eines REGENT-Programms sein.Insbesondere bei Verwendung von PLOT-Software ist die FINISH-Anweisung unbedingt erforderlich.Auch die Statistik der Modulverwaltung wird nur durch FINISH ausgegeben, vor FINISH ist "MESSAGE ACTIVE DEBUG" sinnvoll.

6.4 Der Subbsystemaufruf, die ENTER-Anweisung

Der Aufruf eines REGENT-Subsystems in einem REGENT-Programm geschieht mittels der Systemanweisung ENTER.

Syntax

ENTER subsystem [sonstiges,]

Bedeutung

Das Subsystem mit dem Namen "subsystemname" wird aktiv. Falls der Name abkürzbar ist, müssen die ersten i Buchstaben übereinstimmen. Aus wieviel Buchstaben die Abkürzung besteht, legt der Subsystemprogrammierer fest. Die Bedeutung und die Syntax von "sonstiges" liegt ebenfalls in der Hand des Subsystemprogrammierers.

ENTER ist keine ausführbare Anweisung, sondern der Beginn eines Subsystemblockes. Von ENTER bis zum zugehörigen END ist das "Environment" des Subsystems vorhanden, d.h. alle subsystemspezifischen Daten sind zugänglich, außer Systemanweisungen sind auch die zum Subsystem gehörigen POL-Anweisungen gültig. ENTER darf nur stehen, wo auch eine BEGIN- oder PROC-Anweisung stehen dürfte.

Beispiele

```
ENTER PLS; ENTER STRUDL POOL(40000);
```

6.5 Der Subsystemabschluß

Ein Subsystem wird abgeschlossen mittels der Anweisung

```
END subsystemname [sonstiges] ;
```

Diese Anweisung markiert das Ende eines Subsystemblockes. Die Daten des Subsystems sind nicht mehr zugreifbar, die Subsystem-POL wird nicht mehr erkannt. Die Bedeutung von"sonstiges"hängt vom Subsystem ab.

Subsystemblöcke können geschachtelt werden wie BEGIN- oder PROCEDURE-Blöcke in PL/1. In den inneren Blöcken kann außer auf die eigene Subsystem-Datenstruktur auch auf die Daten der umgebenden Subsysteme zugegriffen werden. Dies ermöglicht die Kommunikation von Daten über Subsystemgrenzen hinweg. POL-Anweisungen sind jedoch nur gültig für das gerade aktive Subsystem, nicht für die umgebenden.

Beispiel

Anweisungen	POL gültig von Subsystem	Datenstruktur zugreifbar für Subsysteme
ENTER A;		
	A	A
ENTER B;		
	B	A,B
ENTER C;		
	C	A,B,C
END C;		
	B	A,B
END B;		
	A	A
END A;		

6.6 Sonstige Systemanweisungen

a) MESSAGE-Anweisungen

Die MESSAGE-Anweisungen (siehe System-Handbuch S.6-47) sind in allen Subsystemsprachen verfügbar.

b) PL/1-Anweisungen

Alle PL/1-Anweisungen sind in REGENT-Programmen erlaubt.

c) DDS-Unterprogramme

Dem Subsystemanwender steht eine Reihe von Unterprogrammen zur Verfügung, die bei der Verwendung von DDS hilfreich sein können:

CALL QQDREO	(Wahl der Reorganisationsstrategie)
CALL QQDBDD	Basisdeskriptor-Dump
CALL QQDPOL	DDS-Poolstatistik
CALL QQDSTY	DDS-Dump(komplett)
CALL QQDSTR	DDS-Dump (Teil-Dump)
CALL QQDPSZ	Ändern der DDS-Poolsize

Diese Aufrufe werden im System-Handbuch ab S.6-15 erläutert.

d) RMM-Unterprogramme

Auch dem Subsystemanwender der RMM benutzt werden folgende Unterprogramme angeboten:

CALL QQMMAP	(Listen des RMM-Pools)
CALL QQMREO	(Reorganisieren des RMM-Pools)

Die Aufrufe werden im System-Handbuch ab S.6-15 erläutert.

e) Allgemeine Unterprogramme

Zum Messen der verstrichenen CPU-Zeit werden zwei Routinen zur Verfügung gestellt, die folgendermaßen aufgerufen werden können:

CALL QQTTIME und
CALL QQSTIME

Die Aufrufe werden im System-Handbuch ab S.6-15 erläutert.

6.7 Druckausgabe

Eine Subsystemanwendung im REGENT-System läuft in drei Schritten ab, der POL-Übersetzung, der PL/1-Kompilation und der Ausführung. Diesen drei Schritten entsprechen bei Anwendung der REGENT-Job-Control-Language-Prozedur drei OS-Job-Steps. In jedem dieser Schritte kann Druckausgabe erzeugt werden. Die bei allen REGENT-Anwendungen standardmäßig erzeugten Druckerlisten werden hier erläutert.

POL-Übersetzungszeit

Der REGENT-Übersetzer gibt eine mit Anweisungsnummern versehene Liste der Eingabe aus. Fehlt in der REGENT-Option der Parameter LIST/NOLIST oder wird nur List angegeben, erfolgt die Ausgabe auf den Ausgabedataset SYSPRINT. Bei NOLIST erfolgt keine Ausgabe. Mit LIST-ddname kann die Ausgabe auf die Datei mit dem DD-Namen ddname geleitet werden. In einer ersten Spalte erscheinen die vom REGENT-Übersetzer vergebenen Anweisungsnummern. Auf sie wird in Übersetzungszeit-Fehlermeldungen Bezug genommen. Jede Anweisung beginnt auf einer neuen Zeile. Der REGENT-Übersetzer behandelt auch die Anweisungen nach IF....THEN und nach ELSE als neue Anweisungen. Auch bei der Aufreihung von Deklarationen in einer DECLARE-Anweisung wird jeder neue Name als neue Anweisung gezählt.

Beispiel

```
5   DCL   (A,
6            B,
7             C, BIN FIXED(15);
8   IF L THEN
9                 GOTO M;
```

Auf den Eingabekarten stehende Karten-Numerierungen werden ebenfalls aufgelistet, allerdings nur eine Nummer pro Anweisung. Bei Anweisungen über mehrere Karten erscheint also nur auf der letzten Karte die Nummer.

Fehlermeldungen erfolgen ebenfalls auf die Datei SYSPRINT, sie stehen zwischen den gelisteten Eingabekarten. Eine Fehlermeldung steht dabei vor der fehlerhaften Anweisung. (Format der Fehlermeldungen siehe Kapitel 13). Meldungen des REGENT-Übersetzers tragen die Kennzeichnung REGENT und POL, Subsystem-Fehlermeldungen sind durch den Subsystem-Namen gekennzeichnet.

PL/1-Kompilation

Das vom REGENT-Übersetzer nach PL/1 gewandelte Programm wird durch den PL/1-Compiler weiterverarbeitet. Dabei werden die üblichen Compilerlisten erzeugt. Die Druckausgabe des Compilers läßt sich durch Angabe von Parametern steuern, der Parameter auf der EXEC-Karte heißt CPARM.

Beispiel

```
// EXEC REGENT,CPARM='MACRO,NIS'
```

Falls nicht der Parameter NOSOURCE spezifiziert wurde, erzeugt der Compiler eine Liste des vorübersetzten PL/1-Programms. In dieser Liste sind die Anweisungsnummern des REGENT-Übersetzers als Kommentare eingefügt. Sollte der PL/1-Compiler Fehlermeldungen erzeugen, kann anhand dieser Kommentare leicht festgestellt werden, welche Eingabeanweisung für die Compiler-Fehlermeldung verantwortlich ist.

Ausführungszeit

Die Druckausgabe zur Ausführungszeit ist abhängig von den verwendeten Subsystemen. REGENT gibt lediglich Fehlernachrichten auf SYSPRINT aus, siehe Kapitel 13.

6.8

S Y S T E M - H A N D B U C H

Beispiel

```
CALL QQDBDD(QQ,'DUMP',SYSPRINT);
```

```
*****************************BASISDESCRIPTOR-DUMP*******************************
ID='DUMP';

BASEDESCRIPTOR-ADDR=              495728
DESCM.ADR=        495680                          DESCM.ENUMB=         2    DESCM.ESIZE=        12

DESCM.STR1='00000010'B

DESCM.STR2='00000000'B  DESCM.STR3='00000000'B  DESCM.STR4='00000000'B;
```

CALL QQDBDD

Syntax

CALL QQDBDD(QQ, text, printfile);

text: = char-expr.
printfile: = printfile-expr.

Erläuterung

QQDBDD druckt unter der Überschrift'text' alle definierten Basisdescriptoren mit ihren Attributen auf den Printfile.

Es bedeuten:

ADR : Adresse des Folgeelements

ENVMP : Zahl der Elemente im Folgeelement

ESIZE : Größe (in Bytes) eines Elements im Folgeelement

STR1 : BIT 1 ON DISK
2 REDEFINED WHILE ON DISK
3 WRITTEN INTO
4, 5, 6 } INTERNAL USE
7 UNRELEASED
8 HIGH PRIORITY

STR2 : BIT 1-3 STEP-CLASS
4-5 INTERNAL USE

STR3 : BIT 1-3 INTERNAL USE
4 ¬ DEFINED
5 ¬ ALLOCATED
6-8 DESCRIPTOR TYPE
000 KNOTEN
001 REF
010 BLATT

STR3 : INTERNAL USE

Beispiel:

```
CALL QQDPOL (QQ,'TEST1¦SYSPRINT);
```

CALL QQDPOL

Syntax

```
CALL QQDPOL (QQ, text, printfile);

     text::      = char (100) var-expr.
     printfile:: = printfile-expr.
```

Erläuterung

Es wird gemäß Fehlermeldung DDS502 eine Statistik mit der Überschrift 'text' auf den Printfile 'printfile' ausgegeben.
Siehe S.13-14.

Beispiel:

```
CALL QQDPSZ(QQ, 20000);
```

Syntax

```
CALL QQDPSZ (QQ, newsize);

     newsize:: = bin-fixed(31)-expr.
```

Erläuterung

Die DDS-Poolsize wird auf den Wert 'newsize' gesetzt. Ist die neue Poolsize kleiner als die aktuelle Poolgröße, so wird reorganisiert. Wird dabei die neue Größe nicht erreicht, so wird die aktuelle Größe statt 'newsize' angenommen.

Beispiel:

```
CALL QQDREO(QQ, 1, 1, 1);
```

CALL QQDREO

Syntax

CALL QQDREO(QQ, priority, blocktype, needed);

priority, blocktype, needed:: = bin-fixed(15) expr.

Erläuterung

QQDREO erlaubt die Änderung der Reorganisationsstrategie für den virtuellen Datenpool.

priority 0 : die DDS-Prioritäten werden nicht berücksichtigt

1 : die DDS-Prioritäten werden berücksichtigt

blocktype 0 : es werden nur Blätter ausgelagert

1 : es können alle DDS-Typen ausgelagert werden

needed 0 : die Reorganisation erfolgt unabhängig von dem jeweils aktuell benötigten Platz

1 : die Reorganisation wird abgebrochen, wenn der aktuell benötigte Platz frei geworden ist

Default ist CALL QQDREO(QQ,0,0,0);

Beispiel

CALL QQDSTR(QQ,BASE1, 'BASE1-STRU', SYSPRINT);

DCL BASE1 DESCRIPTOR;

```
NAME='BASE_STRU';
1.........  DEF  ALOC  KNOTEN  NUMB=  2  SIZE=  12  STEP=0  ALIGN=0  OFFS=0  PRTY=00000010
.2........  DEF  ALOC  KNOTEN  NUMB=  3  SIZE=  12  STEP=0  ALIGN=0  OFFS=0  PRTY=00000010
..3.......  DEF  ALOC  KNOTEN  NUMB=  4  SIZE=  12  STEP=0  ALIGN=0  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
..3.......  DEF  ALOC  KNOTEN  NUMB=  4  SIZE=  12  STEP=0  ALIGN=0  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
..3.......  DEF  ALOC  KNOTEN  NUMB=  4  SIZE=  12  STEP=0  ALIGN=0  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
.2........  DEF  ALOC  KNOTEN  NUMB=  3  SIZE=  12  STEP=0  ALIGN=0  OFFS=0  PRTY=00000010
..3.......  DEF  ALOC  KNOTEN  NUMB=  4  SIZE=  12  STEP=0  ALIGN=0  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
..3.......  DEF  ALOC  KNOTEN  NUMB=  4  SIZE=  12  STEP=0  ALIGN=0  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
..3.......  DEF  ALOC  KNOTEN  NUMB=  4  SIZE=  12  STEP=0  ALIGN=0  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
...4......  DEF ¬ALOC  BLATT   NUMB=  5  SIZE=   4  STEP=0  ALIGN=2  OFFS=0  PRTY=00000010
```

Syntax

```
CALL QQDSTR (QQ, base, text, printfile);

     base::      = basisdescriptor-ref.
     text::      = char (100) var-expr.
     printfile:: = printfile-expr.
```

Erläuterung

QQDSTR druckt mit der Überschrift 'text' die Struktur eines dynamischen Datenfeldes in PL/1-Art auf den angegebenen Printfile.

Dabei bedeuten:

DEF	:	Element ist definiert (z.B. durch DEFINE)
¬DEF	:	Element ist noch nicht definiert
ALOC	:	Element ist allokiert,d.h. es wurde auch bereits referiert
¬ALOC	:	Element ist definiert aber nicht allokiert
KNOTEN, BLATT	:	Element ist vom Typ Knoten oder Blatt
NVMB	:	Zahl der Elemente in einem Knoten oder Blatt
SIZE	:	Größe (in Bytes) eines Knoten- oder Blattelementes
STEP	:	Bereich, in den der in der STEP-Option des DEFINE angegebene Wert fällt.
ALIGN,OFFS	:	interne Größen
PRTY	:	BIT 1 ON DISK 2 REDEFINED WHILE ON DISK 3 BEEN STORED INTO 5,6 INTERNAL USE 7 UNRELEASED 8 HIGH PRIORITY

Beispiel:

```
CALL QQDSTX (QQ,'STRUCTURE-DUMP',SYSPRINT);
```

CALL QQDSTX

Syntax

CALL QQDSTX(QQ, text, printfile);

text:: = char-expr.
printfile:: = printfile-expr.

Erläuterung

QQDSTX druckt mit der Überschrift 'text' alle zum Zeitpunkt des Aufrufs definierten DDS-Strukturen gemäß QQDSTR auf dem angegebenen Printfile.

Als DDS-Identifikation wird die Kernspeicheradresse des Basisdeskriptors ausgedruckt (NAME =).

Ergänzend dazu werden alle Basisdeskriptoren gemäß QQDBDD gelistet.

Beispiel

```
BEGIN;
DCL QQMMAP ENTRY(PTR)EXT;
CALL QQMMAP(QQ);
END;
```

REGENT MODULE MANAGMENT POOLDUMP INFORMATION

SUMMARY OF RMM - MODULES USED IN CORE :

M_NAME	ENTRY_ADR	M_LENGTH	LOAD_CNT	LINK_CNT	D_RETRIEVE	USE_CNT	QUEUE
FOPLBELA	0A65A4	00011A68	0	0	1	1	IPQ
GILEAVE	0A5104	00000700	0	0	2	4	IPQ
GIPLOT	097A84	00005588	0	0	32	102	IPQ
GILTRAN	09D9C4	00003648	0	0	30	198	IPQ
GIALASS	079964	000016A8	0	0	28	114	IPQ
GIREENT	0A5904	00000708	0	0	2	3	IPQ
FOREADB2	0971CC	000008B8	0	0	1	1	IPQ
FOREADB1	076034	00000FD8	0	0	1	1	IPQ
GIC.SYMB	0C65F4	00009A18	0	0	2	3	IPQ
GICHAN	09D524	00003AE8	0	0	3	4	OPQ
FOPLGEOM	0A5804	00018808	0	0	1	1	OPQ
GISPEC	076E1C	000011F0	0	0	5	5	OPQ
GITEXT2	09709C	000009E8	0	0	3	3	OPQ
GIPOIN2	0A5034	000007D0	0	0	19	82	OPQ
GICOL62	076FB4	00001058	0	0	3	5	OPQ
GIPOLY	079E5C	000011B0	0	0	3	3	OPQ
GINPOL	07901C	00000E40	0	0	2	2	OPQ
GIARGTN	07776C	000008A0	0	0	13	52	OPQ
GILINE	07632C	00000CE0	0	0	23	26	OPQ
GIINIT	076E1C	000011F0	0	0	1	1	OPQ
FOINIT	097E54	0000A1B8	0	0	1	1	OPQ
FOREADGM	079134	00000ED8	0	0	1	1	OPQ
FOREADMH	09759C	000008B8	0	0	1	1	OPQ
FOPLOPEN	0A9574	0000AA98	0	0	1	1	OPQ
GIOPPL	07A2A4	00000B78	0	0	1	1	OPQ
GINEUZE	0BAFDC	00001030	0	0	1	1	OPQ

```
SPECIFIED RMM POOL SIZE            :  165000 BYTES
MAXIMUM POOL USED   FOR MODULES    :  165000 BYTES
MINIMUM POOL NEEDED FOR MODULES    :  161704 BYTES
LENGTH OF ACTIVE   PROGRAM QUEUE   :       0 BYTES
LENGTH OF INACTIVE PROGRAM QUEUE   :  163216 BYTES
```

CALL QQMMAP

Syntax

```
DCL QQMMAP ENTRY(PTR)EXT;
CALL QQMMAP(QQ);
```

Erläuterung

QQMMAP druckt zu einem bestimmten Zeitpunkt den Inhalt des RMM-Pools aus. Anhand der Liste ist zu ersehen, ob der jeweilige Module aktive, inaktive oder früher einmal aktiv war. Zusätzlich wird die Adresse innerhalb des Kernspeichers und die Modullänge ausgegeben. Beide Werte sind hexadezimal. Ferner werden Zähler aufgelistet, die die Anzahl der Plattenzugriffe für diesen Modul und die Anzahl der Benutzungen für diesen Modul angeben. Bei Anwendung im Anwenderprogramm muß für diese Prozedur die obige DCL ENTRY EXTERNAL-Anweisung angegeben werden.

Beispiel:

```
CALL QQMREO(QQ);
```

CALL QQMREO

Syntax

```
CALL QQMREO(QQ);
```

Erläuterung

Der Aufruf von QQMREO bewirkt eine Reorganisation des RMM-Pools (= REGENT-Modul-Managment Pools). Es werden bei dieser Aktion alle im Kernspeicher befindlichen, nichtaktiven Module gelöscht. Diese sind solche Module, die bei der momentanen REGENT-Aktivität nicht benutzt werden.

Beispiel

```
DCL QQSTIME ENTRY(BIN FOAT(21)) EXTERNAL;
DCL REFERENZ_VALUE BIN FLOAT(21);
REFERENZ_VALUE = 1000.;
CALL QQSTIME(REFERENZ_VALUE);
```

CALL QQSTIME

Syntax

```
DCL QQSTIME ENTRY(BIN FLOAT(21))EXTERNAL;
CALL QQSTIME(rf);
```

rf::= Referenzwert in Sekunden mit dem der Timer gesetzt wird, BIN FLOAT(21).

Erläuterung

QQSTIME ruft mit dem angegebenen Referenzwert das OS-Makro STIMER auf. Im Zusammenspiel mit der Routine QQTTIME ist eine Zeitmessung möglich (siehe Beispiel zur Routinenbeschreibung QQTTIME).

Beispiel:

```
DCL QQTTIME ENTRY(BIN FLOAT(21)) EXTERNAL;
DCL TIME_REMAINING  BIN FLOAT(21);
CALL QQTTIME(TIME_REMAINING);
```

Beispiel zur Zeitmessung

```
DCL QQTTIME ENTRY(BIN FLOAT(21)) EXTERNAL;
DCL QQSTIME ENTRY(BIN FLOAT(21)) EXTERNAL;
DCL REFERENZ_TIME BIN FLOAT(21);
DCL TIME_REMAINING BIN FLOAT(21);
DCL TIME_DIFFERENZ BIN FLOAT(21);
REFERENZ_TIME = 1000.;
CALL QQSTIME(REFERENZ_TIME);
     .
     .
CALL QQTTIME(TIME_REMAINING);
TIME_DIFFERENZ = REFERENZ_TIME - TIME_REMAINING;
```

CALL QQTTIME

Syntax

CALL QQTTIME(a);

a:: = verbliebene Restzeit in Sekunden seit dem letzten Aufruf von QQSTIME.

Erläuterung

QQTTIME liefert bei Aufruf einen verbliebenen Restwert an Zeiteinheiten in Sekunden. Die Routine QQTTIME benutzt das OS-Makro TTIMER.

Das untere Beispiel auf der linken Seite beschreibt den kompletten Vorgang der Zeitmessung unter Anwendung der QQSTIME und QQTTIME-Routinen.

<u>Beispiel</u>

```
DCL GI_PTR PTR;
ENTER SUB1;                       \
   :                               } 1)
ENTER GIPSY;                      /     \
   :                                     } 2)
END GIPSY LEAVE(GI_PTR);                /   \
   :                                         } 3)
ENTER GIPSY REENTER(GI_PTR);                /
END GIPSY;                                   } 4)
END SUB1;                                 } 5)
ENTER SUB2;                              \ 6)
   :                                      }
END SUB2;                                /
```

<u>Erläuterung</u>

1) Sprache von SUB1 ist gültig
2) SUB1 nicht unterbrochen aber Sprache von GIPSY ist gültig, GIPSY wird verlassen ohne es abzuschließen (durch LEAVE).
3) Sprache von SUB1 ist gültig
4) GIPSY wird weitergeführt, Sprache von GIPSY ist gültig, dann wird GIPSY abgeschlossen.
5) Sprache von SUB1 ist gültig, danach wird SUB1 abgeschlossen
6) SUB2 wird aufgerufen, Sprache von SUB2 ist gültig bis zum Abschluß durch die END-Anweisung

Syntax

ENTER subsystemname [sonstiges];

Erläuterung

Mit der ENTER-Anweisung wird ein Subsystem aufgerufen. "subsystemname" ist der Name des Subsystems, bei abkürzbaren Namen sind nur die ersten signifikanten Buchstaben erforderlich. Von ENTER bis zur zugehörigen END-Anweisung ist die Subsystemsprache gültig.

Die Bedeutung von "sonstiges" hängt vom Subsystem ab und wird im betreffenden Subsystem-Handbuch beschrieben.

Anmerkung für Subsystemersteller:

"sonstiges" wird in der INITIAL CLAUSE abgearbeitet.

Beispiel

1) Siehe Beispiel zu ENTER vorige Seite

2) Falsch:

```
ENTER  SUB1;
ENTER  SUB2;
END  SUB1;
END  SUB2;
```

Richtig:

```
ENTER  SUB1;
ENTER SUB2;
END  SUB2;
END SUB1;
```

Erläuterung:

Bei dem geschachtelten Aufruf von Subsystemen ist die Einhaltung der Blockstruktur zu beachten. Mit Hilfe von END LEAVE und REENTER können trotzdem parallel Subsysteme betrieben werden, wenn der Subsystemersteller dies vorgesehen hat.

Syntax

END subsystemname [sonstiges] ;

Erläuterung:

Mit der END-Anweisung wird ein Subsystem abgeschlossen, "subsystemname" ist der Name des Subsystems, wenn er abkürzbar ist, müssen nur die signifikanten Zeichen angegeben werden. Die END-Subsystemanweisung kann nur paarweise mit der zugehörigen ENTER-Anweisung stehen. ENTER-END-Paare können geschachtelt werden, dabei ist die Blockstruktur einzuhalten.

Die Bedeutung von "sonstiges" hängt vom Subsystem ab. Wenn der Subsystemersteller dies vorsieht, kann durch END name LEAVE ein Subsystem nur unterbrochen werden, es wird durch ENTER name REENTER wieder weitergeführt.

Anmerkung für Subsystemersteller:

Die Abarbeitung von "sonstiges" geschieht in der END CLAUSE.

Beispiel

```
      .
      .
      .
      .
      .

1) MESSAGE ACTIVE DEBUG;

   FINISH;

   END;

2) ON ERROR BEGIN
       ON ERROR SYSTEM;
       MESSAGE ACTIVE DEBUG;
       CALL PLIDUMP('T','FINISH');
       FINISH;
   END;
```

Erläuterung

1) Normaler Abschluß eines REGENT-Programms

2) Durch diese Anweisungen wird im Fehlerfalle die FINISH-Anweisung ausgeführt, die Statistiken werden gedruckt, ein kurzer PLIDUMP mit Programm-Aufruf-Trace wird gedruckt.

Syntax

FINISH;

Erläuterung

Mit FINISH wird das REGENT-System abgeschlossen, Kontrollblöcke werden gelöscht. Falls die MESSAGE-Klasse DEBUG aktiv ist, druckt die Modulverwaltung und die Datenverwaltung je eine Statistik aus- Die FINISH-Anweisung schließt außerdem die PLOT-Dateien ab, wenn der PLOT-Parameter verwendet wurde.

Daher ist bei PLOT unbedingt FINISH erforderlich.

Anmerkung:

Im nächsten REGENT-Release wird ein automatisches FINISH erzeugt.

Erläuterung der Statistiken:

Siehe Fehlernachrichten Kapitel 13, DDS502 und RMM050 bis RMM055, S.13-14, 13-15 und 13-59 bis 13-60

Beispiel

```
MESSAGE;
MESSAGE DEBUG TEXT('STATEMENT 103 AUSGEFÜHRT');


DCL X CHAR(10);
DCL Z;
GET LIST(X,Z);
MESSAGE TEXT('WERT VON  ' || X ||' IST '||Z);

MESSAGE SEVERE TEXT('MODUL FEHLT');
```

MESSAGE

Syntax

MESSAGE [level] [text] ;
level = D | I | W | E | S | T oder in der vollständigen Form:
DEBUG | INFORMATIVE | WARNING | ERROR |
SEVERE | TERMINAL

text::= TEXT (character-string-expression)

Erläuterung

1) Die Optionen level und text können in beliebiger Reihenfolge stehen oder fehlen

2) Standard für level ist I

3) Standard für den Text ist der Nullstring

4) Der Textausdruck muß sich in eine Zeichenkette von maximal 100 Zeichen Länge umwandeln lassen.

5) Diese Anweisung kann auch in einer Anweisungsdefinition (siehe Kap. 5) oder in einem Anwenderprogramm stehen.

Beispiel

MESSAGE ACTIVE DEBUG;

Nachrichten, die nur zu Testzwecken dienen, werden aktiviert.

MESSAGE
ACTIVE

Syntax

MESSAGE ACTIVE [level] * ;

Erläuterung

1) Wenn die LEVEL-Option fehlt, werden alle Nachrichtenlevel aktiviert; sonst werden die angegebenen Level aktiviert. Genauer: ihr Aktivierungsgrad wird um 1 erhöht.

2) Für level kann eine der gültigen level-Angaben stehen (D I ...); siehe MESSAGE-Anweisung.

3) Soweit der Aktivierungsgrad der Nachrichtenlevel nicht beeinflußt wird, sind D und I inaktiv (Grad 0), die übrigen aktiv (Grad 1).

4) Siehe MESSAGE INACTIVE

5) Diese Anweisung kann auch in einer Statement-definition (siehe Kap. 2) oder in einem Anwenderprogramm stehen.

Beispiel

```
MESSAGE COUNT (N_ERROR) ERROR;
```

Die Zahl der Nachrichten vom Level ERROR, die seit dem letzten Nullsetzen des ERROR-Zählwertes aufgetreten sind, werden in N_ERROR gespeichert.

```
MESSAGE RESET W;
DO I = 1 TO 500;
CALL UNTERPROGRAMM;
END;
MESSAGE COUNT(K) WARNING;
MESSAGE COUNT(K1) W TOTAL;
PUT LIST (K,K1);
```

Es wird gezählt, wieviele Nachrichten vom Level WARNING bei den 500 Aufrufen von UNTERPROGRAMM erzeugt wurden. Das Ergebnis wird in K gespeichert und ausgedruckt. Die Gesamtzahl der in diesem Lauf bisher überhaupt erzeugten Nachrichten vom Level WARNING wird in K1 gespeichert und gedruckt.

MESSAGE
COUNT

Syntax

MESSAGE COUNT(elementreference) level [TOTAL] ;

elementreference::= binary-fixed(15)-Variable

Erläuterung

1) In der Variablen, die durch elementreference definiert ist, wird der Stand der Nachrichtenzählung für den angegebenen Level gespeichert; und zwar

   ohne TOTAL: der Stand des mit MESSAGE RESET beeinflußbaren Zählwerks

   mit TOTAL: die vom Eintritt in das REGENT-System an laufende und nicht beeinflußbare Gesamtzählung

2) Für level muß eine der gültigen level-Angaben stehen (D I ...); siehe MESSAGE-Anweisung.

Beispiel

MESSAGE DUMP;
Nach allen Nachrichten wird ein Trace angegeben

MESSAGE DUMP ('TFH') T S E;
Nach Nachrichten vom Level TERMINAL, SEVERE, ERROR wird ein ausführlicher Dump erzeugt, jedoch ohne Kontrollblöcke.

MESSAGE
DUMP

Syntax

MESSAGE DUMP(parameter) [level] * ;
MESSAGE DUMP OFF [level] * ;

Erläuterung

1) Wenn die Level-Option fehlt, wird in der ersten Form(ON) für alle Nachrichtenlevel im Anschluß an das Drucken der folgenden Nachrichten auch ein PLIDUMP erzeugt. Bei der zweiten Form (OFF) wird diese Reaktion ausgeschaltet. Falls einzelne Level angegeben sind, werden nur diese beeinflußt.

2) Der Parameter steuert die Art des erzeugten Dumps in gleicher Weise wie der entsprechende Parameter der PL/1-Hilfsroutine PLIDUMP. Er muß die Form einer Zeichenkette haben.
   T bewirkt Trace-Ausgabe
   NT unterdrückt Trace-Ausgabe
   F bewirkt Ausgabe der File-Informationen
   B bewirkt Ausgabe der Kontrollblöcke
   H bewirkt einen Kernspeicherausdruck (hexadezimal)

3) Wenn die optionen fehlen, so wird nur ein Trace erzeugt. Ein Trace wird stets erzeugt, wenn bei MESSAGE DUMP nicht ausdrücklich NT angegeben wurde.

Beispiel

```
MESSAGE INACTIVE;
MESSAGE ACT S T;
```

Zunächst werden alle Nachrichtenlevel inaktiviert, anschließend die Level SEVERE und TERMINAL wieder aktiviert.

MESSAGE
INACTIVE

Syntax

MESSAGE INACTIVE [level] * ;

Erläuterung

1) Wenn die LEVEL-Option fehlt, werden alle Nachrichtenlevel in ihrem Aktivierungsgrad um 1 erniedrigt; sonst nur die angegebenen Level.

2) Für level kann eine der gültigen level-Angaben stehen (D I ....); siehe MESSAGE-Anweisung

3) Soweit der Aktivierungsgrad der Nachrichtenlevel nicht beeinflußt wird, sind D und I inaktiv (Grad O), die übrigen aktiv (Grad 1)

4) Siehe MESSAGE ACTIVE.

5) Diese Anweisung kann auch in einer Anweisungsdefinition (siehe Kap. 2) oder in einem Anwenderprogramm stehen.

Beispiel

MESSAGE RESET;
Alle Nachrichtenzählwerke werden auf 0 gesetzt

MESSAGE RESET E S T;

Die Nachrichtenzählwerke für die Level ERROR, SEVERE und TERMINAL werden auf 0 gesetzt.

Syntax

MESSAGE RESET [level] * ;

Erläuterung

1) Wenn die Level-Option fehlt, werden alle Nachrichtenzählwerke auf 0 gesetzt, sonst nur die angegebenen Level.

2) Für level kann eine der gültigen Level-Angaben stehen (D I .....); siehe MESSAGE-Anweisung

3) Siehe MESSAGE COUNT

4) Diese Anweisung kann auch in einer Anweisungsdefinition (Siehe Kap. 2) oder in einem Anwenderprogramm stehen.

Beispiele

1) FALL1: PROC OPTIONS(MAIN)
REGENT(BANK);

2) FALL2: PROC OPTIONS(MAIN)
REGENT(LIST,PLOT,BANK,MPOOL=1000);

3) FALL3: PROC OPTIONS(MAIN)
REGENT(NOBANK, NODA);

Erläuterung

Bei 1) und 2) wird die Dateiverwaltung benötigt, bei 3) nicht.

REGENT(BANK)

(NOBANK)

Syntax

REGENT(.... { BANK / NOBANK } ....)

Erläuterung:

Parameter der REGENT-Option auf der PROCEDURE-Anweisung des Anwenderprogramms.

BANK: Die REGENT-Dateiverwaltung wird benötigt.

NOBANK: Die REGENT-Dateiverwaltung wird nicht benötigt.

Die Angabe von BANK, wenn die Dateiverwaltung nicht erforderlich ist, führt zu größerem Speicherplatzbedarf.
Die Angabe von NOBANK, obwohl die Dateiverwaltung erforderlich ist, führt zu einer Fehlermeldung.

Beispiele

```
1)   DA1:   PROCEDURE OPTIONS(MAIN) REGENT;

2)   DA2:   PROC OPTIONS(MAIN)
               REGENT(DARRAYS,DPOOL=10000);

3)   DA3:   PROC OPTIONS(MAIN)
               REGENT(LIST,DA,PLOT);

4)   NODA4:  PROC OPTIONS(MAIN)
                  REGENT(NODA);
```

Erläuterung

In Beispiel 1) bis 3) wird die Verwaltung für dynamische Datenstrukturen benötigt, in Beispiel 4) nicht. In Beispiel 1) wird die Datenverwaltung verfügbar gemacht ohne besondere REGENT-Option, da DARRAYS der Standardwert ist.

REGENT(DARRAYS)

(NODA)

Syntax

REGENT(...{DARRAYS / NODA}....)

Erläuterung

Parameter der REGENT-Option auf der PROCEDURE-Anweisung des Anwenderprogramms.

DARRAYS: Die REGENT-Datenverwaltung für dynamische Datenfelder wird benötigt.

NODA: Die REGENT-Datenverwaltung wird nicht benötigt.

Die Angabe von DA, wenn die Datenverwaltung nicht erforderlich ist, führt zu größerem Speicherplatzbedarf. Die Angabe von NODA, obwohl die Datenverwaltung erforderlich ist, führt zu einer Fehlermeldung.

Beispiel

```
1)    P1:   PROC REGENT(DPOOL=10000) OPTIONS(MAIN);

2)    P2:   PROCEDURE OPTIONS(MAIN)
               REGENT(PLOT,BANK,DA,DPOOL=100000,LIST);

3)    P3:   PROC OPTIONS(MAIN)
               REGENT(NODA,DPOOL=1000);
```

Erläuterung

In 1) wird die Poolsize für die Datenverwaltung auf 10000 gesetzt, in 2) auf 100000.

In Beispiel 3) wird die DPOOL-Angabe ignoriert, da die Datenverwaltung durch NODA ausgeschaltet ist.

Syntax

REGENT(....,DPOOL=poolsize,....)

Erläuterung

Parameter der REGENT-Option auf der PROCEDURE-Anweisung des Anwenderprogramms.

poolsize: Größe des dynamischen Datenbereiches der Datenverwaltung in Bytes.

Wurde außerdem NODA angegeben, wird der Parameter DPOOL ignoriert. Standardwert für "poolsize" ist 100000.

Beispiele

```
1)   P1:   PROC OPTIONS(MAIN)REGENT;

2)   P2:   PROC OPTIONS(MAIN)REGENT(INIT);

3)   *SUBSYSTEM ABC;
     *MODULE A1;
     *PROCESS REGENT;
      A1: PROC(X,Y,Z) REGENT(SUB=ABC,INIT);
       ENTER GIPSY;
        .
        .
        .
4)   *SUBSYSTEM ABC;
     *MODULE A2;
     *PROCESS REGENT;
      A2: PROC REGENT(SUB=ABC);
        .
        .
        .
5)    P5: PROC REGENT (NOINIT,BANK,PLOT);
```

Erläuterung

Beispiele 1) und 2) sind identisch, normale Anwenderprogramme. In Beispiel 3) und 4) werden die POL-Module A1 und A2 für das Subsystem ABC erzeugt, in den Routinen können andere Subsysteme aufgerufen werden. Bei 4) wird NOINIT durch SUB=ABC impliziert. In Beispiel 5) wird eine externe POL-Prozedur erzeugt, die den Laufzeitzustand von außen übernimmt (wegen NOINIT). Die Parameter BANK und PLOT werden ignoriert.

REGENT(INIT)

(NOINIT)

Syntax

REGENT(....{INIT / NOINIT}.....)

Erläuterung

Nur wichtig bei der Subsystementwicklung.
Parameter der REGENT-Option auf der PROCEDURE-Anweisung eines Anwenderprogramms oder eines Moduls, der in einer Subsystemsprache geschrieben ist. Ein Hauptprogramm benötigt die Option INIT, das ist der Standardwert für Anwenderprogramme. Ein POL-Modul muß mit NOINIT übersetzt werden. Ist die REGENT-Option SUBSYSTEM oder NOSUBSYSTEM vorhanden, wird NOINIT angenommen, da diese Parameter ebenfalls nur für POL-Module angegeben werden können.

INIT : Das REGENT-Laufzeitsystem wird zu Beginn des Programms gestartet und initialisiert.

NOINIT : Der Zustand des REGENT-Laufzeitsystems wird vom rufenden Programm in den Modul übernommen. Bei NOINIT werden daher die Parameter PLOT/NOPLOT, MOD/NOMOD, DA/NODA, BANK/NOBANK, MPOOL, DPOOL ignoriert.

Beispiele

```
1)   P1:   PROC REGENT(LIST) OPTIONS(MAIN);

2)   P2:   PROC REGENT(NOINIT, SUB=ABC);

3)   P3:   PROC OPTIONS(MAIN) REGENT(LIST=LISTE);

4)   P4:   PROC REGENT(NOLIST,NOINIT);
```

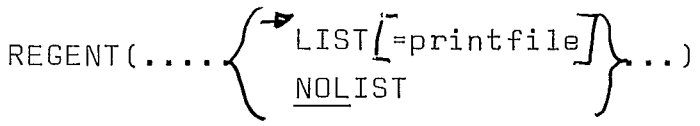
Erläuterung

In Beispiel 1) bis 3) wird die Programmliste ausgegeben, bei 1) und 2) auf SYSPRINT, bei 3) auf LISTE.
In Beispiel 4) wird die Programmliste des REGENT-Übersetzers unterdrückt.

REGENT(LIST)

(NOLIST)

Syntax



Erläuterung

Parameter der REGENT-Option auf der PROCEDURE-Anweisung eines Anwenderprogramms oder eines POL-Moduls.

LIST : Das POL-Programm wird vom REGENT-Übersetzer gelistet, die Ausgabe erfolgt auf SYSPRINT.

LIST = printfile:
Das POL-Programm wird auf die Datei mit dem DD-Namen "printfile" ausgedruckt. Eine gültige DD-Karte muß vorhanden sein.

NOLIST: Die Programmliste wird nicht ausgegeben.

Die Listen enthalten Anweisungsnummern und evtl. Fehlermeldungen. Die Anweisungsnummern finden sich im erzeugten PL/1-Programm in Kommentar wieder. Fehlermeldungen werden auch bei NOLIST auf SYSPRINT ausgegeben.

Beispiele

1) P1: PROC OPTIONS(MAIN) REGENT;

2) P2: PROC OPTIONS(MAIN) REGENT(MOD);

3) P3: PROC OPTIONS(MAIN) REGENT(NOMOD);

Erläuterung

In Beispiel 1) und 2) kann die Modulverwaltung benutzt werden, in 3) nicht.

REGENT(MOD)

(NOMOD)

Syntax

REGENT(...{ → MOD / NOMOD }...)

Erläuterung

Parameter der REGENT-Option auf der PROCEDURE-Anweisung des Anwenderprogramms.

MOD: Die REGENT-Modulverwaltung wird benötigt.

NOMOD: Die REGENT-Modulverwaltung wird nicht benötigt.

Die Angabe von MOD, wenn die Modulverwaltung nicht erforderlich ist, führt zu größerem Speicherplatzbedarf.
Die Angabe von NOMOD, obwohl die Modulverwaltung erforderlich ist, führt zu einer Fehlermeldung.
Die Angabe von NOMOD dürfte nur in seltenen Ausnahmefällen sinnvoll sein.

Beispiel

```
1)   P1:  PROC OPTIONS(MAIN) REGENT(MPOOL=10000);
2)   P2:  PROC OPTIONS(MAIN) REGENT(MPOOL=50000);
3)   P3:  PROCEDURE REGENT(NOMOD,MPOOL=1000)OPTIONS(MAIN);
```

Erläuterung

In Beispiel 1) wird die Poolsize für die Module auf 10000 gesetzt, in 2) auf 50000 Bytes.
In 3) wird die Angabe MPOOL ignoriert, da die Modulverwaltung durch NOMOD ausgeschaltet ist.

Syntax

REGENT(...,MPOOL=poolsize,...)

Erläuterung

Parameter der REGENT-Option auf der PROCEDURE-Anweisung eines Anwenderprogrammes. Mit MPOOL wird der Speicherplatzbereich für das dynamische Laden von Moduln festgelegt. "poolsize" ist die Größe des Modulpools in Bytes, Standardwert ist 100000.Nur im Notfall benutzt die Modulverwaltung mehr Speicherplatz zum dynamischen Laden von Moduln.
Bei NOMOD wird MPOOL ignoriert.

Beispiele

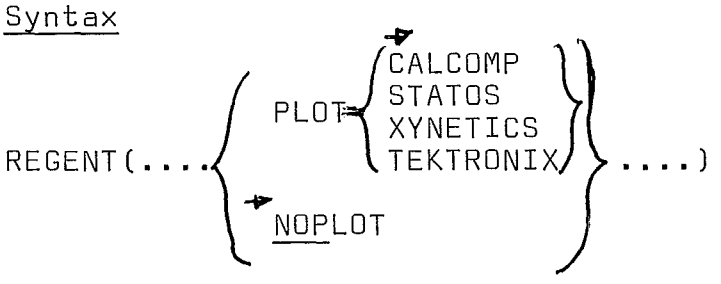
```
1)  P1:   PROC OPTIONS(MAIN) REGENT(PLOT);
2)  P2:   PROCEDURE REGENT(PLOT=STATOS)OPTIONS(MAIN);
3)  P3:   PROCEDURE REGENT(NODA,PLOT=XYNETICS,NOBANK)OPTIONS(MAIN);
4)  P4:   PROC OPTIONS(MAIN) REGENT(PLOT=TEKTRONIX);
5)  P5:   PROC OPTIONS(MAIN) REGENT;
6)  P6:   PROC OPTIONS(MAIN) REGENT(NOPLOT,NODA);
```

Erläuterung

In Beispiel 1) wird die Zeichenausgabe auf den Calcomp-Plotter gelenkt, in 2) auf den Statos-Plotter, in 3) auf den Xynetics-Zeichentisch.
Beispiel 4) kann nur interaktiv an einem Tektronix-Bildschirm ausgeführt werden. In den Beispielen 5) und 6) kann keine Zeichenausgabe erzeugt werden.

REGENT(PLOT)
(NOPLOT)

Syntax



Erläuterung

Parameter der REGENT-Option auf der PROCEDURE-Anweisung des Anwenderprogramms. Es wird die REGENT-Plotschnittstelle ein- oder ausgeschaltet, das gewünschte graphische Ausgabegerät kann angegeben werden.

NOPLOT: Keine Zeichenausgabe in diesem Job.

PLOT=plotter: Zeichenausgabe ist in diesem Job möglich. Das gewünschte Zeichengerät: Calcomp, Statos oder Xynetics kann angegeben werden. PLOT-TEKTRONIX erzeugt Ausgabe am Tektronix 4014-Bildschirmgerät, wenn das Anwenderprogramm interaktiv unter TSO läuft. Für Stapel-Jobs muß eine gültige PLOTTAPE-DD-Karte eingefügt werden.

Diese Angaben sind abhängig von der Installation

Wird in einem Job mit NOPLOT die Zeichenschnittstelle angesprochen, erfolgt eine Fehlermeldung.
PLOT erfordert als letzte Programmanweisung die FINISH-Anweisung (siehe 6-39).

Beispiele

1) siehe Beispiele zu REGENT(NOINIT)

2)
```
*SUBSYSTEM SUB1;
*MODULE SPLOT;
*PROCESS REGENT;
 SPLOT: PROC(x,y,z) REGENT(SUB=SUB1);
   DCL (x,y,z)...;
     :
   ENTER GIPSY REENTER(GI_PTR);
     :
     :   Plot Anweisungen mit GIPSY
   END GIPSY LEAVE(GI_PTR);
 END SPLOT;
```

3)
```
MAIN: PROC OPTIONS(MAIN) REGENT;
   :
  ENTER SUB2;
     :
     DCL PP ENTRY(PTR,....)EXT;
     CALL PP(QQ,...);
  END SUB2;
END MAIN;

*PROCESS;
PP: PROC(QQ,...)REGENT(SUB=SUB2);
 :
 :     Sub2-Anweisungen
END PP;
```

Erläuterung

Beispiel 2) zeigt die Erzeugung eines Moduls Subsystem SUB, der mit Hilfe von GIPSY Zeichenausgabe erzeugt.
Beispiel 3) zeigt die Benutzung einer externen POL-Prozedur PP, die aus dem Hauptprogramm MAIN aufgerufen wird. Da sie im Subsystem SUB2 aufgerufen wird, muß sie auch den Parameter SUB=SUB2 besitzen.

REGENT(SUBSYSTEM)

(NOSUBSYSTEM)

Syntax

REGENT(....{ SUBSYSTEM=subname / → NOSUBSYSTEM })

subname: Name eines existierenden Subsystems

Erläuterung

Parameter der REGENT-Option auf der PROCEDURE-Anweisung einer externen POL-Prozedur oder eines Moduls, der in der Sprache eines anderen Subsystems geschrieben ist. "subname" bezeichnet den Namen des Subsystems, von dem diese Prozedur bzw. dieser Modul aufgerufen wird. Der Name kann auch abgekürzt werden, wenn er abkürzbar ist. Diese Option ist nur sinnvoll bei NOINIT, daher wird bei SUB oder NOSUB die Option NOINIT angenommen.

In den meisten Fällen ist diese Option erforderlich bei der Erstellung von POL-Moduln für ein Subsystem während der Subsystem-Entwicklung.

KAPITEL 7

Empfehlungen für den Subsystemersteller

7.1 Planung

Obwohl es sich eigentlich von selbst versteht, sei an dieser Stelle nochmals darauf hingewiesen, daß eine sorgfältige Planung für die Qualität und termingerechte Fertigstellung von Subsystemen unbedingte Voraussetzung ist.

7.2 Subsystemdatenstrukturen und POL-Module

REGENT erlaubt es, in Subsystemen Module aufzurufen, die in der Sprache anderer Subsysteme erstellt wurden. Wenn man nun ein Subsystem plant, das möglicherweise von einem anderen Subsystem aufgerufen werden wird, so sind einige Einschränkungen zu beachten, die sich aus folgenden Tatsachen ergeben:

Wenn ein PL/1-Programm, das nicht zum Main-modul gehört,

FILE-Konstanten Deklarationen oder
CONTROLLED-Variable (nicht als Parameter)

enthält, so treten unvorhersehbare Fehler auf.

Aus diesen Gründen sind diese Dinge in PLR nicht erlaubt. In der gegebenen Situation muß also der Subsystemersteller darauf achten, daß die Subsystemdatenstrukturen derartige Deklarationen nicht enthalten. Zweckmäßig erscheint folgende Lösung:

a) Grundsätzliche Vermeidung von CONTROLLED-Variablen

b) Die Benutzungsanleitung schreibt dem Subsystemanwender vor, die im Subsystem erforderlichen FILE-Konstanten zu deklarieren. Diese können dann als Variable in geeigneten Anweisungen vom Subsystem übernommen werden.

```
! Beispiel
!
! Ungeeignet ist:
!
!     DATA COMMON;
!         DCL 1,
!            2 #FILE FILE INIT( FILE);
!            2 #TITLE CHAR(8);
!         DCL #FILE FILE;
!     END DATA;
!     CLAUSE INITIAL;
!         EXECUTE;
!            #TITLE=NEXT_EXPRESSION;
!             OPEN FILE(#FILE) TITLE(#TITLE);
!         END EXECUTE;
!     END CLAUSE;
!
! Geeigneter_ist:
!
!     DATA COMMON;
!         DCL 1,
!            2 #FILE FILE,
!            2 #TITLE CHAR(8);
!     END DATA;
!     CLAUSE INITIAL;
!         DCL TITLE CHAR(10);
!         DCL NAME CHAR(8) VARYING;
!         NAME=NEXT_IDENTIFIER;
!         TITLE='''' || NAME || '''';
!         EXECUTE #FILE=NAME;;
!         IF ISEXPRESSION THEN
!            EXECUTE #TITLE=NEXT_EXPR;;
!         ELSE
!            EXECUTE #TITLE=TITLE;;
!         EXECUTE
!            OPEN FILE(#FILE) TITLE(#TITLE);;
!     END CLAUSE;
```

! Erläuterung
!
! In der ersten Form verlangt die Benutzungsanleitung die Angabe
! eines DD-Namens im Anschluß an den Subsystemnamen in der
! ENTER-Anweisung. In der zweiten Form muß auf der ENTER-Anwei-
! sung ein FILE-Name stehen (evtl. gefolgt von einem DD-Namen),
! der natürlich zuvor deklariert sein muß.
!
! Erste Form:
! ENTER subsystem 'DATEI';
!
! Zweite Form:
! DCL BAND FILE;
! ENTER subsystem BAND 'DATEI';
! oder
! DCL DATEI FILE;
! ENTER subsystem DATEI;

7.3 COMMON

Der COMMON ist in allen Teilen des Subsystems zugänglich. Er kann daher folgende Funktionen übernehmen:

- Kommunikation zwischen dem POL-Programm und den Modulen
- Kommunikation der Module untereinander

Es hat sich gezeigt, daß es wenigstens in größeren Subsystemen nicht zweckmäßig ist, diese Funktionen dadurch zu realisieren, daß alle Variablen unmittelbar in den COMMON aufgenommen werden. Dadurch wird der Änderungsdienst erschwert. Jede Änderung am COMMON bewirkt nämlich, daß alle bereits erstellten Subsystemmodule neu generiert werden müssen. Bewährt hat sich folgende Technik:

```
DCL  1  COMMON,
       2  Daten, die grundsätzlich überall im Subsystem
          verfügbar sein müssen,
       2  #(50) PTR INIT((50)(NULL()).
```

Über die Pointer # sind dann überall die über Zeigertechnik angebundenen BASED-Datenstrukturen zugänglich. Die zugehörigen Deklarationen werden über %INCLUDE des PL/1-Makroprozesses nur in diejenigen Teile des Subsystems aufgenommen, die unmittelbar mit diesen Daten arbeiten.

7.4 Verwendung von Prozeduren als Eingabe

In vielen Anwendungsfällen ist es zweckmäßig, dem Subsystemanwender die Angabe von Prozeduren zu erlauben. Ein Beispiel soll dies erläutern. In einem Subsystem SSS wird in einem Modul die Zustandsgleichung eines Stoffes benötigt. Sie soll wie folgt aufgerufen werden.

```
#DRUCK(DICHTE,TEMPERATUR)
```

Alle Variablen seien vom Typ BIN FLOAT(21). Eine zweckmäßige Realisierung ist die folgende:

7.4.1 COMMON

```
DCL  1  COMMON,
       .
       .
       .
     2  #DRUCK ENTRY(BIN FLOAT(21),
           BIB FLOAT(21))RETURNS(BIN FLOAT(21)),
       .
       .
       .
```

7.4.2 Macrozeitdatenstruktur

```
DCL   1  MACRO,
      2  NR_DRUCK PIC  '99' INIT(0);
```

7.4.3 Statement ZUSTANDSGLEICHUNG

```
STATEMENT 'ZUST.AND';
   NR_DRUCK=NR_DRUCK 1;
          DCL PROCNAME CHAR(8);
          PROCNAME='ZUST' ||NR_DRUCK;
   EXEC;
     PROCNAME:PROC(R,T)RETURNS(BIN FLOAT(21));
     DCL(R,T)BIN FLOAT(21);
     RETURN(NEXT_EXPRESSION);
     END PROCNAME;
     DRUCK=PROCNAME;
     END EXEC;
END STATEMENT;
```

7.4.4 Benutzungsanleitung

Syntax: ZUSTAND expression;

Semantik: expression ist ein Ausdruck in den Variablen R und T (und evtl. anderen Variablen), der den Druck als Funktion der Dichte R und Temperatur T berechnet. Alle Variablen sind BIN FLOAT(21).

7.4.5 Anwendungsbeispiel

```
ENTER SSS;
-------
-------
ZUSTAND DRUCK(R,T);
DRUCK:PROC(R,T)RETURNS(BIN FLOAT(21));
     DCL(R,T)BIN FLOAT(21);
     /* Eine Prozedur die DRUCK errechnet */
END DRUCK;
```

KAPITEL 8

REGENT und das Betriebssystem

8.1 Allgemeines

REGENT wurde auf einer Anlage vom Typ IBM/370 entwickelt. Die Tatsache, daß der Systemkern fast ausschließlich in PL/1 erstellt wurde, wird künftige Umstellungen auf andere Anlagen, die ebenfalls PL/1 unterstützen, erleichtern. Dennoch ist abzusehen, daß derartige Umstellungen zusätzlichen Aufwand erfordern, da der Systemkern - der ja das Interface zwischen den anlagenunabhängigen Subsystemen und der jeweiligen Anlage darstellt - in einigen Punkten von den Eigenheiten der Anlage abhängt, für die er bereitgestellt ist. Im folgenden sind diese Abhängigkeiten dargestellt. Dabei wurde gegliedert in

Hardwareeigenschaften
Betriebssystemkonzepte
Betriebssystemdatenstrukturen
Betriebssystemprogramme
Assemblersprache
PL/1-Sprachdefinition
PL/1-Compilereigenheiten
PL/1-Laufzeitsystem.

Diese Abhängigkeiten sind eigentlich für den REGENT-Anwender (sowohl für den Subsystementwickler wie auch erst recht für den Subsystemanwender) ohne Interesse. Es ist jedoch eine im Umgang mit ICES gemachte Erfahrung, daß in Ausnahmefällen, beispielsweise bei unerwarteten Fehlern, die Information über den Zusammenhang zwischen Systemkern und Rechenanlage von Nutzen sein kann.

8.2 Abhängigkeit von Hardware-Eigenschaften

8.2.1 Arbeitsspeicher

Für eine sinnvolle Benutzung von REGENT ist ein Arbeitsspeicher entsprechend etwa 240K Bytes (besser 300K) nötig.

8.2.2 Peripherie

Es muß mindestens ein Peripheriegerät für Direktzugriff verfügbar sein. Für permanente Dateien benötigt REGENT einen Peripheriespeicherplatz von mindestens 8 M Bytes (z.B.Platteneinheit Typ 2314 oder 3330).Zur Implementierung ist eine 9-Spur-Bandeinheit(800 bpi oder 1600 bpi) erforderlich.

8.2.3 Wortstruktur

An einigen Stellen wird von der Wortstruktur der Anlage Gebrauch gemacht. Dazu gehören die Merkmale

A: kleinste adressierbare Einheit = 1 Byte = 8 bit
B: 1 Wort = 4 Byte
C: 1 Byte = Speicherplatz für 1 alphanumerisches Zeichen
D: Alignment
E: Codes für Zahlendarstellung und Zeichen

Auf diese Merkmale wird wie folgt Bezug genommen:

REGENT-Bereich	Bezug	Merkmal
Dynamische Datenstrukturen, Datenbank	Behandlung von Daten als Zeichenkette oder Bitkette	A,B,C
	Wandeln von Bitstrings in BIN FIXED-Variable	E
PLS, PLR	Zeichenklassen des Scanners	E
PLR	Berechnung von Länge und Alignment von Datenaggregaten	A,B,C,D

8.3 Betriebssystemkonzepte

REGENT stützt sich auf einige Eigenschaften des Betriebssystems, um Fähigkeiten bereitzustellen, die über den Rahmen der mit PL/1 realisierbaren Fähigkeiten hinausgehen oder aber mit PL/1 nur umständlich bereitzustellen wären.

8.3.1 Programmbibliotheken

REGENT benutzt die Fähigkeit des Betriebssystems, private Programmbibliotheken bereitzustellen, aus denen heraus Programme mit Mitteln der Steuersprache aufgerufen werden können, wobei gleichzeitig diesen Programmen eine Zeichenkette als Parameter übergeben werden kann.

8.3.2 Unterprogrammbibliotheken

REGENT benutzt die Fähigkeit des Betriebssystems, private Unterprogrammbibliotheken bereitzustellen, aus denen heraus Unterprogramme bei der Erstellung neuer Programme durch das Bindeprogramm (Linkage Editor oder Loader) entweder automatisch (Automatic Library Call) oder durch besondere Anweisungen in das neue Programm eingebunden werden können.

8.3.3 Verketten von Programm- und Unterprogrammbibliotheken

REGENT benutzt die Fähigkeit des Betriebssystems, Programm- bzw. Unterprogrammbibliotheken zu verketten.

8.3.4 Dynamischer Aufruf von Programmen

REGENT benutzt die Fähigkeit des Betriebssystems, Programme, die in verschiedenen Programmbibliotheken lagern, aus anderen Programmen heraus dynamisch mit Argumenten aufzurufen, auch ausdrücklich in den Arbeitsspeicher zu laden, dann aufzurufen und wieder zu löschen. REGENT verläßt sich darauf, daß das Betriebssystem Speicherplatz, der nach Beendigung oder Löschen derart dynamisch gerufener Programme frei wird, nutzbringend wieder verwendet.

8.3.5 Kennzeichnung von Programmen und Unterprogrammen

REGENT geht davon aus, daß Programme und Unterprogramme durch Zeichenketten von bis zu 8 Zeichen gekennzeichnet werden.

8.3.6 Partitioned Data Set

REGENT benutzt (im Subsystem PLS) die Fähigkeit des Betriebssystems, innerhalb einer Betriebssystemdatei eine weitere Unterteilung (mit Kennzeichnungen von bis zu 8 Zeichen Länge) in sequentielle Teildateien (Member) vorzusehen, solche Teildateien zu erzeugen, zu prüfen, zu ändern und zu löschen.

8.3.7 Reservieren von Dateien

REGENT benutzt die Fähigkeit des Betriebssystems, Dateien zeitweise für den exklusiven Zugriff durch ein laufendes Programm zu reservieren.

8.4 Betriebssystemdatenstrukturen

Folgende Datenstrukturen des Betriebssystems werden in REGENT benutzt:

Datenstruktur	Verwendung
Data Control Block (DCB)	Initialisieren einer Bank Initialisieren eines Bereiches bei dynamischen Datenstrukturen dynamischer Modulaufruf aus Privatbibliotheken
Data Set Control Block (DSCB)	Bearbeiten von Partitioned Data Sets Vervollständigen von DD-Informationen
Task Input Output Table (TIOT) Unit Control Block (UCB)	Reservieren einer Datei für exklusiven Gebrauch
Job File Control Block (JFCB)	Ergänzen einer unvollständigen Dateibeschreibung durch Angaben der DD-Karte
Directory eines Partitioned Dataset	Suche eines Moduls beim dynamischen Programmaufruf Löschen von Subsystemen
Object-Modulstruktur (ESD, TXT, RLD, END)	Modulgenerator

8.5 Betriebssystemprogramme

REGENT kann derzeit unter folgenden Betriebssystemen eingesetzt werden:

OS/MVT ab Release 21.0
OS/MVS ab Release 3.
OS/VS1 ab Release 5.0

Folgende Betriebssystemprogramme werden entweder mit Hilfe der Steuersprache oder dynamisch aus anderen Programmen aufgerufen:

Programm	Verwendung
PL/1-Optimizing-Compiler	Modulgenerator Subsystem PLS Implementierung REGENT-Prozedur
OS/360-Linkage-Editor (mindestens die 88k-Version)	Modulgenerator Implementierung Subsystem PLS
OS/360-Loader (ersetzbar durch Linkage Editor)	Implementierung REGENT-Prozedur
Assembler-Compiler	Implementierung
Utilities IEBCOPY	Subsystem PLS, Modulgenerator
IEBUPDTE	Implementierung
IEBGENER	Implementierung

8.6 Assemblersprache

Einige Teile des REGENT-Systemkerns sind als Assemblerprogramme erstellt. Dies ist entweder notwendig, um auf Fähigkeiten zuzugreifen, die PL/1 nicht verfügbar machen kann, oder zweckmäßig, um größere Effektivität zu erreichen. In diesen Assemblerprogrammen wird auf Betriebssystemfähigkeiten über folgende Makroaufrufe zugegriffen:

OPEN
CLOSE
LINK } für LINK auf Loadmodule die sich auf allgemeinen oder speziellen Bibliotheken befinden

EXTRACT ENQ RESERVE DEQ	Reservieren und Freigeben von Dateien auf einer spezifischen Einheit befinden
STOW DCBD FIND WRITE CHECK READ	zum Verarbeiten von Members eines Partitioned Data Sets Zugriffsmethode BPAM
BLDL LOAD DELETE GETMAIN FREEMAIN	für das Modul-Management
OBTAIN CAMLIST	zum Feststellen des freien Platzes eines Data Sets
RDJFCB RETURN WTO SNAP TIME ABEND DCB DCBS TTIMER STIMER	werden an verschiedenen Stellen zum Teil mehrmals benutzt

8.7 PL/1-Sprachdefinition

Im folgenden sind einige Fähigkeiten aufgelistet, die evtl. nicht in allen PL/1-Versionen verfügbar sein werden, die aber in REGENT benutzt werden:

8.7.1 Parameter

Benutzt wird die Fähigkeit, alle Datentypen - insbesondere auch ENTRY, FILE und CONTROLLED-Variable - als Parameter an Unterprogramme zu übergeben, unabhängig davon, ob sie einzeln oder in einem Datenaggregat vereinbart sind. Benutzt wird ferner die Möglichkeit, in Hauptprogramme einen Parameter in Form einer Zeichenkette zu übernehmen.

8.7.2 Direktzugriffdateien

Benutzt werden Dateien vom Typ REGIONAL(1) und REGIONAL(3), wobei die Satzlänge und die Dateilänge aus Angaben des Betriebssystems übernommen werden.

8.8 PL/1-Compilereigenheiten

Die Fähigkeiten des PL/1-Optimizing-Compilers werden ausgenutzt. Auch der Checkout-Compiler kann benutzt werden.

8.9 PL/1-Laufzeitsystem

In einigen Punkten besteht eine Abhängigkeit zwischen REGENT und dem Laufzeitsystem von PL/1. (Unter Laufzeitsystem versteht man die Unterprogramme und internen Datenstrukturen, die für den Ablauf eines PL/1-Programms notwendig sind und ohne Wissen des Programmierers eingebaut werden.)

8.9.1 PL/1-Laufzeitprogramme

REGENT geht davon aus, daß mindestens folgende PL/1-Laufzeitprogramme die Kontrolle nach einem Aufruf wieder genau an die Stelle zurückgeben, von der sie aufgerufen wurden:

IBMBOCLA	IBMBOCLB	IBMBOCLC	IBMBOCLD
IBMBKDMA	IBMBCACA	IBMBCCCB	IBMBCCCC
IBMBCCCA	IBMBCWZH	IBMBSAOA	IBMBSEDA
IBMBSEDB	IBMBSPLA	IBMBSPLB	IBMBSPLC
IBMBCVDY	IBMBBCIA	IBMBSEOA	IBMBCBCA
IBMBCCSA	IBMBCTHD	IBMBCTHX	IBMBCTHF
IBMBCTHP	IBMBCTHE	IBMBCUIX	IBMBCUID
IBMBCUIP	IBMBCUIE	IBMBCUIF	IBMBPAFA
IBMBPAFB	IBMBSISA	IBMBSISB	IBMBSIST
IBMBCKDP	IBMBCKZP	IBMBCKDD	IBMBCKZD
IBMBSBOA	IBMBSFOA	IBMBSXCA	IBMBSXCB
IBMBSXCC	IBMBSXCD	IBMBBGKA	IBMBBGKB
IBMBBGKC	IBMBCEZB	IBMBCEDB	IBMBCEFX
IBMBCEZX	IBMBCEDX	IBMBCEZF	IBMBCEDF
IBMBCCAA	IBMBCCAR	IBMBCGZA	IBMBCODE
IBMBCODP	IBMBCOZE	IBMBRIOA	IBMBRIOB
IBMBRIOC	IBMBRIOD	IBMBCPBD	IBMBCPBP
IBMBCPBE	IBMBCPBX	IBMBCPBF	IBMBAMMA
IBMBAMMB	IBMBAMMC	IBMBAMMD	IBMBCHFE
IBMBCHFP	IBMBCHXY	IBMBCHFH	IBMBCHXD
IBMBCHXF	IBMBSDOD	IBMBSDOC	IBMBSDOB
IBMBSDOA	IBMBSDOT	IBMBSLOA	IBMBSLOB
IBMBSIOA	IBMBSIOB	IBMBSIOC	IBMBSIOD
IBMBSIOE	IBMBSIOT	IBMBCWDH	IBMBCVZY
IBMBCHXE	IBMBCHXP	IBMBCHFY	IBMBCHXH
IBMBCHFD	QQDUMMYV		

Ferner werden im PLR-Precompiler einige PL/1-Laufzeitprogramme zur Platzersparnis durch Anweisungen an den Linkage-Editor in Overlay-Segmente gelegt. Es sind dies:

IBMBSIO1	IELCGOA	IBMBSFO1
IBMBMDS1	IELCGOC	IBMBSPL1
IBMBSBO1	IBMBSXC1	IBMBSXC1
IELCGOB	IBMBSEO1	IBMBCBC1
IBMBSED1	IBMBSAO1	IBMBCAC1
		IBMBCCS1

Hinsichtlich aller anderen PL/1-Laufzeitprogramme setzt REGENT voraus, daß sie keine Daten intern speichern (m.a.W., daß sie REENTRANT sind). Außerdem wird die Routine IBMBRIOA in besonderer Weise benutzt. REGENT setzt voraus, daß eine PL/1-Anweisung WRITE FILE(--) FROM(variable); in einen Unterprogrammaufruf dieser Routine übersetzt wird. Ab Optimising Compiler Version 1, Release 3.0 wird diese letzte Funktion durch eine entsprechende PL/1-Builtin-Function (STORAGE) realisiert.

8.9.2 PL/1-Laufzeit-Datenstrukturen

In den Assemblerprogrammen wird (mit wenigen Ausnahmen) dieselbe Konvention hinsichtlich der Daten- und Kontrollübergabe zwischen Unterprogrammen benutzt wie in PL/1. Ferner werden folgende PL/1-Laufzeit-Datenstrukturen benutzt:

Declare Control Block (DCLCB), File Control Block (FCB), Task Communication Area (TCA), Pseudoregistervektor (PRV), Major Free Area (MFA), Free Area Chain (FAC), Record Descriptor (RD).

Ferner werden grundsätzlich die Konventionen von PL/1-Programmen für den Unterprogrammaufruf und die Argumentübergabe eingehalten.

8.10 JCL-Prozeduren für das REGENT-System

REGENT läßt sich am einfachsten benutzen, wenn man die für die einzelnen Aufgaben erforderlichen Programmaufrufe und Dateibeschreibungen in der Job Control Language als katalogisierte Prozeduren zusammenfaßt. Auf der Rechenanlage Kernforschungszentrums Karlsruhe stehen folgende Prozeduren zur Verfügung:

REGENT	Übersetzung und Ausführung von REGENT-Programmen. Anwendung der REGENT-eigenen Subsysteme DABAL und PLS.
QQPCL	Erzeugen von Moduln und Routinen mit dem Modulgenerator
QQINTER	Benutzung des REGENT-Interpretierers für die Implementierung des Gesamtsystems.

```
//REGENT      PROC RUNIT=3330,RVOL=REGICE,RDSN=REGENT,
//            PLIB=REGENT,PUNIT=3330,PVOL=REGICE,
//            BLKF=968,BLKV=2004,DISK=SYSDA,
//            CPARM='C,SYN,INCLUDE',GPARM='PRINT,MAP/ISASIZE(40K)'
//P           EXEC PGM=PLSTRAN,PARM='ISA(40K)'
//COMFILE     DD   UNIT=&DISK,DISP=(NEW,PASS),
//            DCB=(BLKSIZE=800,LRECL=80,RECFM=FB,DSORG=PO),
//            SPACE=(1600,(1200,100,20))
//PLIDUMP     DD   SYSOUT=A,DCB=(BLKSIZE=&BLKV,LRECL=125,RECFM=VBA)
//PLIFILE     DD   UNIT=&DISK,DISP=(NEW,PASS),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120),SPACE=(TRK,(10,10))
//QQCLIB      DD   DSN=&PLIB..PLSTRAN.MODS,UNIT=&PUNIT,VOL=SER=&PVOL,
//            DISP=SHR
//            DD   DSN=&RDSN..PLSTRAN.MODS,UNIT=&RUNIT,VOL=SER=&RVOL,
//            DISP=SHR
//STEPLIB     DD   DSN=&RDSN..MODS,UNIT=&RUNIT,VOL=SER=&RVOL,DISP=SHR
//SUBLIB      DD   DSN=&RDSN..PLSTRAN.DATA,UNIT=&RUNIT,VOL=SER=&RVOL,
//            DISP=SHR
//SYSPRINT    DD   SYSOUT=A,DCB=(BLKSIZE=&BLKV,LRECL=125,RECFM=VBA)
//C           EXEC PGM=IEL0AA,PARM='&CPARM',COND=(9,LT)
//QQMAC       DD   DSN=&RDSN..PLRMAC,UNIT=&RUNIT,VOL=SER=&RVOL,DISP=SHR
//SYSIN       DD   DSN=*.P.PLIFILE,UNIT=&DISK,DISP=(OLD,DELETE)
//SYSLIN      DD   UNIT=&DISK,SPACE=(3200,(500)),
//            DCB=(BLKSIZE=3200,RECFM=FB),DISP=(NEW,PASS)
//SYSPRINT    DD   SYSOUT=A,DCB=(BLKSIZE=&BLKV,LRECL=125,RECFM=FBA)
//SYSUT1      DD   UNIT=&DISK,SPACE=(3303,(150)),DCB=BLKSIZE=3303
//SYSUT3      DD   UNIT=&DISK,SPACE=(3303,(100)),DCB=BLKSIZE=3303
//G           EXEC PGM=LOADER,COND=(9,LT),PARM='&GPARM'
//COMFILE     DD   DSN=*.P.COMFILE,DISP=(OLD,DELETE)
//FT06F001    DD   SYSOUT=A,DCB=(BLKSIZE=1955,LRECL=133,RECFM=FBA)
//PLIDUMP     DD   SYSOUT=A,DCB=(BLKSIZE=&BLKV,LRECL=125,RECFM=VBA)
//PLIOBJ      DD   DISP=(NEW,DELETE),SPACE=(800,(1200,100)),
//            DCB=(BLKSIZE=800,LRECL=80,RECFM=FB),UNIT=&DISK
//PLOTWK01    DD   UNIT=&DISK,SPACE=(CYL,(10,5)),DISP=(NEW,DELETE)
//PLOTWK02    DD   UNIT=&DISK,SPACE=(CYL,(10,5)),DISP=(NEW,DELETE)
//PLSLIB      DD   DSN=&PLIB..PLSTRAN.MODS,UNIT=&PUNIT,VOL=SER=&PVOL,
//            DISP=SHR,DCB=BLKSIZE=6447
//QQDDSF      DD   UNIT=&DISK,DCB=(DSORG=DA,KEYLEN=6,BLKSIZE=1000),
//            SPACE=(CYL,(10)),DISP=(NEW,DELETE)
//STEPLIB     DD   DSN=&PLIB..MODS,UNIT=&PUNIT,VOL=SER=&PVOL,DISP=SHR
//            DD   DSN=&RDSN..MODS,UNIT=&RUNIT,VOL=SER=&RVOL,DISP=SHR
//SUBLIB      DD   DSN=&RDSN..PLSTRAN.DATA,UNIT=&RUNIT,VOL=SER=&RVOL,
//            DISP=SHR
//SYSLIB      DD   DSN=SYS1.PL1LIB,DISP=SHR
//            DD   DSN=LOAD.PLMATH,DISP=SHR
//            DD   DSN=&PLIB..LOAD,UNIT=&PUNIT,VOL=SER=&PVOL,DISP=SHR
//            DD   DSN=&RDSN..LOAD,UNIT=&RUNIT,VOL=SER=&RVOL,DISP=SHR
//            DD   DSN=GFK.PL1LIB,DISP=SHR
//            DD   DSN=SYS1.FORTLIB,DISP=SHR
//            DD   DSN=GFK.FORTLIB,DISP=SHR
//SYSLIN      DD   DSN=*.C.SYSLIN,DISP=(OLD,DELETE)
//SYSLMOD     DD   DSN=&PLIB..MODS,UNIT=&PUNIT,VOL=SER=&PVOL,DISP=SHR,
//            DCB=BLKSIZE=6447
//SYSLOUT     DD   SYSOUT=A,DCB=(BLKSIZE=&BLKF,LRECL=121,RECFM=FBM)
//SYSPRINT    DD   SYSOUT=A,DCB=(BLKSIZE=&BLKV,LRECL=125,RECFM=VBA)
//SYSUT1      DD   UNIT=&DISK,DISP=(NEW,DELETE),
//            DCB=BLKSIZE=3303,SPACE=(3303,(200,100),,CONTIG)
//WORKFIL     DD   DISP=(NEW,DELETE),UNIT=&DISK,
//            DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB),SPACE=(TRK,(50,50))
//REGENT      PEND
```

```
//QQPCL       PROC RUNIT=3330,RDSN=REGENT,RVOL=REGICE,DISK=SYSDA,
//            PLIB=REGENT,PVOL=REGICE,PUNIT=3330,
//            BLKV=2004,BLKF=968,MPARM=LIST
//P           EXEC PGM=QQMODGEN,PARM='ISA(100K)/&MPARM'
//PRINTER     DD   SYSOUT=A,DCB=(BLKSIZE=&BLKV,LRECL=125,RECFM=VBA)
//QQCLIB      DD   DSN=&PLIB..PLSTRAN.MODS,UNIT=&PUNIT,VOL=SER=&PVOL,
//            DCB=BLKSIZE=6447,DISP=SHR
//            DD   DSN=&RDSN..PLSTRAN.MODS,UNIT=&RUNIT,VOL=SER=&RVOL,
//            DCB=BLKSIZE=6447,DISP=SHR
//QQCSYSN     DD   UNIT=&DISK,SPACE=(1680,(500)),
//            DCB=(BLKSIZE=1680,LRECL=80,RECFM=FB),DISP=(NEW,DELETE)
//QQMAC       DD   DSN=&RDSN..PLRMAC,UNIT=&RUNIT,VOL=SER=&RVOL,DISP=SHR
//QQPSYSN     DD   UNIT=&DISK,SPACE=(1680,(500)),
//            DCB=(BLKSIZE=1680,LRECL=80,RECFM=FB),DISP=(NEW,DELETE)
//QQROUTLB    DD   DSN=&PLIB..LOAD,UNIT=&PUNIT,VOL=SER=&PVOL,DISP=SHR,
//            DCB=BLKSIZE=6447
//SUBLIB      DD   DSN=&RDSN..PLSTRAN.DATA,UNIT=&RUNIT,VOL=SER=&RVOL,
//            DISP=SHR
//STEPLIB     DD   DSN=&PLIB..MODS,UNIT=&PUNIT,VOL=SER=&PVOL,DISP=SHR
//            DD   DSN=&RDSN..MODS,UNIT=&RUNIT,VOL=SER=&RVOL,DISP=SHR
//SYSCOUT     DD   UNIT=&DISK,SPACE=(80,(5000)),
//            DCB=(BLKSIZE=80,LRECL=80,RECFM=F),DISP=(MOD,PASS)
//SYSLIB      DD   DSN=&PLIB..LOAD,UNIT=&PUNIT,VOL=SER=&PVOL,DISP=SHR
//            DD   DSN=&RDSN..LOAD,UNIT=&RUNIT,VOL=SER=&RVOL,DISP=SHR
//            DD   DSN=SYS1.PL1LIB,DISP=SHR
//            DD   DSN=LOAD.PLMATH,DISP=SHR
//            DD   DSN=GFK.PL1LIB,DISP=SHR
//            DD   DSN=SYS1.FORTLIB,DISP=SHR
//            DD   DSN=GFK.FORTLIB,DISP=SHR
//SYSLIN      DD   DSN=*.SYSCOUT,DISP=(OLD,DELETE),VOL=REF=*.SYSCOUT
//SYSLMOD     DD   DSN=&PLIB..MODS,UNIT=&PUNIT,VOL=SER=&PVOL,DISP=SHR,
//            DCB=BLKSIZE=6447
//SYSLOUT     DD   SYSOUT=A,DCB=(BLKSIZE=&BLKF,LRECL=121,RECFM=FBA)
//SYSPRINT    DD   SYSOUT=A,DCB=(BLKSIZE=&BLKV,LRECL=125,RECFM=VBA)
//SYSUT1      DD   UNIT=&DISK,SPACE=(3303,(3000)),DCB=BLKSIZE=3303
//SYSUT2      DD   UNIT=&DISK,SPACE=(TRK,(100,20)),
//            DCB=(BLKSIZE=3200,LRECL=80,RECFM=FB)
//QQPCL       PEND
```

Aufruf:

```
//EXEC REGENT,␣RUNIT=runit,RVOL=rvol,RDSN=rdsn,
// PLIB=plib,PUNIT=punit,PVOL=pvol,CPARM=cparm,
// GPARM=gparm
// EXEC QQPCL,RUNIT=runit,RVOL=rvol,RDSN=rdsn,
// PLIB=plib,PVOL=pvol,PUNIT=punit,MPARM=mparm
```

runit,rvol,rdsn: Unit=Name, Volume und erster Teil des DS-Namens für REGENT-Bibliotheken.

punit,pvol,plib: Unit-Name, Volume und erster Teil des DS-Namens für private Bibliotheken.

cparm:PL/1-Compiler Parameter

gparm:Loader und PL/1-Laufzeitparameter

mparm:Modulgenerator-Parameter, siehe S.3-13.

8.11 Die Dateien des REGENT-Systems

Es folgt eine Zusammenstellung der Dateien, die in den REGENT-Prozeduren definiert sind.

Programmbibliotheken

Datei	DSNAME	DDNAME	Bemerkung
Steplib	&RDSN..MODS und &PLIB..MODS	STEPLIB	Standardbibliothek für Module. Wird über LINK oder EXEC PGM= angesprochen
Modul-Library	&RDSN..MODS oder &PLIB..MODS oder andere	SYSLMOD	Wird im Modulgenerator beschrieben, sofern nicht ein anderer DDNAME angegeben wurde. Bei Benutzen des Linkage Editor muß PARM='DCBS' angegeben werden.
State-ment-driver library	&RDSN..PLSTRAN.MODS und/oder &PLIB..PLSTRAN.MODS	QQCLIB PLSLIB	Wird in PLSTRAN gelesen. Wird in PLS von Linkage-Editor beschrieben

Routinen-Bibliotheken

Datei	DSNAME	DDNAME	Bemerkung
Syslib	&PLIB..LOAD &RDSN..LOAD SYS1.PL1LIB GFK.PL1LIB SYS1.FORTLIB GFK.FORTLIB	SYSLIB	Wird vom Loader oder Linkage-Editor gelesen
Routine-library	&RDSN..LOAD oder &PLIB..LOAD	QQROUTLB	Wird vom Modulgenerator mit subsystem-spezifischen PLR-Routinen und evtl. auch POL-Routinen beschrieben, sofern nicht ein anderer DDNAME angegeben ist. (s.Modul-Library)

Printdateien

Datei	REFCM LRECL	DDNAME	BEMERKUNG
Sysprint	VBA 125	SYSPRINT	Standardausgabedatei
Comprint	VBA 125	SYSPRINT	Compiler-Ausgabe in der REGENT-Prozedur
Printer	VBA 125	PRINTER	Modulgenerator-Ausgabe
Syslout	FBM 121	SYSLOUT	Loaderausgabe
Linksys-print	FBM 121	SYSLOUT	Linkage-Editor-Ausgabe
Utiprint	FBM 121	SYSLOUT	Utility-Ausgabe in Interpretierer
Regprint	VBA 125	REGPRINT	Protokollausgabe des REGENT-Interpretierers
Plidump	VBA 125	PLIDUMP	

Kartenformat-Hilfsdateien

Datei	DDNAME	Bemerkung
copy-statement	QQCØSI1 QQCØSI2	Wird in PLS als Eingabefile für IEBCOPY benutzt
partitioned-card-image-file	CØMFILE	partitioned dataset! Wird von PLS zur Übersetzungszeit beschrieben und zur Ausführungszeit gelesen.
plifile	PLIFILE QQCSYSIN	Wird vom REGENT-Übersetzer und vom PLR-Precompiler mit Eingabe für den PL/1-Compiler beschrieben
workfile	WORKFIL QQPSYSIN	Wird in PLS als Hilfsdatei benutzt. Wird vom Modulgenerator mit Eingabe für den PLR-Precompiler beschrieben
syscout	PLIØBJ SYSCOUT SYSCOUT	Achtung: DISP=MOD Compiler-output (Objektcode) wird in PLS beschrieben. Wird im Modulgenerator beschrieben. Wird im Interpretierer vom Compiler beschrieben.
syslin	SYSLIN	Wird vom Linkage-Editor gelesen im Modulgenerator und Interpretierer
Interpretierer-Eingabe	REGIN	Wird vom Interpretierer gelesen.

Permanente Kartenformat-Dateien

Datei	DSNAME	DDNAME	Bemerkung
Regent-Macro-Library	&RDSN..PLRMAC	QQMAC	Wird vom Modulgenerator und REGENT-Übersetzer gelesen. (Partitioned Data Set)
Subsystem-Übersetzer-Tabellen	SUBLIB &RDSN.. PLSTRAN.DATA		Wird vom Modulgenerator und REGENT-Übersetzer gelesen. Wird von PLS beschrieben. (Partitioned Data Set)

REGENT-Laufzeitdateien

Datei	DDNAME	Bemerkung
Überlaufbereich für Dynamische Datenstrukturen	QQDDSF	Wird von der Verwaltung Dynamischer Datenstrukturen als Erweiterung des Arbeitsspeichers benutzt
REGENT-Datenbanken	beliebig	DD-Karten müssen vom Anwender mit DCB=(DSORG=DA,KEYLEN=6,BLKSIZE=beliebig) beigefügt werden

Betriebssystemhilfsdateien

DDNAME	Bemerkungen
SYSUT1	Vom Compiler, Linkage-Editor und Utilities benutzt
SYSUT2	
SYSUT3	
SYSUT4	
SYSUT5	

8.12 Namensschnittstellen zum Betriebssystem

In REGENT gibt es einige Namensschnittstellen zum Betriebssystem: z.B. die Namen von Compilern, des Linkage Editors u.a. Diese Namen sind entweder 8 Zeichen lang oder z.B. beim PARM-Parameter maximal 100 Zeichen. Bei einer Umstellung auf eine andere Anlage müssen oft hier Änderungen vorgenommen werden. Ähnliches kann auch bei Änderungen des Betriebssystems selbst geschehen. Diesen Änderungen muß sich REGENT anpassen, ohne daß es ganz neu generiert wird. Außerdem soll die Lösung dieser Aufgabe so sein, daß sie erweiterbar ist. Es soll vermieden werden, daß alle Systeminterfaces dieser Art auf einen Schlag umgestellt werden müssen.

Folgende Lösung wurde realisiert: Es wird eine über die "Quick Link" Technik aufrufbare Routine bereitgestellt, welche als Eingabe eine Ziffer erhält und als Ausgabe die gewünschte Zeichenkette liefert. Die Bedeutung der Ziffer kann einer Tabelle entnommen werden, die auf der nächsten Seite beschrieben ist.

Die Interfacesstelle befindet sich in dem OS-Modul QQSYSØØ und kann folgendermaßen aufgerufen werden:

```
DCL IFACE ENTRY(CHAR(8), BIN FIXED(15), CHAR(x) VARYING)
                        INIT(QQLINK);
DCL QQLINK ENTRY EXTERNAL;
DCL STRING CHAR(100) VARYING;
CALL IFACE('QQSYSØØ',Code,STRING);
```

Die Zuordnung zwischen Eingabecode (zweites Argument) und gelieferter Zeichenkette ergibt sich aus der Tabelle, die auf der folgenden Seite aufgeführt ist.

Nr.	Bedeutung	Standard-Wert
1	OPTIMIZING-COMPILER PL/1	IELOAA
2	CHECKOUT-COMPILER PL/1	IEN512NS
3	ASSEMBLER H COMPILER	IEV90
4	ASSEMBLER F COMPILER	IEUASM
5	FORTRAN H COMPILER	IFEAAB
6	FORTRAN G COMPILER	IEYXORT
7	LINKAGE EDITOR 88K	IEWL
8	LINKAGE EDITOR 128K	IEWLF128
9	LOADER	LOADER
10	IEHPROGM	IEHPROGM
11	IEHLIST	IEHLIST
12	IEBGENER	IEBGENER
13	IEBUPDTE	IEBUPDTE
14	IEBCOPY	IEBCOPY
15	VOLUME-NAME	REGICE
16	DATA-SET-NAME	REGENT
17	MIN SPACE FÜR COMPRESS(TRKS)	10
18	MIN SPACE FÜR COMPRESS(%)	10
101	PARM.C für Interpretierer	'A,AG,XREF'
102	PARM.C für Modulgenerator	''=Nullstring
103	PARM.C für PLS	INCLUDE,NA,NAG,NOP,NSTG,NX, C,SYN
104	PARM.L für Interpretierer	''=Nullstring
105	PARM.L für Modulgenerator (Modulerzeugung)	LIST,MAP,REUS,RENT,DCBS, SIZE=(108K, 26K)
106	PARM.L für PLS	SIZE=(106K, 24K),DCBS, REUS, RENT
107	PARM.A für QQMODASS	''=Nullstring
108	PARM.L für Modulgenerator (Routineerzeugung)	LIST,MAP,REUS,RENT,DCBS, SIZE=(108K, 26K), NCAL

Liegt der Eingabecode außerhalb des Tabellenbereiches, so wird der String 'INVALID_CHARACTER_STRING' zurückgeliefert.

8.13 Implementierung

Unter Implementierung wird die Generierung des REGENT-Systems von Load-Modulen oder von Source-Statements verstanden.
Gleich welcher Art der Generierung durchgeführt werden muß, die Schritte zur Erstellung sind folgende:

1) Allokierung der Bibliotheken

2) Bootstrap bei ev. Betriebssystemanpassungen

3) Erzeugung des REGENT-Systems

8.13.1 Allokierung

Vor der Implementierung müssen auf einer 3330 Platte bzw. 2314 Platte 6 Partitioned Datasets allokiert werden. Folgende Tabelle gibt Auskunft über den Namen, die Größe und die DCB-Parameter dieser Datasets.

N a m e	Space in TRK[x]	BLKSIZE[x]	LRECL[xx]	RECFM[xx]
REGENT.LOAD	350	6447	-	U
REGENT.MODS	650	6447	-	U
REGENT.PLSTRAN.MODS	200	6447	-	U
REGENT.PLSTRAN.DATA	50	3120	80	FB
REGENT.MACLIB	5	3120	80	FB
REGENT.PLRMAC	60	3120	80	FB
x)empfohlen xx) nötig				

Die Größen der SPACE-Angaben beziehen sich auf eine 2314 Platte. Die Blockungsgröße von 6447 Byte wurde deswegen gewählt, damit es keine Blockungsschwierigkeiten bei einer Umstellung von 3330 Platten auf 2314 Platten oder umgekehrt gibt. Außerdem wird bei dieser BLKSIZE eine günstige Ausnützung beider Plattentypen gewährleistet (2314 Platte=88%, 3330 Platte=98%).

Die Namen der Bibliotheken sind nicht zwingend. Es müßten aber bei Änderung der Namen alle JCL-REGENT-Proceduren (QQINTER, REGENT und QQPCL) geändert werden. Bei der SPACE-Angabe der einzelnen Bibliotheken wurde berücksichtigt, daß das REGENT-System aus dem Systemkern und den Subsystemen DABAL, EDIT, QQTEST und GIPSY besteht.

8.13.2 Bootstrap und Anpassung an das jeweilige Betriebssystem

Der Bootstrap-Teil des REGENT-Systems enthält sowohl Routinen die zur Generierung des Systems von Source notwendig sind als auch die Betriebssystemschnittstelle. Diese ist im Kapitel 8.12 beschrieben. Sind einzelne Namen dieser Tabelle an das Betriebssystem anzupassen, so muß dies vor der Generierung geschehen. Zu diesem Zweck ist der Bootstrap-Teil mit der beiliegenden JCL zu starten. Bei der Generierung des REGENT-Systems kann dieser Schritt dann entfallen, wenn keine Anpassung notwendig ist. Auf jeden Fall muß dieser Schritt bei der Generierung, die von Source ausgeht, durchgeführt werden.

8.13.3 Generierung des REGENT-Systems von Load-Modulen

Bei der Auslieferung des REGENT-Systems in Load-Modul-Version kann man von den Voraussetzungen ausgehen, daß sich das REGENT-System als unloaded Version auf einem Magnetband befindet. Die 6 REGENT-Bibliotheken sind also mit dem Utilityprogramm IEHMOVE auf 6 getrennte Files eines Magnetbandes vor der Auslieferung kopiert worden. Die Generierung kann dann in folgenden Schritten ablaufen:

- Allokierung der 6 REGENT-Bibliotheken (siehe Kap.8.13.1)
- evtl.Anpassung und Laden der Systeminterfacestelle (siehe Kap.8.12)
- Kopierlauf mit dem Utility IEHMOVE vom Magnetband auf die Platte

Nun steht das REGENT-System zur Benutzung zur Verfügung. Sind außer dem Systemkern noch die Plotfähigkeiten mitgeliefert worden, so ist eine Anpassung an die vorhandene Plotsoftware getrennt durchzuführen. Siehe hierzu Kap.8.13.5

8.13.4 Generierung des Systemkerns von Source

Die REGENT-Systemkern-Source besteht aus ca. 50000 Karten. Man kann sie in

- Assembler Makros und Strukturen
- PL/1-Makros und Strukturen
- Assembler Programm
- Linkage Editor Eingabe

unterteilen. Würde man nun versuchen, die einzelnen Teilbereiche auf die herkömmliche Art und Weise zu laden, so wäre dies nur mit vielen Jobs möglich, sehr zeitaufwendig und unübersichtlich. Außerdem ist bei der Generierung ein genauer zeitlicher Ablauf vorgeschrieben, denn das Generieren eines Teilbereiches setzt das Vorhandensein eines anderen voraus. Aus diesem Grund wird zur Generierung ein Interpretierer bereitgestellt, der anhand von Steueranweisungen dynamisch einzelne Prozessoren (Compiler, Assembler, Linkage Editor,Utilities etc.) aufrufen kann. Er verarbeitet die angelieferte REGENT-Source sequentiell in Verbindung mit diesen Steueranweisungen. Aufgerufen wird der Interpretierer mit der JCL-Procedure QQINTER. Seine Arbeitsweise ist in Kapitel 8.13.6. beschrieben.

Bei der Auslieferung des REGENT-Systems in Source-Form kann man von folgenden Voraussetzungen ausgehen:

- die Source befindet sich in REGENT-Teilbereiche eingeteilt auf verschiedene Files eines Magnetbandes;
- in der Source befinden sich die Steueranweisungen für den Interpretierer.

Sind diese Voraussetzungen gegeben, so ist folgendermaßen zu verfahren:

- Zuerst muß der Bootstrapteil verfügbar gemacht, die Systeminterfacestelle gegebenenfalls angepaßt, die beigefügte JCL geändert und auf die REGENT-Bibliothek geladen werden.
- Danach können Schritt für Schritt die einzelnen Teilbereiche mit Hilfe des Interpretierers implementiert werden.

Bei jedem Teilbereich sind alle Programme, Strukturen, Makros, Module, Linkage-Editor-Eingabe und evtl. Steueranweisungen für die Modulgeneratoren so angeordnet, daß sie in zeitlich richtiger Reihenfolge in die Bibliotheken geladen werden. Eine Vertauschung der Implementierungsschritte ist aus diesem Grund nicht möglich. Ebenso eine Vertauschung der Implementierung der einzelnen Teilbereiche.

Ein ausgeliefertes REGENT-Source-Magnetband wird in der Regel wie folgt aufgebaut sein:

File 1: Bootstrap-Teil + JCL-Proceduren (Quick-Link-Programme, Systeminterfacestelle, Interpretierer, Proceduren QQINTER, QQPCL und REGENT).

File 2: REGENT-Modulverwaltung, Nachrichtenverwaltung, Modulgeneratoren (QQMODGEN, QQMODASS und QQMENDC) und Serviceroutinen.

File 3: REGENT-Übersetzer

File 4: REGENT-PLR-Precompiler

File 5: Dynamische Datenverwaltung

File 6: Dateiverwaltung

File 7: Die Plot-Software

File 8: Die Subsysteme DABAL und EDIT

8.13.5 Anpassung der Plotsoftware

Die Plot-Schnittstelle für die Ausgabe von Zeichnungen ist in Kapitel 12 beschrieben. Bei einer Neuinplementierung müssen die geräteabhängigen Grundroutinen und bestimmte Kontrollblöcke an die Gegebenheiten der jeweiligen Plot-Peripherie angepaßt werden. Die Kontrollblöcke und die Programme für die Zeichen - schnittstelle befinden sich in Source-Form auf dem Dataset IMPL06. DATA.

Zuerst werden mit IEBUPDTE PL/1-Datendeklarationen für spätere %INCLUDE'S geladen, danach folgt ein Assembler-Link-Step für das geräteunabhängige Plot-Interface. Danach wird für jedes Gerät eine PL/1-Prozedur compiliert und gelinkt, die für die Initialisierung der Plot-Datenstruktur und das Dazubinden der Plot-Basisroutinen in den Main-Modul sorgt. Die Initialisierungsroutine (z.B.QQCINI für Calcomp) enthält Deklarationen für die Basisroutinen als EXTERNAL ENTRY. Durch den Parameter PLOT=CALCOMP wird vom REGENT-Übersetzer generiert: DCL QQCINI EXTERNAL ENTRY;CALL QQCINI(QQ); Durch die Deklaration in QQCINI werden dann die richtigen Plot-Basisroutinen mit zum Main-Modul dazugeladen, da das Member QQCINI in REGENT.LOAD schon alle Referenzen richtig gelöst hat.

Die ENTRIES der Basis-Routinen werden in der Plotstruktur QQPLOTSTRU, die am PointerQQPLOTOUTPUT der REGENT-Struktur hängt, eingetragen. Ein Aufruf auf eine der Basisroutinen in einem Modul wird auf die Interface-Assembler-Routine gelenkt, die als ALIAS die Namen der Basisroutinen besitzt. Die Interface-Routine beschafft sich über den Hilfspointer QQ1, der STATIC EXTERNAL ist, die Adresse der Plotstruktur QQPLOTSTRU. Aus dieser Struktur holt sie die Adressen der "echten" Plot-Routinen im Main-Modul und verzweigt dorthin.

Mithin muß in IMPLO6.DATA folgendes geändert oder ergänzt werden:

1) Die Namen der "echten"Plot-Routinen in der Interface-Routine PLOTCC und in den Initialisierungsroutinen sind anzupassen.

2) Basisroutinen, die nicht verfügbar sind (z.B.PEN,MLTPLE) sind wegzulassen, die entsprechenden Plätze in der Plotstruktur werden leergelassen.

3) Basisroutinen, die dazukommen (z.B.CALL COLOR), müssen anstelle einer anderen Routine eingesetzt werden. Notfalls ist QQPLOSTRU zu erweitern(dies würde ein Neuübersetzen aller Module erforderlich machen).

4) Für jedes Gerät ist eine Initialisierungsroutine QQnINI zu schreiben entsprechend dem Schema der vorhandenen Routinen. "n" ist frei wählbar, darf aber nicht mit den vorhandenen Namen kollidieren (QQMINI, QQDINI, QQSINI, QQCINI, QQYINI, QQXINI, QQBINI, QQZINI sind schon vergeben). Für den Fall NOPLOT ist eine Dummy-Routine bereitgestellt, die eine Fehlermeldung liefert, wenn eine der Plotroutinen aufgerufen wird.

5) Jede Initialisierungsroutine ist so zu binden, daß die Referenzen auf die jeweilige Plot-Software gelöst werden (z.B. durch INCLUDE PLOTLIB1(PLOT) usw.)

8.13.6 Arbeitsweise des Interpretierers

Der Interpretierer ist ein in PL/1 geschriebenes Programm und arbeitet anhand von formatfreien Steueranweisungen sequentiell die Eingabe ab. Er erwartet diese Steueranweisung, die zwischen der sonstigen Eingabe (z.B.Programmsource) positioniert ist, über den File mit dem DD-Namen REGIN. Die Steueranweisungen beginnen immer auf Spalte 2, enden mit einem Strichpunkt und haben allgemein folgenden Aufbau:

Befehl, Programmname, Parm-Parameter, DD-Name, End of File Bedingung.

Folgende Befehle sind gültig und bewirken beschriebene Aktivitäten (unterstrichene Namensteile sind zugelassene Abkürzungen):

- EXECUTE — Es wird ein Programm ausgeführt oder bei fehlendem Programmnamen der OS-LOADER aufgerufen.

- COMPILE — Es wird ein PL/1-Compiler aufgerufen. Dies ist immer der PL/1-Optimizing Compiler, wenn per PGM-Option kein anderer angegeben wurde.

- ASSEMBLER — Es wird ein Assembler aufgerufen. Dies ist immer der H-Assembler, wenn per PGM-Option kein anderer angegeben wurde.

- LINKAGE-EDITOR — Es wird ein Linkage-Editor aufgerufen. Dies ist immer der IEWLF880=Linkage-Editor 88 K-Version, wenn per PGM-Option kein anderer angegeben wurde.

- UTILITY — Es wird ein Utility-Programm aufgerufen. Der Name des Utility-Programmes muß in der PGM-Option angegeben werden.

- COPY — Es wird ein internes Programm aktiviert, das einen sequentiellen Dataset auf einen anderen kopiert.

Bis auf das bei dem COPY-Statement verwendete Programm werden alle anderen Prozessoren dynamisch aufgerufen. Nach diesen Steueranweisungen schließt sich der Input für die einzelnen Prozessoren an und wird durch die End of File-Bedingung abgeschlossen. Dieser Delimiter muß bei jeder Steueranweisung angegeben werden. Es folgt die genaue Beschreibung der einzelnen Befehle.(Alle Angaben müssen einem gültigen PL/1-GET-DATA-Format entsprechen).

- EXEC - Befehl:

  Eingabemöglichkeit: PGM, PARM, EOF, SYSIN, SYSLIN, SYSLIB, SYSLOUT.

  Standardannahmen : PGM - OS - Loader
  SYSIN = 'SYSIN'
  SYSLIN = 'F168080'
  SYSLIB = 'SYSLIB'
  SYSLOUT = 'SYSLOUT'

- COMP - Befehl:

  Eingabemöglichkeit: PGM,PARM, EOF, SYSLIN, SYSLIB, SYSIN, SYSPRINT, SYSPUNCH, SYSUT1, SYSCIN

  Standardannahmen : PGM = PL/1 Optimizing Compiler

- ASSE - Befehl:

  Eingabemöglichkeit: PGM, PARM, EOF, SYSLIN, SYSIN, SYSPRINT, SYSPUNCH, SYSUT1, SYSTERM, SYSGO.

Standardannahmen: PGM = Assembler H
SYSLIN = 'F168080M'
SYSLIB = 'QQMAC'
SYSIN = ''
SYSPRINT = 'SYSPRINT'
SYSPUNCH = 'SYSPUNCH'
SYSUT1 = 'SYSUT1'
SYSUT2 = 'SYSUT2'
SYSUT3 = 'SYSUT3'
SYSGO = 'F168080M'
SYSTERM = 'SYSTERM'

- UTIL - Befehl:

Eingabemöglichkeit: PGM, PARM, EOF, SYSIN, SYSPRINT, SYSUT1, SYSUT2, SYSUT3, SYSUT4.

Standardannahmen: SYSIN 3 ''
SYSPRINT = 'SYSLOUT'
SYSUT1 = 'SYSUT1'
SYSUT2 = 'SYSUT2'
SYSUT3 = 'SYSUT3'
SYSUT4 = 'SYSUT4'

- COPY - Befehl:

Eingabemöglichkeit: IN, OUT, EOF.

Standardannahmen: IN = 'REGIN'
OUT = ''

Bemerkungen: Der Copy-Befehl kopiert einen sequentiellen Data-Set von dem File mit dem DD-Namen IN auf einen File mit dem DD-Namen OUT. Fehlt die OUT-Option, oder ist der Input-File gleich dem Output-File erfolgt kein Kopiervorgang.

- LINK - Befehl:

  Eingabemöglichkeit: PGM, PARM, EOF, SYSLIN, SYSLMOD, SYSLIB.

  Standardannahmen: PGM = 'IEWLF880'

Zu den einzelnen Befehlen folgt je ein Beispiel mit Erläuterung.

EXEC PGM = 'QQMODGEN', PARM = 'ISA(12K)/LIST', EOF = '**EOF';

Bedeutung:

Es wird dynamisch der Modulgenerator aufgerufen und an ihm die angegebenen PARM-Parameter übergeben. Die Eingabe für den Modulgenerator beginnt unmittelbar nach dieser Steueranweisung und endet bei dem nächsten **EOF-Statement.

COMP PARM = 'A,AG,XREEF,OPT(2)',EOF='**EOF';

Bedeutung:

Es wird dynamisch der PL/1-Optimizing-Compiler aufgerufen und an ihn die angegebenen PARM-Parameter übergeben. Die Programmsource, die der Compiler übersetzen soll, beginnt unmittelbar nach dieser Steueranweisung und endet bei dem nächsten **EOF-Statement.

ASSEMBLER PARM = 'LOAD, LIST, NODECK', SYSIN = 'QQPSYN';

Bedeutung:

Es wird dynamisch der Assembler, Level H, aufgerufen und an ihn die angegebenen PARM-Parameter übergeben. Das Assemblerprogramm, welches assembliert werden soll, befindet sich auf einem File, der mit dem DD-Namen QQPSYSIN angesprochen werden kann. Da keine End of File Bedingung in der EOF-Option angegeben wurde, ist der ganze Fileinhalt Eingabe für den Assembler.

```
UTILITY PGM = 'IEBUPDTE', PARM = 'NEW', SYSUT2 = 'QQMAC',
                                          EOF = '**EOF';
```

Bedeutung:

Es wird dynamisch das Utility-Programm IEBUPDTE aufgerufen. Als PARM-Parameter wird ihm NEW übergeben.Die Eingabe findet das Utility-Programm unmittelbar nach dieser Steueranweisung und das Ende durch die **EOF-Anweisung markiert. Die Ausgabe von IEBUPDTE gelangt auf den File, der mit dem DD-Namen QQMAC angesprochen wurde.

```
COPY IN = 'INFILE', OUT = 'OUTFILE';
```

Bedeutung:

Es wird mit Hilfe einer PL/1-Procedure, innerhalb des Interpretierers, der Inhalt des Files mit dem DD-Namen INFILE auf den File, mit dem DD-Namen OUTFILE kopiert.

Sowohl die Namen der einzelnen OS-Prozessoren (Compiler, Assembler, Utilities u.a.) als auch deren Standardparameter werden in QQINTER aus der in QQSYSSØØ angelegten OS-Interfacestelle entnommen. Lediglich die DD-Namen-Listen dieser Programme sind intern angelegt.

KAPITEL 9

9.1 DABAL - Handbuch

Einführung

Die REGENT-Dateiverwaltung dient der dynamischen Verwaltung von Speicherplatz innerhalb einer Betriebssystemdatei. Der zur Verfügung stehende Platz wird durch die Dateiverwaltung in Abschnitte, sog. Datapools gegliedert. Datapools dienen der Aufnahme von Benutzerdaten. Ein Catalogpool umfaßt eine Menge von Datapools und/oder Catalogpools, so daß eine Datapoolmenge baumartig strukturiert werden kann. Die Wurzel eines solchen Baumes ist der Hauptkatalog, die Bank. Die Grundfähigkeiten der Dateiverwaltung wie Initialisieren und Strukturieren einer Bank und die Ein-Ausgabe von Daten in Datapools stehen dem Subsystementwickler durch PLR-Statements zur Verfügung. Diese und komplexere bzw. speziellere Fähigkeiten werden dem Benutzer auf der Subsystemebene durch DABAL (Database-Language) zugänglich gemacht.

DABAL umfaßt zwei Bereiche: die Verwendung (Strukturaufbau bzw. -änderung) und die Verwaltung von Datenbanken. Letzterem dienen die DABAL-Befehle: COMPRESS, SAVE, RESTORE, COPY, CHANGE PASSWORD, CHANGE LENGTH, CHANGE NAME, STATUS.

Einmal initialisiert, können auf der Subsystemebene die REGENT-Dateien als Datenspeicher benutzt werden, um beispielsweise Eingabedaten für andere Subsysteme in REGENT-Dateien bereitzustellen bzw. Subsystemergebnisse aus REGENT-Dateien zu entnehmen. DABAL umfaßt dazu auch die Ein-Ausgabebefehle zur dynamischen Struktur- und Datenmanipulation, die auch in PLR zur Verfügung stehen: INITIATE, OPEN, CLOSE, STORE, RETRIEVE, SET.

Zur Gewährleistung eines Zugriffsschutzes gelten folgende Operationstypen und zulässige Vater-Sohn-Zugriffsmodi:

Operationstypen

Operation \ Typ	INPUT	OUTPUT	UPDATE
STORE		×	
RETRIEVE	×		
OPEN DELETE	×		
OPEN DELETE			×
INITIATE POOL		×	

Zulässige Vater-Sohn-Modi

Sohn \ Vater	INPUT	OUTPUT	UPDATE
INPUT	×	×	×
OUTPUT		×	×
UPDATE			×

REGENT-Dateien können aufgrund ihrer Struktur (Dateienbaum) und ihrer Zugriffseigenschaften (direkter Zugriff) sinnvoll für Spezialanwendungen eingesetzt werden, beispielsweise zur Verwaltung von Quellprogrammen. Dazu dienen die DABAL-Befehle CHANGE CARD, EXPAND, PACK, FIND, PRINT, RENUM. Sie interpretieren den referierten Datapool als Textdatei mit einer Kartenlänge von 80 Zeichen, wobei die Zeichen 73 ... 80 der Numerierung dienen. Die Karten der Datei werden durch ihre Folgenummer benannt.

Der Fehlerbehandlung dient die Variable DBERROR (BIN FIXED(15)-Variable), in der ein Fehlercode angeliefert wird, der dem negativen Wert der DABAL-Fehlermeldungsnummer entspricht. Bei Auftreten eines Fehlers wird die FINISH-Condition gesetzt. Ein DABAL-Programm enthält zum Standard-Fehlerabbruch den ON-Block

```
ON FINISH BEGIN;
   IF DBERROR ¬=0 THEN STOP;
END;
```

Wird bei den Operationen STORE oder RETRIEVE das Poolende erreicht, so wird die ENDFILE-Condition für den Datenbankfile gesetzt.

Beispiel

```
CHANGE PASS(R='NEWC',UP='NEWUP') ON POOL (POOL 1);
```

Ändert die Paßwörter für das Lesen des Pools POOL1.

```
CHANGE LENGTH(2000) POOL(POOL2);
DCL NEWNAME CHAR(8) INIT('NEWNAME');
CHANGE NAME(NEWNAME) POOL(POOL3);
```

CHANGE
attribute

Syntax

```
CHANGE option [option]3 ON typ (ref);
              1

option::=PASSWORD(list)
         LENGTH (bin-fixed-31-expr.)
         NAME (char-expr.)

typ::=BANK|POOL

list::=siehe INITIATE
```

Erläuterung

Dieser Befehl ermöglicht die Änderung eines Schlosses, des externen Pool- oder Bank-Namens (TITLE-Option) und der Länge einer Bank, eines Catalogpools oder Datapools. Dazu muß der Pool (Bank) mit dem entsprechenden Schlüssel geöffnet worden sein: für CHANGE PASSWORD und CHANGE NAME mit dem CHANGE-PASSWORD-Schlüssel und für CHANGE LENGTH mit dem OUTPUT-Schlüssel.
Die Länge eines Pools kann nicht kleiner werden als der Füllfaktor, d.h. es können keine Informationen verloren gehen.

Beispiel

Gegeben sei ein Datapool, dessen 10.Karte folgende Zeichenkette enthalte:

1234567890ABCDE

```
DCL N BIN FIXED(31) INIT(10);
CHANGE CARD N IN POOL (TEXT_POOL) V REPL STR '45' 'XXXX';
```

ergibt: 123XXXX67890ABCDE

```
CHANGE CARD 10 POOL (TEXT_POOL) REPL COL 11 14 BY 'XXXXXXX';
```

ergibt: 1234567890XXXXXXXE

```
CHANGE CARD 10 POOL (TEXT_POOL) CONT STR 'AB' 'XXX';
```

ergibt: 1234567890XXX

```
CHANGE CARD 10 POOL (TEXT_POOL) CONT COL 3 WITH 'XX';
```

ergibt: 12XX

Syntax

```
CHANGE CARD sequ-nr1 [TO sequ-nr2]
       IN  POOL(poolref) option [VERIFY];

   option::=option1   option2   option3   option4
   option1::=REPLACE STRING oldstring  BY  newstring
   option2::=REPLACE COLUMN first last  BY  newstring
   option3::=CONTINUE STRING beginstring  WITH  newstring
   option4::=CONTINUE COLUMN column  WITH  newstring

   oldstring,newstring::=char-expr.
   first,last column::=bin-fixed-15-expr.
   sequ-nr1, sequ-nr2::=bin-fixed-31-expr.
```

Erläuterung

Mit CHANGE CARD werden in dem Datapool 'poolref' (Textdatei) die Karten sequ-nr1 bis sequ-nr2 geändert. Dabei gibt es vier Möglichkeiten:

- Ersetzen der Zeichenkette'oldstring'durch die Zeichenkette'newstring'

- Ersetzen der Zeichen zwischen den Spalten'first'und'last'durch die Zeichenkette'newstring'

- Fortsetzen einer Zeichenkette ab'beginstring'durch die Zeichenkette'newstring'

- Fortsetzen einer Zeichenkette ab Spalte'column'durch die Zeichenkette'newstring'

Die VERIFY-Option bewirkt das Ausdrucken der geänderten Zeilen auf SYSPRINT.

Beispiele

```
OPEN POOL(POOL1) TITLE('POOL1') IN POOL(POOLO) OUTPUT
        DELETE DATA OLD;

CLOSE POOL(POOLO);
        Schließt auch Pool 'POOL1' und löscht ihn dabei
        (DELETE beim OPEN)

CLOSE BANK(BANK1);

OPEN POOL(POOL2) TITLE(POOLTITLE) IN BANK(BA)
        RELEASE OUTPUT OLD DATA;

CLOSE POOL(POOL2);
        Schließt Pool 'POOL2' und gibt dabei ungenutzten
        Speicherplatz frei (RELEASE im OPEN)
```

Syntax

```
CLOSE  typ  (ref);

typ::=BANK|POOL
```

Erläuterung

Diese Operation schließt eine Datenbank und einen Pool und zwar, falls vorhanden, implizit auch sämtliche Pools des bezeichneten Teilbaumes. Nach einem CLOSE ist ein Zugriff auf Elemente des bearbeiteten Teilbaumes nicht zulässig, bevor sie nicht neu eröffnet worden sind. Entsprechend der beim OPEN Pool angegebenen Modi (KEEP, DELETE, RELEASE) werden die Elemente beim Schließen erhalten, gelöscht, oder es wird ungenutzter Platz freigegeben.

Beispiel

siehe DESCRIBE

```
COMPRESS BANK(BANK1) INOUT('DD1') NEWBLK(2000);
```

dazu erforderlich:

```
//G.DD1 DD UNIT=SYSDA,SPACE=(TRK,20),
//       DISP=(NEW,DELETE)
```

Komprimiert die Bank 'BANK1' mit Hilfe des Files 'DD1' und ändert die Blocksize von 1000 auf 2000.

Syntax

COMPRESS BANK(bankref) INOUT(dd2) [NEWBLK(nblk)] ;

nblk::=bin-fixed-31-expr.

Erläuterung

COMPRESS beseitigt die Sekundärspeicherfragmentierung, d.h. es stellt den gesamten freien Platz der Datenbank zusammenhängend zur Verfügung. Dazu wird eine Hilfs-(Direkt-Zugriffs-) Datei benötigt, deren DD-Name in der INOUT-Klausel angegeben wird. Darüber hinaus besteht die Möglichkeit, die Datenbank umzublocken (NEWBLK-Klausel), wodurch die Effektivität des Zugriffs beeinflußt wird.

Beispiele

```
OPEN POOL(POOLX) TITLE('A') IN BANK(BANK1)
         OLD OUTPUT KEEP CATALOG;

OPEN POOL(POOLY) TITLE('B') IN BANK(BANK1)
         OLD UPDATE KEEP CATALOG;

COPY STRUCTURE POOL(POOLY) TO POOL(POOLX);
OPEN BANK(BANK2);
COPY STRUCTURE BANK(BANK1) TO BANK(BANK2);

DCL FILE1 FILE;
DCL FILE2 FILE;

OPEN POOL(POOL1) TITLE(POOLNAME) IN BANK(Y)
         MOD OUTPUT KEEP DATA;
COPY DATA FILE(FILE1) TO POOL(POOL1);
COPY DATA POOL(POOL1) TO POOL(NEW)
         LRECL(800)  NUMBER(10,10);

COPY DATA POOL(NEW) TO FILE(FILE2);
```

Anmerkung: Pool NEW muß bereits initiatisiert und geöffent sein.

Syntax

```
COPY option;

   option::=STRUCTURE  art  (ref1) [TO] art  (ref2) |
            DATA typ [TO] typ [LRECL(lrecl)] [NUMBER
                                          [(begin,step)]]

   typ::=POOL(ref3) | FILE(file1)

   lrecl,begin,step::=bin-fixed-31-expr.

   title::=char-8-expr.

   art::=BANK |POOL

   Anmerkung: LRECL nur bei POOL TO POOL
```

Erläuterung

COPY STRUCTURE

Mit diesem Befehl wird ein Pool(Bank) einem Catalogpool(Bank)zugewiesen, wobei die Zuweisung sich auf die Struktur und die Daten bezieht. Die beiden Operanden dürfen der gleichen Bank angehören. Eine eventuell vorhandene Unterstruktur des Target-Elements wird zerstört. Bei dieser Operation darf ein Operand nicht Element des anderen Operanden sein.

COPY DATA

Diese Operation kopiert Benutzerdaten von einem Datapool in einen anderen Datapool, von einem Datapool in eine sequentielle Betriebssystemdatei oder aus einer sequentiellen Betriebssystemdatei in einen Datapool (POOL( ) TO POOL( ), POOL( ) TO FILE( ), FILE( ) TO POOL( )). Bei der POOL-TO-POOL-Option kann eine Record-Länge (LRECL-Klausel) angegeben werden, die die Effektivität beeinflußt. In den anderen Fällen wird die RECORD-Länge der Betriebssystem-Datei (DCB-Parm) verwendet. Soll die Datei als Kartendatei interpretiert werden, so kann eine Numerierung vorgenommen werden: NUMBER-Klausel, in der die erste Nummer und die Schrittweite angegeben wird (Default:10,10).

Beispiele

Die Beispiele enthalten die jeweils für weitere Operationen erforderlichen DD-Karten.

```
DESC BANK1 BANK;
     erfordert: //G.BANK1  DD ... DCB=BLKSIZE=1000

DESC BANK2 BANK FILE(FILE1);
DCL FILE1 FILE ENV(REGIONAL(1),F);
     erfordert: //G.FILE1 DD ...

DESC BANK3 BANK TITLE('DD3');
     erfordert: //G.DD3 DD ...

DESC BANK4 BANK FILE(FILE1) TITLE(T);
DCL T CHAR(8) INIT('DD4');
     erfordert: //G.DD4 DD ...

DESC BANK5 BANK VARIABLE INIT(BANK1);
DESC POOL1 POOL INIT(NULL( ));
DESC POOL_ARRAY(20) POOL;

ON ENDFILE (BANK1 -> QQFILE) GOTO B;
ON ENDFILE (FILE1) GOTO X;
```

Syntax

DESCRIBE name option [storage-attr]

option::=BANK [FILE(file)] [TITLE(title)] |
BANK VARIABLE [INIT(ref)] |
[(dim)] POOL [INIT(ref)]

storage-attr.::=STATIC | AUTOMATIC | parameter

dim::=integer-expr.

title::=char-8-expr.

Erläuterung

DABAL kennt zwei Datenbankobjekte: Die BANK und den POOL. Die BANK repräsentiert die Datenbank insgesamt, die POOLS repräsentieren Catalog- oder Datapools, je nach Option im INIT-Befehl. Durch die FILE-Klausel der BANK-Konstanten wird die Beziehung zwischen der BANK und dem File hergestellt, auf dem sich die Datenbank befindet. In der TITLE-Klausel wird die zugehörige DD-Karte benannt. Default-Wert für 'title' ist 'name', wenn die FILE-Klausel nicht verwendet wurde, sonst 'file'. Für den File wird ein Default-Wert erzeugt, der mit name —>QQFILE referiert werden kann.

Ein Datenbank-File muß das ENVIRONMENT (REGIONAL(1),F) besitzen. Einer Bankvariablen muß vor der Verwendung in einem DABAL-Statement der Wert einer Konstanten zugewiesen werden, oder sie muß Parameter sein.

Beispiele

```
EXPAND POOL(POOL1) POOL(EXP_POOL);

EXPAND POOL(POOL1) INDD('DDEXTEXT');

EXPAND PLI POOL(PLIPOOL) POOL(EXP2_POOL);
```

DD-Karten

Wird nur EXPAND ohne PLI verwendet, so genügt eine Workdatei, die über QQWORKDA angesprochen wird (Attribute siehe unten). Bei EXPAND PLI (dynamischer Aufruf des Checkout-Compilers) sind außerdem DD-Karten für SYSPLIC, SYSUT1, SYSUT2, SYSFORM und CHKPRINT erforderlich. Die DCB-Parameter sind dabei genau so zu übernehmen, wie im Beispiel codiert.

```
//G.QQWORKDA DD UNIT=SYSDA,DISP=(NEW,DELETE),SPACE=(TRK,50),
// DCB=(DSORG=DA,BLKSIZE=1680,LRECL=80,RECFM=F)
//G.CHKPRINT DD DUMMY
//G.SYSPLIC DD DSN=SYSX.LINKLIB,DISP=SHR
//G.SYSUT1 DD DSN=&&SYSUT1,UNIT=SYSDA,DCB=BLKSIZE=1024,
//   SPACE=(1024,(60,60),,CONTIG)
//G.SYSFORM DD DSN=&&SYSFORM,UNIT=SYSDA,DCB=BLKSIZE=80,
//   SPACE=(80,(500,500))
//G.SYSUT2 DD DSN=&&SYSUT2,UNIT=SYSDA,DCB=LRECL=84,
//   SPACE=(TRK,(10,10))
```

Syntax

```
EXPAND [PLI] POOL(ref) option;

option::=POOL(ref) | INDD(char-8-expr.)
```

Erläuterung

EXPAND fügt in eine mit PACK bearbeitete Textdatei wieder Blanks ein und schreibt das Ergebnis in den Datapool 'ref' bzw. auf die in der INDD-Klausel durch den DD-Namen bezeichnete OS-Datei. Wurde mit der Option PLI komprimiert, so muß EXPAND PLI verwendet werden. Die Expansion erfolgt dann mit der FORMAT-Option des PL/1-Checkout-Compilers. Dabei wird allerdings nicht die ursprüngliche Version der Textdatei restauriert, sondern lediglich eine formatierte Version.

Bei nicht PLI-Dateien wird die ursprüngliche Textversion wieder restauriert.

Beispiele

```
FIND POOL(TEXT_POOL) STRING('SUCH MICH')
          SET(FOUND_CARD);

FIND POOL(TEXT_POOL) STRING('2851') POS(73,80)
          DECK(CARD1,CARD100) SET(FOUND_CARD);

FIND POOL(TEXT_POOL) STRING(TEXT_STRING)
          SET(CARD.FOUND) VERIFY;
```

Druckt 1.Karte mit gesuchtem String aus.

Syntax

FIND $[opt1]_{3}^{3}$ $[opt2]_{o}^{3}$;

opt1::=POOL(ref) | STRING(string) | SET(sequnr)

opt2::=POS(expr1, expr2) | DECK(expr3, expr4) | VERIFY

expr1, ..., expr4::=integer-expr.

sequnr::=bin-fixed-31-var

string::=char-expr.

Erläuterung

Der FIND-Befehl weist der in der SET-Klausel benannten Variablen die Folgenummer der ersten Karte des Datapools 'ref' (Textdatei) zu, die die Zeichenkette 'string' enthält. Die Suche kann eingeschränkt werden auf die Spalten expr1 bis expr2 und die Karten mit den Folgenummern expr3 bis expr4. Wird die Option VERIFY benutzt, so wird die gefundene Zeile ausgedruckt (SYSPRINT).
Ist die Suche erfolglos, so wird in der SET-Klausel der Wert -1 zurückgeliefert.

Beispiele

Deklarationen siehe DESCRIBE.

```
INIT BANK(BANK1);
```

initialisiert eine ungeschützte Bank mit Defaultwerten.

```
INIT BANK(BANK2) SPACE(100) FREE(500)
     PASS(ALL='ALL',WP,UP='WP_UP');

INIT BANK(BANK3) TITLE('NAMENSSCHUTZ')
     PASS(ALL PRIV='AP');
```

Syntax

```
INITIATE BANK(bankreference) [TITLE(ext-bankname)] [attribute]*
                                                    [passwords];

   attributes::=FREELIST(bin-fixed-31-expression)|
                DELETELIST(bin-fixed-31-expression)|
                NODELETELIST|
                LISTPAGE(bin-fixed-31-expression)|
                SPACE(bin-fixed-31-expression)

   passwords::=PASSWORDS(list) | NOPASSWORDS | NOPASS

   list::=passtype [,passtype]* = char(8)-expression


   passtype::=ALL | ALL PRIV | ALL NOPRIV |
              R | READ | W | WRITE | U | UPDATE |
              C | CHANGE PASSWORD | RP | READ
              PRIV | WP | WRITE PRIV | UP |
              UPDATE PRIV | CP | CHANGE PASSWORD PRIV
```

Erläuterung

INITIATE initialisiert auf der der Bank zugeordneten Datei des Betriebssystems eine Datenbank, die mit Hilfe der TITLE-Klausel einen Namen erhalten kann. Dieser Name ist nicht erforderlich, da eine eindeutige Identifikation durch die Zuordnung der Betriebssystemdatei erfolgt, die nur eine Bank enthalten kann. Er dient lediglich als zusätzlicher Zugriffsschutz.
Mit den angeführten Klauseln werden die Längen von Verwaltungslisten und die Länge des Hauptkatalogs (SPACE-Klausel, Default:16) angegeben. FREELIST (Default:64) und DELETELIST (Default:16) sind Listen für die Verwaltung freien Speicherplatzes. LISTPAGE (Default:16) gibt die Länge von Segmenten dieser internen Verwaltungslisten an.

Mit der PASSWORD-Klausel können Schlösser zum Zugriffsschutz installiert werden. Für den Zugriffsschutz werden vier Operationen unterschieden: READ, WRITE, UPDATE, CHANGE PASSWORD. Soll ein Schloß nicht nur beim Öffnen der BANK wirken, sondern auch für alle BANK-Elemente, so müssen die Optionen ...PRIV verwendet werden (Generalschlüssel).
Die Spezifikationen der 'list' werden wie Zuweisungen interpretiert, d.h. Mehrfachzuweisungen und Ändern durch neuerliches Zuweisen sind möglich. Die Schlösser sind in drei Gruppen eingeteilt: ALL erfaßt alle Schlösser, ALL PRIV erfaßt alle Generalschlösser und ALL NOPRIV alle einfachen Schlösser. Wird keine PASSWORD-Klausel angegeben (oder bei NOPASS, NOPASSWORDS), so werden für alle Schlüsser Blanks angenommen. Wird die PASS-Klausel verwendet, so wird die Bank gegen alle nicht aufgeführten Operationstypen verriegelt.

Der gesamte Platz für die Datenbank wird durch den SPACE-Parameter auf der DD-Karte angegeben. Die Länge der Datenbankpuffer entspricht der auf der DD-Karte angegebenen BLKSIZE.
In der TITLE-Klausel kann ein Name für die Bank angegeben werden. Dieser Name muß dann auch beim Öffnen der Bank verwendet werden (TITLE-Klausel).

Beispiele

```
DESC POOL(5) POOL;
DESC BANK1 BANK;
INITIATE BANK(BANK1);
OPEN BANK(BANK1) OUTPUT;
INITIATE POOL(POOL(1)) TITLE('POOL1') SPACE(20)
         IN BANK(BANK1) PASS(R,W='RW') CATALOG;
OPEN POOL(POOL(1)) TITLE('POOL1') IN BANK(BANK1)
         PASS('RW') OLD KEEP OUTPUT CATALOG;
DCL PIC PICTURE '9';
DO PIC=2 TO 5;
  INIT POOL(POOL(PIC)) IN POOL(POOL(1)) TITLE('POOL' ||PIC)
         DATA SPACE(100);
END;
```

Syntax

```
INITIATE POOL(pool)TITLE(title)[SPACE(space)]
        where [type][PASSWORD(list)] ;

    where::=IN BANK(bankref) | IN POOL(poolref)

    type::=CATALOG | DATA Default: DATA

    list::=siehe INIT BANK

    title::=char-8-expr.
```

Erläuterung

Mit dieser Operation wird ein Catalogpool (CATALOG) oder Datapool (DATA) angelegt und in einen Katalog (IN-Klausel) eingetragen. Da in einen Katalog wieder ein Katalog eingefügt werden kann, ist der Aufbau einer Hierarchie möglich. Der neue Pool erhält den in der TITLE-Klausel angegebenen Namen (Katalogeintrag), der aufgrund des einstufigen Bezugs zu einem Vaterelement, nur in diesem Katalog (an einer Hierarchiegabel) eindeutig sein muß.
Mit der SPACE-Klausel wird die gewünschte Zahl der Katalogeinträge bzw. die Länge eines Datapools in Bytes angegeben. Ein Zugriffsschutz kann mit Hilfe der PASSWORD-Klausel wie bei INITIATE BANK definiert werden.

Die Operation INITIATE POOL ist eine OUTPUT-Operation,d.h. sie ist nur zulässig, wenn das in der IN-Klausel genannte Element mit OUTPUT oder UPDATE eröffnet ist.
Im Unterschied zu PLR sind in der IN-Klausel die Schlüssel BANK bzw. POOL erforderlich.

Beispiele

Siehe DESCRIBE und INITIATE BANK

```
OPEN BANK(BANK1) BUFNO(16) INPUT;
OPEN BANK(BANK2) PASS('ALL') OUTPUT;
OPEN BANK(BANK3) TITLE('NAMENSSCHUTZ') PASS('AP') UPDATE;
```

Syntax

```
OPEN BANK (bankreference) [TITLE(ext-bankname)]
          [PASSWORD(char(8)-expression)]
          [BUFNO(bin-fixed(15)expression)]
          [mode];
ext-bankname::=(44)''falls nicht vorhanden, sonst:
               char(44)-expression
mode::=INPUT | OUTPUT | UPDATE
```

Erläuterung

OPEN BANK öffnet eine Datenbank.
Die Beziehung zu der Betriebssystemdatei, auf der sich die Datenbank befindet, wird über die BANK-Variable ('bankreference') hergestellt. Ist die Datenbank mit der TITLE- bzw. der PASSWORD-Klausel initialisiert worden, so sind diese Klauseln auch im OPEN erforderlich. Der Mode kennzeichnet die für die Bank gewünschte Operationsart. Mit der BUFNO-Klausel wird die Anzahl der I/O-Puffer festgelegt (Default:2).

Beispiele

```
OPEN POOL(ROOT) TITLE('WURZEL') IN POOL(POOLX) OLD KEEP
        CATALOG UPDATE PASS('OUT');

OPEN POOL(POOL2) TITLE('A5') IN POOL(ROOT)
        CAT PASS('READ_PA') OUTPUT;

OPEN POOL(POOL3) TITLE(TITLESTRING) IN POOL(POOL2);

OPEN POOL(POOL4) TITLE('XX') IN POOL(ROOT) DELETE;

CLOSE POOL(ROOT);
        schließt alle Pools, wobei Pool'POOL4' gelöscht wird.
```

Syntax

```
OPEN POOL(pool)TITLE(title) where [opendisp][closedisp] [mode]
                [type][PASSWORD(pass)];
        where::=IN BANK (bankref) |
                 IN POOL (poolref)
        opendisp::=OLD | MOD
        closedisp::=KEEP | DELETE | RELEASE
        mode::=INPUT | OUTPUT | UPDATE
        type::=CATALOG | DATA
        pass::=char-8-expr.
        Default: OLD, INPUT, KEEP, DATA
```

Erläuterung

OPEN POOL bereitet einen Katalog - oder Datapool auf den Zugriff vor, so daß er über die Poolvariable'pool'referiert werden kann. Dazu wird der Pool mit dem Namen'title'in dem mit der IN-Klausel angegebenen Katalog (Bank oder Pool) gesucht. Ist ein Pool dieses Namens vorhanden, so wird überprüft, ob es sich um den geforderten Typ (DATA oder CATALOG) handelt, ob das angegebene Schlüsselwort zu dem gewünschten Operationsmodus (INPUT, OUTPUT, UPDATE) berechtigt und ob dieser Modus dem des Vaters (IN-Klausel) nicht widerspricht. Bei positivem Abschluß dieser Prüfungen ist die durch'title' bezeichnete Datei über'pool'zu referieren. Die Optionen OLD und MOD beziehen sich auf Datapools und bewirken, daß der Ein-Ausgabezeiger für sequentiellen Zugriff (siehe STORE und RETRIEVE) auf den Poolanfang (OLD) bzw. auf den Füllstand zeigt (MOD). Sollen also in einer bestehenden Datei an die bereits vorhandenen Daten durch sequentiellen Zugriff neue Daten angefügt werden, so muß der Datenpool mit OPEN...MOD eröffnet werden.
Im Unterschied zu PLR sind in der IN-Klausel die Schlüssel BANK bzw. POOL erforderlich.

OPEN POOL
(cont.)

Mit dem OPEN-Befehl wird darüber hinaus angegeben, ob der jeweilige Pool bei einem nachfolgenden CLOSE vollständig erhalten bleiben soll (KEEP), gelöscht (DELETE) oder nur ungenutzer Platz freigegeben werden soll (RELEASE).

Ein OPEN DELETE erfordert, daß der Vater (IN-Klausel) mit UPDATE eröffnet ist, da die Bankstruktur geändert wird; somit hängt der erforderliche Modus des Vaters von dem gewünschten Modus des zu öffnenden Elements ab.

Beispiele

```
PACK PLI POOL(PLI_TEXT_POOL);

PACK POOL(TEXT_POOL) NUMBER(10,100);
```

Syntax

PACK [PLI] POOL(ref) [NUMBER(start,incr)] ;

start,incr::=bin-fixed-31-expr

Erläuterung

PACK PLI beseitigt in einer PL/1-Textdatei alle überflüssigen Blanks und numeriert die Karten neu, wenn die NUMBER-Klausel verwendet wird. Die Numerierung beginnt mit 'start', das Numerierungsinkrement ist 'incr'.

Wird die PLI-Option nicht verwendet, so werden Blank-Ketten durch ein nicht druckbares Zeichen und ein bzw. zwei Bytes ersetzt, die die Kettenlänge darstellen.

Beispiele

```
PRINT POOL(TEXT_DATEI);

PRINT POOL(POOL1) FILE(PRINT2);

DCL PRINT2 FILE PRINT;

PRINT POOL(T_POOL) ID('UEBERSCHRIFT') NUMBER(1,1)
      DECK(CARD1,CARDX);
```

Syntax

```
PRINT POOL(ref) [option]0^4 ;

      option::=FILE(printfile) | ID(ident) |
               NUMBER [(begin,step)] | DECK(anfang,ende)

      begin,step,anfang,ende::=bin-fixed-31-expr.
```

Erläuterung

Mit PRINT wird die Textdatei 'ref' gedruckt, wobei jeder Karte (Zeile) ihre Folgenummer in der Datei vorangestellt wird. Mit der FILE-Klausel wird der gewünschte Printfile angegeben (Default: SYSPRINT), 'ident' ist eine gewünschte Überschrift, NUMBER bewirkt eine Numerierung der Zeilen (beginnend bei 'begin' mit dem Inkrement 'step') und DECK erlaubt die Beschränkung der PRINT-Operation auf die Karten 'anfang' bis 'ende'.

Beispiele

```
RENUM POOL(TEXT_POOL) 1 1;

RENUM POOL(DATAPOOL) FIRST_NUM INCREMENT;

DCL (FIRST_NUM,INCREMENT) BIN FIXED(31);
```

Syntax

```
RENUM POOL(ref) first incr;
      first,incr::=bin-fixed-31-expr.
```

Erläuterung

Mit RENUM werden die Karten der Textdatei 'ref' numeriert. Die Numerierung beginnt mit first, incr ist das Numerierungsincrement.

Beispiele

```
RESTORE FROM('DD1') TO('DD2');
```

mit JCL:

```
//G.DD1 DD UNIT=TAPEX, VOL=SER=BANKSA,
//       DSN=BANK3,LABEL=3,DISP=(,PASS)
//G.DD2 DD DSN=IRE846.BANK3,UNIT=3330,
//       VOL=SER=REGICE,DISP=(NEW,KEEP),
//       SPACE=(TRK,50),DCB=BLKSIZE=1000
```

Syntax

RESTORE [BANK] FROM(dd1) TO(dd2);

dd1,dd2::=char(8)-expr.

dd1: Sequentielle Datei

dd2: DA-Datei

Erläuterung

RESTORE kopiert eine Bank von einer sequentiellen Datei (z.B.Band) in eine REGIONAL(1)-Datei, so daß wieder alle Datenbankzugriffe möglich sind. In der FROM-Klausel wird der DD-Name der sequentiellen Datei, in der TO-Klausel der DD-Name der Direkt-Zugriffs-Datei angegeben. Wird für die Datenbank mehr als der erforderliche Platz bereitgestellt, so wird auch der zusätzliche Platz verfügbar.

Beispiele

```
RETRIEVE POOL(DATAPOOL) TO(I);
RETRIEVE POOL(D) TO(WERT) FROM(POOLOFFSET)
                 NEXT(OFFSET_OF_NEXT_RECORD);
RETRIEVE POOL(D) TO(W) SET(OFFSET_OF_THIS_RECORD);
DCL STRING CHAR(LENGTH);
RETRIEVE POOL(D) TO(STRING);
DCL P PTR INIT(ADDR(STRING));
RETRIEVE POOL(D) TO(P) LENGTH(LENGTH-2);
```

Syntax

```
RETRIEVE POOL(ref) option [option2]0,2;

    option::=SET(adr) | FROM(offset) [NEXT(adr)]
    option2::=TO(variable) |
              TO(pointer) LENGTH(length)
    length,offset::=bin-fixed-31-expr
    adr::=bin-fixed-31-variable
```

Erläuterung

Mit diesem Befehl werden Benutzerdaten aus einem Datapool, d.h. einem Pool, der mit OPEN POOL...DATA eröffnet wurde, in eine in der TO-Klausel spezifizierte Variable eingelesen (oder in einen Speicherbereich, der durch 'pointer' und 'length' beschrieben wird).

Beim Zugriff ist ein Datapool für den Benutzer gekennzeichnet durch seine Länge in Bytes, seinen Füllstand und bei sequentiellem Zugriff durch einen Zeiger, der das nächste Byte bezeichnet. Dieser Zeiger zeigt nach einem OPEN OLD auf das erste Byte.

RETRIEVE bietet die Möglichkeit, sequentiell zu arbeiten oder direkt auf einzelne Bytes oder Bytegruppen zuzugreifen. Bei einem sequentiellen Zugriff (RETRIEVE...TO) wird der Ein-Ausgabe-Zeiger jeweils um die Recordlänge weitergesetzt.
Beim direkten Zugriff werden die Bytes des Pools durch ihren Offset vom Poolbeginn adressiert. Zur Kennzeichnung des direkten Zugriffs wird eine zusätzliche Befehlsklausel verwendet, in der die Adresse angegeben wird (RETRIEVE...FROM(adr)).
Der Ein-Ausgabe-Zeiger wird bei direktem Zugriff nicht verändert.
In der SET-Klausel wird eine Variable spezifiziert, in der der Offset des gerade bearbeiteten Records zurückgeliefert wird. Sie ist vor allem für sequentielles Arbeiten gedacht, um danach direkt zugreifen zu können. Mit der NEXT-Klausel wird der Offset des auf den bearbeiteten Record folgenden Bytes geliefert.

Beispiel

```
SAVE FROM('BANK1') TO('BAND1') FACTOR(5);
```

mit JCL:

```
//G.BANK1 DD DSN=IRE846.BANK1,UNIT=3330,
//         VOL=SER=REGICE,DISP=(OLD,KEEP)
//G.BAND1 DD DSN=SAVE2,UNIT=TAPEX,
//         LABEL=4,DISP=(,PASS)
```

Syntax

SAVE [BANK] FROM(dd1) TO(dd2)[FACTOR(bin-fixed-15-expr.)]

dd1,dd2::=char-8-expr.

dd1: DA-Datei
dd2: Sequentielle Datei

Erläuterung

SAVE kopiert eine Bank von einer REGIONAL(1)-Datei auf eine CONSECUTIVE-Datei, also beispielsweise auf ein Band. In diesem Zustand ist allerdings kein Datenbankzugriff möglich. In der FROM-Klausel wird der DD-Name der REGIONAL(1)-Datei (der Direct-Access-Datei) angegeben, in der TO-Klausel der der sequentiellen Datei. Zur Erhöhung der Effektivität kann in der FACTOR-Klausel ein Blockungsfaktor spezifiziert werden, der Default-Wert ist 1.

Beispiel

```
SET POOL(DATAPOOL) NEXT(120);

RETRIEVE POOL(DATAPOOL) TO(RECORD);
      (liest ab Offset 120)
```

Syntax

```
SET POOL(ref) NEXT(offset-next-io);

    offset-next-io::=bin-fixed-31-expr.
```

Erläuterung

SET ist eine Datapool-Operation, die es erlaubt, den Ein-Ausgabe-Zeiger explizit zu setzen, um mit der sequentiellen Ein-Ausgabe an einem beliebigen Punkt der Datei beginnen zu können.

Beispiel

```
DCL 1 RECORD,
      2 NAME CHAR (10),
      2 ALTER BIN FIXED (15),
      2 GEHALT DEC FLOAT (6);
 DESC PERS POOL;
OPEN POOL (PERS).....
/  Der Pool PERS enthalte 100 Records des Typs RECORD /

SORT POOL (PERS) LRECL (16) CHAR (10);
SET POOL (PERS) NEXT (o);
CALL PRINT;
SORT POOL (PERS) LRECL (16) OFFS (10) BFI 15;
SET POOL (PERS) NEXT (0);
CALL PRINT;

PRINT: PROC;
   DO i = 1 TO 100;
    RETRIEVE POOL (PERS) TO (RECORD);
    PUT SKIP LIST (RECORD);
   END; END PRINT;
```

Syntax

SORT

```
SORT POOL (poolref) [FROM (first)] [TO(last)]
    [LRECL(lrecl)] [OFFS(offs)] [DESCENDING] type;
     type:: = BFI15|BFI31 | BFL21 |BFL53 |
              BFL109|DFI5|
              DFL6|DFL16|DFL33|
              CHAR(string)

     first, last, lrecl, offs::=bin-fixed_31_expr.
     string::= bin_fixed_15_expr.
```

Erläuterung

Der Datapool 'poolref' wird vom Record 'first' bis zum Rekord 'last' sortiert. Der Default-Wert für 'first' ist 1, der für 'last' der letzte Satz der Datei. Als Recordlänge wird 80 angenommen, wenn nicht durch die LRECL-Klausel ein anderer Wert angegeben wird.
Der Sortierschlüssel kann an beliebiger Stelle im Rekord stehen, sein Offset innerhalb des Records wird in Bytes in der OFFS-Klausel angegeben (Default:0).

Mögliche Sortierschlüsseltypen sind:

BIN FIXED(15), BIN FIXED(31), BIN FLOAT(21), BIN FLOAT(53), BIN FIXED(109), DECIMAL FIXED(5), DECIMAL FLOAT(6), DECIMAL FLOAT(16), DECIMAL FLOAT(33), CHAR(bin)_fixed_15_expr.)

Die Datei wird aufsteigend sortiert, wenn nicht DESCENDING angegeben wird.

Beispiele

```
STATUS STATISTIC;
STATUS STRUCTURE BANK(BANK1);
STATUS DUMP BANK(BANK1);
STATUS ATTRIBUTE POOL(DATAPOOL)
       MAX(LENGTH_OF_POOL)
       TYP(BANK_OR_CAT_OR_DATA)
       NEXT(OFFS_NEXT_REC_USING_SEQUENTIAL_IO)
       ACT(NUMBER_OF_USED_BYTES);
STATUS OPEN;
STATUS SPACE PRINT BANK(BANK1);
STATUS SPACE(ALL,FREE_SPACE,MAX_FREE_ELEMENT,UNUSED,
       FREE_MANAGEMENT_ENTRIES) BANK(BANK1);
```

Syntax

```
STATUS option;
    option::=opt1 | opt2 | opt3 | opt4 | opt5 | opt6
    opt1::=STATISTIC[FILE(printfile)]
    opt2::=STRUCTURE[OF] [BANK](bankref)[FILE(printfile)]
    opt3::=DUMP[BANK](bankref)
    opt4::=ATTRIBUTE[TYP(v1)] [ACT(v2)] [MAX(v3)] [NEXT(v4)]
                                                  typ(ref);
    opt5::=OPEN
    opt6::=SPACE [(v1,...,v5)] BANK(bankref) |
          SPACE PRINT [FILE(printfile)] BANK(bankref)
    v1,...v5::=bin-fixed-31-variable
    typ::=BANK |POOL
```

Erläuterung

STATUS ist eine Operation, die dem Benutzer Informationen über den Zustand einer Datenbank, eines Pools oder über die Systemkern-aktivitäten liefert. STATUS hat eine Reihe von Optionen, durch die verschiedene Teilinformationen angesprochen werden.

STATUS STATISTIC liefert eine Statistik über die Häufigkeit der Benutzung einzelner Operationen wie OPEN, CLOSE, STORE, RETRIEVE, PUTSTRING, GETSTRING, PL/1-I/O, wobei für OPEN und CLOSE unterschieden wird zwischen Banken und Pools und zudem registriert wird, wieviel Banken bzw. Pools zu irgendeinem Zeitpunkt maximal eröffnet waren (PUTSTRING,GETSTRING sind implizite I/O-Operationen).

STATUS STRUCTURE druckt für eine Datenbank eine Tabelle sämtlicher Datenbank-Strukturelemente (BANK, CATALOGPOOL, DATAPOOL) in Preorderfolge, wobei für jedes Element die Hierarchiestufe, der Name, der Typ, die maximale Länge (Zahl der Einträge bzw.der Bytes) und die aktuelle Länge angegeben werden.

STATUS
(cont.)

STATUS DUMP liefert einen lesbaren Dump einer Datenbank, der alle Sekundär-Speicherelemente mit ihren momentanen Werten umfaßt (nicht allerdings die Benutzerdaten in einem Datapool).

STATUS ATTRIBUTE liefert für Banken und Pools den Typ, die maximale Länge (Zahl der Einträge oder Bytes), die aktuelle Länge und bei DATAPOOLS den Wert des Ein-Ausgabe-Zeigers, d.h. den Pool-Offset des bei sequentieller Ein-Ausgabe folgenden Records.
(Bedeutung der Typ-Werte:
1: BANK 2: CATALOGPOOL 3: DATAPOOL)

STATUS OPEN druckt sämtliche Arbeitsspeicher-Kontrollblöcke in lesbarer Form auf einen PRINT-FILE.

STATUS SPACE ermittelt für eine Datenbank die Sekundärspeicher-Platzverteilung. Die Operation liefert den gesamten für die Bank zur Verfügung stehenden Platz (v_1), den gesamten noch freien Platz (v_2), den Umfang des größten freien Elements (v_3), den freien Platz hinter dem Datenbankfüllstand (v_4) und die Zahl der noch freien Einträge in den Platzverwaltungslisten (v_5). Diese Daten ermöglichen eine Aussage über die Speicherfragmentierung. Wird die PRINT-Option verwendet, so wird eine Tabelle dieser Werte ausgedruckt. Der Default-Wert für alle Print-Files ist SYSPRINT.

Beispiele

```
STORE POOL(DATAPOOL) FROM(RECORD);
STORE POOL(D) FROM(I) TO(POOL_OFFSET);
DCL P PTR INIT(ADDR(STRING));
DCL STRING CHAR(500);
STORE POOL(D) FROM(P) LENGTH(20);
STORE POOL(POOL1) FROM(I) SET(OFFSET_OF_THIS_RECORD);
STORE POOL(POOL2) FROM(I) TO(200) NEXT(OFFSET_OF_NEXT_RECORD);
```

Syntax

```
STORE POOL(ref)option[option2]2o;

      option1::=FROM(variable)|
                FROM(pointer)LENGTH(length)
      option2::=SET(adr)|TO(offset)[NEXT(adr)]
      offset,length::=bin-fixed-31-expr.
      adr::=bin-fixed-31-variable
```

Erläuterung

Dieser Befehl dient dem Schreiben von Benutzerdaten aus der Variablen 'variable' (oder aus dem durch 'pointer' und 'length' beschriebenen Speicherbereich) der FROM-Klausel in einen Datapool, d.h. einen Pool, der mit OPEN POOL...DATA eröffnet wurde. Der Pool muß die Attribute OUTPUT oder UPDATE haben.
Beim Zugriff ist ein Datenpool für den Benutzer gekennzeichnet durch seine Länge in Bytes, seinen Füllstand und bei sequentiellem Zygriff durch einen Zeiger, der das nächste Byte bezeichnet. Dieser Zeiger zeigt nach einem OPEN auf das erste Byte, bei einem OPEN MOD auf das erste freie Byte. STORE bietet die Möglichkeit, sequentiell zu arbeiten oder direkt auf einzelne Bytes oder Bytegruppen zuzugreifen. Bei einem sequentiellen Zugriff (STORE...FROM) wird der Ein-Ausgabe-Zeiger jeweils um die Recordlänge weitergesetzt.

Beim direkten Zugriff werden die Bytes des Pools durch ihren Offset vom Poolbeginn adressiert. Zur Kennzeichnung des direkten Zugriffs wird eine zusätzliche Befehlsklausel verwendet, in der die Adresse angegeben wird (STORE...TO(adr)). Der Ein-Ausgabe-Zeiger wird nicht verändert.
In der SET-Klausel wird eine Variable spezifiziert, in der der Offset des gerade bearbeiteten Records zurückgeliefert wird. Sie ist vor allem für sequentielles Arbeiten gedacht, um danch evtl. direkt zugreifen zu können. Mit der NEXT-Klausel wird der Offset des auf den bearbeiteten Record folgenden Bytes geliefert.

```
TEXT:PROC OPTIONS(MAIN) REGENT(NODA,BANK);
 ENTER DABAL;
 ON ERROR BEGIN;
  STATUS OPEN;
  CALL PLIDUMP('TFSNHB','ERROR');
 END;
 DESC BANK1 BANK;
 DESC POOL1 POOL;
 DESC POOL2 POOL;
 INIT BANK BANK1;
 OPEN BANK(BANK1)UPDATE;
INIT POOL(POOL1)TITLE('POOL1')IN BANK(BANK1)DATA SPACE(1000);
INIT POOL(POOL2)TITLE('POOL2')IN BANK(BANK1)DATA SPACE(1000);
 OPEN POOL(POOL1)TITLE('POOL1')IN BANK(BANK1)UPDATE;
 OPEN POOL(POOL2)TITLE('POOL2')IN BANK(BANK1)UPDATE;
 DCL CARD CHAR(80) INIT(' KARTE     POOL1');
 DCL PIC PICTURE'99';
 DO PIC=1 TO 5;
  SUBSTR(CARD,8,2)=PIC;
  STORE POOL(POOL1)FROM(CARD);
 END;
 PRINT POOL(POOL1) ID('NACH STORE');
 RENUM POOL(POOL1);
 PRINT POOL(POOL1) ID('NACH RENUM');
 DCL NUM BIN FIXED(31);
 FIND POOL(POOL1) STRING ('KARTE 04') SET(NUM) V;
 CHANGE CARD NUM IN POOL(POOL1) REPLACE COL 20 25 BY 'ABCDEF' V;
 CHANGE CARD 1 IN POOL(POOL1) REPL STRING (15)' ' BY (15)'X' V;
 CHANGE CARD 2 TO 3 IN POOL(POOL1) CONT COL 16 WITH '1234567890' V;
 CHANGE CARD NUM IN POOL(POOL1) REPL STRING 'CDE' BY '1234567890' V;
 PRINT POOL(POOL1) ID('NACH CHANGE CARD');
 PRINT POOL(POOL1) ID('DECH OPTION') DECK(2,3);
 PACK POOL(POOL1);
 PRINT POOL(POOL1)ID('NACH PACK');
 EXPAND POOL(POOL1) POOL(POOL2);
 RENUM POOL(POOL2) 1 1;
 PRINT POOL(POOL2) ID('POOL2');
 END DABAL ;
 PUT SKIP LIST ('                    TEXT TEST FINISHED');
 END TEXT;
```

```
STATUS : PROC OPTIONS(MAIN)REGENT(NODA,BANK);
 ENTER DABAL;
  ON ERROR BEGIN;
    STATUS OPEN; END;
DESC BANK1 BANK;
  DESC POOL1 POOL;
   INIT BANK(BANK1);
  OPEN BANK(BANK1) UPDATE;
INIT POOL(POOL1)TITLE('TEXT')IN BANK(BANK1)SPACE(10000) DATA;
 OPEN POOL(POOL1)TITLE('TEXT')IN BANK(BANK1)UPDATE KEEP;
 DCL CARD CHAR(80)INIT(' KARTE ');
 DCL PIC PICTURE'99';
 DO PIC=1 TO 50;
  SUBSTR(CARD,8,2)=PIC;
 STORE POOL(POOL1)FROM(CARD);
  END;
  STATUS STATISTIC;
  STATUS OPEN;
  STATUS STRUCTURE BANK(BANK1);
  STATUS DUMP BANK(BANK1);
  STATUS SPACE PRINT BANK(BANK1);
  STATUS ATTRIBUTE POOL(POOL1) TYP(TYP) ACT(ACT) MAX(MAX) NEXT(NEXT);
 DCL(TYP,MAX,ACT,NEXT)BIN FIXED(31);
 PUT SKIP DATA(TYP,MAX,ACT,NEXT);
  STATUS STATISTIC;
  END DABAL;
  PUT SKIP LIST('                STATUS TEST  FINISHED');
 END STATUS;
```

```
VERW: PROC OPTIONS(MAIN) REGENT(NODA,BANK);
 ENTER DABAL;
  ON ERROR BEGIN;
   STATUS OPEN;
   STATUS DUMP BANK(BANK1);
   CALL PLIDUMP('TFSNHB','ERROR');
  END;
 DESC BANK1 BANK;
 DESC POOL(9) POOL;
 INIT BANK(BANK1);
 OPEN BANK(BANK1)UPDATE;
 DCL PIC PICTURE'9';
 DO I=1 TO 5;
  PIC=I;
INIT POOL(POOL(I))TITLE('POOL'||PIC)CAT IN BANK(BANK1)SPACE(2);
  OPEN POOL(POOL(I))TITLE('POOL'||PIC) UPDATE CAT IN BANK(BANK1);
 END;
 /* CLOSE DISP = DELETE */
 STATUS OPEN;
INIT POOL(POOL(6))TITLE('DRUCK')SPACE(2000)IN POOL(POOL(1));
 OPEN POOL(POOL(6))TITLE('DRUCK')IN POOL(POOL(1))UPDATE;
INIT POOL(POOL(7))IN POOL(POOL(1))SPACE(3000)TITLE('TEMP');
 OPEN POOL(POOL(7))TITLE('TEMP')IN POOL(POOL(1))UPDATE ;
 STATUS DUMP BANK(BANK1);
 DO I=2 TO 4;
  COPY STRUCTURE POOL(POOL(1))TO POOL(POOL(I));
 END;
 /* DIE CATALOGPOOLS POOL1...POOL5 ENTHALTEN JEWEILS
    ZWEI DATAPOOLS MIT DEN NAMEN 'DRUCK' UND 'TEMP' */
 DCL PIC2 PICTURE'99';
 DCL CARD CHAR(80) INIT(' KARTE');
 DO PIC2=1 TO 20;
  SUBSTR(CARD,8,2)=PIC2;
  STORE POOL(POOL(6))FROM(CARD);
 END;
 COPY DATA POOL(POOL(6))TO FILE(TSO);
 DCL TSO FILE;
 STATUS STRUCTURE BANK(BANK1);
 STATUS SPACE PRINT BANK(BANK1);
 CLOSE BANK(BANK1);
 OPEN BANK(BANK1)UPDATE;
 OPEN POOL(POOL(2))TITLE('POOL2')  DELETE UPDATE IN BANK(BANK1);
 CLOSE POOL(POOL(2));
 STATUS STRUCTURE BANK(BANK1);
 STATUS SPACE PRINT BANK(BANK1);
 STATUS OPEN;
 CLOSE BANK(BANK1);
 COMPRESS BANK(BANK1) INOUT('INOUT') NEWBLK(2000);
 OPEN BANK(BANK1) UPDATE;
 STATUS OPEN;
 STATUS SPACE PRINT BANK(BANK1);
 CLOSE BANK(BANK1);
 SAVE FROM('BANK1') TO('TAPE')FACTOR(5);
 RESTORE FROM('TAPE')TO('BANK2');
 DESC BANK2 BANK;
 OPEN BANK(BANK2)UPDATE;
 STATUS STRUCTURE BANK(BANK2);
 STATUS SPACE PRINT BANK(BANK2);
 OPEN POOL( POOL(1))TITLE('POOL1') OLD KEEP UPDATE
  CATALOG  IN BANK(BANK2);
 CHANGE NAME('CHANGED') POOL(POOL(1));
 CHANGE LENGTH(100) POOL(POOL(1));
 STATUS STRUCTURE BANK(BANK2);
 END DABAL;
 PUT SKIP LIST('              VERW TEST  FINISHED');
 END VERW;
```

```
STORE: PROC OPTIONS(MAIN)REGENT(NODA,BANK);
ENTER DABAL;
  ON ERROR BEGIN;
   STATUS OPEN;
   CALL PLIDUMP('TSNH','ERROR');
  END;
DESC BANK1 BANK FILE(FILE1) TITLE('BANK1');
DESC POOL1 POOL;
INIT BANK(BANK1);
OPEN BANK(BANK1)OUTPUT;
INIT POOL(POOL1)TITLE('POOL1')IN BANK(BANK1)DATA SPACE(10000);
OPEN POOL(POOL1) IN BANK(BANK1) TITLE('POOL1') OUTPUT KEEP;
DCL CARD CHAR(80)INIT(' KARTE    ABCDEFGHIJKLMNOPQRSTUVWXYZ');
 DCL PIC PICTURE'99';
 DO PIC=1 TO 50;
  SUBSTR(CARD,8,2)=PIC;
  STORE POOL(POOL1)FROM(CARD);
 END;
/* DER POOL ENTHAELT 50 KARTEN */
 DCL PIC2 PICTURE'99999999';
 DCL OFFS BIN FIXED(31);
 OFFS=72;
 DCL NUMBER CHAR(8);
 DO PIC2=10 TO 500 BY 10;
  NUMBER=PIC2;
  STORE POOL(POOL1)FROM(NUMBER) TO(OFFS);
  OFFS=OFFS+80;
 END;
 CLOSE BANK(BANK1);
 OPEN BANK(BANK1)INPUT;
 DCL FILE1 FILE ENV(REGIONAL(1),F);
 OPEN POOL(POOL1)TITLE('POOL1') OLD KEEP DATA IN BANK(BANK1);
 ON ENDFILE(FILE1) GO TO WEITER;
 LOOP:
      RETRIEVE POOL(POOL1)TO(CARD);
      PUT SKIP LIST(CARD);
      GO TO LOOP;
 WEITER:;
 RETRIEVE POOL(POOL1) TO(CARD)FROM(7*80);
 PUT SKIP LIST('DIE 8. KARTE: ',CARD);
 ON ENDFILE(FILE1) GO TO W;
 SET POOL(POOL1) NEXT(3*80);
 LOOP2:
       RETRIEVE POOL(POOL1)TO(CARD);
       PUT SKIP LIST(CARD);
       GO TO LOOP2;
 W:;
END DABAL;
PUT SKIP LIST('              STORE TEST  FINISHED');
END STORE;
```

9.2 EDIT-Handbuch

Einführung

Das EDIT-Subsystem erlaubt bei der Bearbeitung von Textdateien im Kartenformat auch das Einfügen und Löschen von Karten. Dazu legt das Subsystem eine Arbeitsdatei auf der Betriebssystemdatei QQWORKDA (DD-Name) an. Die zu bearbeitende Textdatei muß dem Subsystem mit dem ENTER-Befehl übergeben werden:

```
ENTER EDIT POOL (textdatei) ;
```

Die durchgeführten Änderungen werden mit dem Befehl SAVE auf die Textdatei übertragen.

Die Karten in der Arbeitsdatei werden durch eine ein- bzw. zweistufige Folgenummer bezeichnet, deren Stufen durch Doppelpunkte getrerrt sind: z.B. 10:5 . Die erste Folgenummer ist die Folgenummer der Karten im Ausgangs-Zustand. Werden hinter einer Karte mit der Folgenummer f Karten eingefügt, so wird die relative Folgenummer ff einer Karte innerhalb des eingefügten Pakets durch f qualifiziert: z.B. f:ff. Basis für die Bezeichnungen bleiben also bei allen Operationen die Folgenummern der Ausgangsdatei, die unverändert bleiben. Vor der ersten Karte eingefügte Karten werden mit 0 qualifiziert (z.B. 0:5).

Das EDIT-Subsystem wird innerhalb des DABAL-Bereichs aufgerufen:

```
ENTER DABAL;

   DESC POOL1 POOL;

   DESC POOL2 POOL;

   OPEN .......
     .
     .
   ENTER EDIT POOL(POOL1);
      MERGE POOL(POOL2) TO 10;
      SAVE;
   END EDIT;

   CLOSE POOL(POOL1);

   CLOSE POOL(POOL2);

END DABAL;
```

Die Arbeit mit EDIT erfordert im G-Step die DD-Karte:

```
//G.QQWORKDA DD UNIT=SYSDA,DISP=(NEW,DELETE),
              SPACE=(TRK,n),DCB=BLKSIZE=m
```

Beispiele

siehe auch DABAL - CHANGE.

```
DCL(N1,N2,N3,N4) BIN FIXED(31),(A,B,NEWSTRING)CHAR(12);

CHANGE 1 TO 10 REPL STR 'AB' BY 'XYZ';

CHANGE 1:5 CONT COL N1 NEWSTRING;

CHANGE N1:N2 TO N3:N4 CONT STR A B;

FIND STRING('2803') SET(N1:N2) POS(73,80);

CHANGE N1:N2 CONT COL 5 '××××';
```

Syntax

```
CHANGE begin [TO end] option [VERIFY];
    option::=option1 | option2 | option3 | option4
    option1::=REPLACE STRING oldstring [BY] newstring
    option2::=REPLACE COLUMN first last [BY] newstring
    option3::=CONTINUE STRING beginstring [WITH] newstring
    option4::=CONTINUE COLUMN column [WITH] newstring
      oldstring,newstring::=char-expr.
      first,last column::=bin-fixed-15-expr.
      begin,end::=sequ1 [:sequ2]
      sequ1,sequ2::=bin-fixed-31-expr.
```

Erläuterung

Mit CHANGE werden in der Arbeitsdatei die Karten 'begin' bis 'end' geändert. Dabei gibt es vier Möglichkeiten:

- Ersetzen der Zeichenkette oldstring durch die Zeichenkette newstring.

- Ersetzen der Zeichen zwischen den Spalten first und last durch die Zeichenkette newstring.

- Fortsetzen einer Zeichenkette ab beginstring durch die Zeichenkette newstring.

- Fortsetzen einer Zeichenkette ab Spalte column durch die Zeichenkette newstring.

Die VERIFY-Option bewirkt das Ausdrucken der geänderten Zeilen auf SYSPRINT.

Beispiele

```
DELETE 5;

DELETE 5:3;

DELETE 5  5:6;

DELETE N1:N2  N3:N4;

DELETE 1:2  3;

DELETE 0:1;
```

Alle vor der ersten Karte eingefügten Karten und die erste Karte des Ursprungspakets werden gelöscht.

Syntax

```
DELETE begin [TO] [end];
   begin,end::=sequ1 [:sequ2]
   sequ1,sequ2::=bin-fixed-31-expr.
```

Erläuterung

In der Arbeitsdatei werden die Karten 'begin' bis 'end' oder die Karte 'begin' gelöscht.

Beispiele

```
FIND STRING('XY') SET(N1:N2);

FIND STRING('25300') SET(N1:N2) POS(73,80);

FIND STRING(SEARCHSTRING) SET(N1:N2) POS(3,15)
                    DECK(CARD1,CARD_N);

FIND STRING('×××') SET(N1:N2) VERIFY;
```

Syntax

```
FIND [opt1]2 2 [opt2]3 0;

   opt1::=STRING(string) | SET(set1:set2)

   opt2::=POS(col1,col2) | DECK(begin,end)| VERIFY

          string::=char-expr.

          begin,end::=sequ1[:sequ2]

          sequ1,sequ2::=bin-fixed-31-expr.

          col1,col2::=bin-fixed-15-expr.

          set1,set2::=bin-fixed-31-var
```

Erläuterung

Der FIND-Befehl weist dem in der SET-Klausel benannten Variablenpaar die Folgenummer der ersten Karte der Arbeitsdatei zu, die die Zeichenkette 'string' enthält. Die Suche kann eingeschränkt werden auf die Spalten expr1 bis expr2 und die Karten expr4 bis expr5. Die Option VERIFY bewirkt das Drucken der gefundenen Zeile auf SYSPRINT.
Wird die gesuchte Zeichenkette nicht gefunden, so werden set1, set2 auf -1 gesetzt.

Beispiele

```
INSERT 20 '1.INSERTED CARD';

INSERT 20:1 '2.INSERTED CARD';

INSERT N1:N2  NEWCARD;
```

Syntax

INSERT sequ string;

sequ::=bin-fixed-31-expr [:bin-fixed-31-expr.]

string::=char-expr.

Erläuterung

In der Arbeitsdatei wird hinter der Karte mit der Folgenummer sequ eine neue Karte mit der Zeichenkette string eingefügt.

Beispiele

```
OPEN POOL(TEXT_POOL) TITLE('TEXT')
        OLD KEEP INPUT DATA;

MERGE POOL(TEXT_POOL) TO(25);

MERGE POOL(TEXT_POOL) DECK(20,30) TO(N1:N2);
```

Syntax

```
MERGE POOL(pool) [DECK(begin,end)] TO(sequ);

      begin,end,sequ1,sequ2::=bin-fixed-31-expr.

      sequ::=sequ1 [:sequ2]
```

Erläuterung

In der Arbeitsdatei werden hinter der Karte mit der Folgenummer 'sequ' die Karten 'begin' bis 'end' der Textdatei 'pool' eingefügt. Fehlt die DECK-Klausel, so wird die gesamte Datei 'pool' eingefügt. Die Datei 'pool' muß im umfassenden Subsystem DABAL eröffnet worden sein.

Beispiele

```
PRINT;
PRINT ID('UEBERSCHRIFT');
PRINT DECK(CARD1:CARD2,CARD3:CARD4);
PRINT FILE(PFILE);
PRINT DECK(1,100) ID(TEXT) FILE(SYSPRINT);
```

Syntax

```
PRINT [option] 3,
                o;

  option::=DECK(begin,end)|
           FILE(printfile)|
           ID(char-expr)

  begin,end::=sequ1 [:sequ2]

  sequ1,sequ2::=bin-fixed-31-expr
```

Erläuterung

PRINT druckt die Arbeitsdatei oder, falls die DECK-Klausel verwendet wird, die Karten begin bis end auf SYSPRINT oder auf den in der FILE-Klausel angegebenen PRINT-File. Mit der ID-Klausel kann eine Überschrift spezifiziert werden.

Beispiele

```
SAVE;

SAVE PRINT;

SAVE PRINT RENUM(1,1);

SAVE RENUM(N1,INCR);
```

Syntax

SAVE [option]$_{0}^{2}$;

option::=PRINT | RENUM (first,incr)

first,incr::=bin-fixed-31-expr.

Erläuterung

SAVE kopiert die Arbeitsdatei, in der die Änderungen vorgenommen wurden, in die Ausgangsdatei zurück. Dazu kann die Datei neu numeriert werden (RENUM), wobei die Numerierung bei 'first' beginnt und 'incr' das Numerierungsinkrement bedeutet (Default:10,10). Die PRINT-Option veranlaßt einen Kontrollausdruck auf SYSPRINT.

```
EDIT:PROC OPTIONS(MAIN) REGENT(NCCA,BANK);
 ENTER DABAL;
 PUT SKIP LIST('EDIT 1');
 ON ERROR BEGIN;
  STATUS OPEN;
  CALL PLIDUMP('TESNHB','ERROR');
 END;
 DESC BANK1 BANK;
 DESC POOL1 POOL;
 DESC POOL2 POOL;
 INIT BANK BANK1;
 OPEN BANK(BANK1)UPDATE;
INIT POOL(POOL1)TITLE('POOL1')IN BANK(BANK1)SPACE(1000);
 OPEN POOL(POOL1)TITLE('POOL1')IN BANK(BANK1)UPDATE;
INIT POOL(POOL2)TITLE('POOL2')IN BANK(BANK1)SPACE(1000);
 OPEN POOL(POOL2)TITLE('POOL2')IN BANK(BANK1)UPDATE;
 DCL CARD CHAR(80) INIT(' KARTE     POOL1');
 DCL PIC PICTURE'99';
 DO PIC=1 TO 5;
  SUBSTR(CARD,8,2)=PIC;
  STORE POOL(POOL1)FROM(CARD);
 END;
 PRINT POOL(POOL1) ID('NACH STORE');
 RENUM POOL(POOL1);
 COPY DATA POOL(POOL1) TO POOL(POOL2);
 CHANGE CARD 1 TO 5 IN POOL(POOL2) REPL STRING 'POOL1' 'POOL2';
 PRINT POOL(POOL2) ID ('POOL2');

 ENTER EDIT POOL(POOL1);

  INSERT 2 ' INSERTED CARD MIT SEQU_NR 2:1';
  MERGE POOL(POOL2) DECK(2,4) TO(4);
  DCL ADR BIN FIXED(31)BASED;
 PRINT;
  PRINT  DECK(1:0,5:0);
  DELETE 4:2 ;
 DCL (NUM1,NUM2)BIN FIXED(31);
 FIND STRING('POOL2') SET(NUM1:NUM2)V;
 PUT SKIP DATA(NUM1,NUM2);
 FIND STRING('KARTE 04  POOL2' ) SET(NUM1:NUM2)V;
 PUT SKIP DATA(NUM1,NUM2);
 FIND STRING('POOL1')SET(NUM1:NUM2)V;
 PUT SKIP DATA(NUM1,NUM2);
 FIND STRING('KARTE 05')SET(NUM1:NUM2)V;
 PUT SKIP DATA(NUM1,NUM2);
 FIND STRING('########')SET(NUM1:NUM2)V;
 PUT SKIP DATA(NUM1,NUM2);
 FIND STRING('#####')DECK(2,5) SET(NUM1:NUM2) V ;
 PUT SKIP DATA(NUM1,NUM2);
FIND STRING('KARTE')POS(50,60)SET(NUM1:NUM2) V;
PUT SKIP DATA(NUM1,NUM2);
  PRINT;
  CHANGE 4:2 REPLACE STRING 'POOL2' BY 'POOL2 CHANGED';
  CHANGE 4:2 REPLACE STRING 'HANGE' '#' V;
  PRINT;
  SAVE;
  END EDIT;
  PRINT POOL(POOL1)ID('NACH EDIT');
 PUT SKIP LIST('                    EDIT TEST  FINISHED');
 END DABAL ;
 END EDIT;
```

K A P I T E L 10

PLR - Handbuch

Beispiel

```
DCL  P  PTR;
DCL  A  DESCRIPTOR;
DCL  X  DYNAMIC (A);
DEFINE A,(10) DESCRIPTOR;
DEFINE A(5), (10) STEP(4) AS (X);
ATTRIBUTE A(N1) NUMBER(N2) STEP(N3) PRIORITY(N4);
A(5,12) -> X=2.;
FIX A(5) SET(P) ON(1);
ATTRIBUTE A(5) TYPE(M1) NUMBER(M2) STEP(M3) PRIORITY(M4);
RESET a;
```

Ergebnisse durch ATTRIBUTE:

N1: 0

N2: 10

N3: 0

N4: 2, da die Teildatenstruktur referiert wird

M1: 2

M2: 15, da eine Erweiterung um 5 stattfand

M3: 5, da 4 als Schrittweite in STEP nicht akzeptiert wird

M4: 6, da FIX - Anweisung ausgeführt wurde

Syntax

ATTRIBUTE **substructure-reference** option [option] * ;

option::= TYPE(n_1) |
NUMBER(n_2) |
STEP(n_3) |
PRIORITY(n_4)

n_1 ..... n_4 BIN FIXED(15) - Variable.

Erläuterung

1) Es wird in den Variablen n_1 bis n_4 folgende Information gespeichert:

n_1 0 bei Descriptorvektor
2 bei Datenvektor

n_2 Anzahl der Vektorelemente

n_3 Schrittweite (STEP) für Vektorverlängerung

n_4

Wert	Bedeutung		
0	UNLOCKED	RELEASED	LOW
1	UNLOCKED	RELEASED	HIGH
2	UNLOCKED	UNRELEASED	LOW
3	UNLOCKED	UNRELEASED	HIGH
4	LOCKED	RELEASED	LOW
5	LOCKED	RELEASED	HIGH
6	LOCKED	UNRELEASED	LOW
7	LOCKED	UNRELEASED	HIGH

Beispiel

```
DCL (N1.....N9,M1.....M9) BINARY FIXED(31);
DCL BANK1 BANK;
DCL POOL1 POOL;
DCL POOL2 POOL;
DCL CARD CHAR(80) INIT('KARTE 1');
INITIATE BANK (BANK1) FREELIST(10) NODELETELIST LIST PAGE(10)
SPACE(10);
OPEN BANK (BANK1) OUTPUT;
OPEN POOL (POOL1) TITLE('POOL1') IN (BANK1) OUTPUT DELETE NEW
SPACE (1000) DATA;
OPEN POOL (POOL2) TITLE('POOL2') IN (BANK1) NEW SPACE(2000) DATA
OUTPUT;
STORE POOL (POOL1) FROM (CARD);
ATTRIBUTE POOL1 TYPE(N1) MAX(N2) ACT(N3) NEXT(N4) SPACE(N5,N6,
N7, N8, N9);
CLOSE POOL (POOL1);
ATTRIBUTE BANK1 TYPE(M1) MAX(M2) ACT(M3) NEXT(M4) SPACE(M5,M6,
M7, M8, M9);
```

mit:

```
//BANK1 DD UNIT=SYSDA,SPACE=(1000,4),
// DCB=BLKSIZE=1000,DISP=(NEW,DELETE)
```

Ergebnisse durch ATTRIBUTE:

N1:	3	M1:	1
N2:	1000	M2:	10
N3:	80	M3:	2
N4:	80	M4:	0
N5:	4000	M5:	4000
N6:	220	M6:	220
N7:	220	M7:	1420
N8:	220	M8:	1200
N9:	9	M9:	8

Syntax

ATTRIBUTE poolreference option [option] * ;

option::= TYPE(n_1) |
MAX(n_2) |
ACT(n_3) |
NEXT(n_4) |
SPACE(n_5,n_6,n_7,n_8,n_9)

n_1 bis n_9 BIN FIXED(31) Variable

Erläuterung

1) Es wird in den Variablen n_1 bis n_9 folgende Information gespeichert:

n_1: 1 BANK
2 CATALOGPOOL
3 DATAPOOL

n_2: maximale Zahl der Einträge bei n_1 ¬= 3
Länge in Bytes bei n_1 = 3

n_3: aktuelle Zahl der Einträge bei n_1 ¬= 3
Zahl der beschriebenen Bytes (Füllstand) bei n_1 = 3

n_4: 0 bei n_1 ¬= 3
Pooloffset für nächsten E/A-Vorgang bei sequentieller E/A für n_1 = 3

n_5: gesamter für die BANK zur Verfügung stehender Platz in Bytes

n_6: freier Platz nach Füllstand, d.h. bisher noch nicht verwendet.

n_7: gesamter freier Platz in Bytes

n_8: Länge des größten freien Elementes in Bytes

n_9: Anzahl der noch freien Einträge in den Platzverwaltungslisten.

Beispiel

```
A: PROC(P)REGENT;
   DCL P POINTER;
   DCL 1 DATENBANK BASED(P),
       2 BANKEN(3) BANK;
   DCL BANK1 BANK INIT(P->BANKEN(1));
   DCL POOL POOL;
   INITIATE BANK(BANK1);
   OPEN BANK(BANKEN(2));
   OPEN POOL(POOL) IN(BANKEN(2));
   BANK1 = BANKEN(3);
   OPEN BANK(BANK1);
   ----
END
```

bankreference

Syntax

Eine bankreference sieht aus wie eine POINTER-Referenz

Erläuterung

1) Eine bankreference kann nur stehen in der INIT-Option von DECLARE BANK, als aktueller Parameter und in den Anweisungen

   OPEN BANK
   CLOSE BANK
   INITIATE BANK
   OPEN POOL (IN-Option)
   ATTRIBUTE

   sowie in Zuweisungen

Beispiele

zur Anwendung der Hilfsroutinen siehe Kap.6.8 (Seite 6-13 bis 6-33)

```
CALL QQDBDD
     QQDPOL
     QQDPSZ
     QQDREO
     QQDSTR
     QQDSTX
     QQMMAP
     QQMREO
     QQSTIME
     QQTTIME
```

Erläuterung

1) Diese Hilfsroutinen sind im PLR ebenso wie in jeder Subsystemsprache verfügbar. Ihre Wirkung und ihr Aufruf sind im Systemhandbuch Kap.6.8(Seite 6-13 bis 6-33) erläutert.

2) Bei Anwendung in PLR benötigen lediglich die Routinen QQSTIME und QQTTIME eine Entry-Deklaration. Die Entry-Deklarationen der übrigen Hilfsroutinen werden von PLR implizit bereitgestellt.

Beispiel

```
DCL A DESCRIPTOR;
DCL SUB1 ENTRY(,,DESCRIPTOR),
    SUB2 ENTRY(,,,),
    SUB3 ENTRY;

CALL SUB1(X,Y,A);
CALL SUB2(X,Y,Z,A);
CALL SUB3(X,A);

UNTER_PROG:PROC(R);
DCL(X(*),R) DESCRIPTOR
SWITCH X(2)( ),R;
END UNTER_PROG;
```

DESCRIPTOR
parameter

Erläuterung

1) Soll an ein Unterprogramm ein Deskriptor als Parameter übergeben werden, so muß die zugehörige ENTRY-Deklaration entweder an der betreffenden Argumentenstelle

   - das Attribut DESCRIPTOR oder
   - kein Attribut enthalten oder
   - die Parameterdeskriptorliste muß ganz fehlen.

2) Soll in einem Unterprogramm ein Deskriptor als Parameter übernommen werden, so muß dieses Unterprogramm die entsprechende Deklaration enthalten.

3) Deskriptoren können nicht im RETURNS-Attribut auftreten.

Beispiel

```
DCL 1 A BASED(Q),
    2 N, 2 X(3)DESCRIPTOR,
    Y DESCRIPTOR,
    P POINTER;

DCL DATEN(1000) DYNAMIC;
    P -> A.X(2) (4,3,1) -> DATEN(I)

CALL UNTER_PROG(Y,P->X);
```

basedescriptor
reference

Erläuterung

1) Ein Basedeskriptor wird referiert, indem sein ausreichend qualifizierter und indizierter Name verwendet wird.

2) Eine Basedeskriptorreferenz kann auftreten

- als Qualifikation einer Substructure-Referenz,
- als Qualifikation einer Dynamic-Element-Referenz,
- als aktueller Parameter.

Beispiel

Siehe OPEN BANK

CLOSE
BANK

Syntax

CLOSE BANK (bankreference);

Erläuterung

1) Die Bank und alle evtl. in ihr geöffneten Pools werden geschlossen.

Beispiel

Siehe OPEN POOL

CLOSE
POOL

Syntax

```
CLOSE POOL (poolreference);
```

Erläuterung

1) Der Pool und alle in ihm evtl. geöffneten Pools werden geschlossen.

Beispiele

```
DCL (A,B) DESCRIPTOR;
DCL X(3) DYNAMIC(B);
DEFINE A,(10) DESCRIPTOR;
DO I = 1 TO 10;
DEFINE A(I),(I) AS(X);
END;

DO J = 1 TO 10;
CONNECT A(J),B;
```

A(I) kann nun auch als B referiert werden.

```
DO I = 1 TO J;
   B(I)->X = 15.;
END;
END;
```

Alle Variablen von A(1,1)->X(1) bis A(10,10)->X(3) erhalten den Wert 15. zugewiesen. Bei Schleifenende referiert B dieselben Daten wie A(10). Dies kann rückgängig gemacht werden durch

```
RESET B; oder DESTROY B;
```

Syntax

CONNECT substructure-reference$_1$, substructure-reference$_2$

Erläuterung

1) Alle Datenbestände die zuvor über substructure-reference$_1$ referiert werden konnten, können nun auch über substructure-reference$_2$ referiert werden.

2) Falls substructure-reference$_2$ zuvor Datenbestände referierte, so wird impliziert ein DESTROY substructure-reference$_2$ ausgeführt.

Beispiel

```
DCL (A,B) DESCRIPTOR;
DCL 1 PUNKT DYNAMIC(B), 2 X(3);
DEFINE A,(4,4) DESCRIPTOR;
DEFINE B,(100) AS(PUNKT),
DO J = 1 TO 100;
DO I = 1 TO 3;
PUNKT(J).X(I) = 5.* J + I;
END;
END;
DO I = 1 TO 4;
DO I = 1 TO 4;
COPY B INTO(A(J,I));
END;
END;
DESTROY B;
---------
--------
RESET A;
RESET B;
```

COPY
dynamic substructure

Syntax

COPY substructure-reference$_1$ INTO (substructure-reference$_2$);

Erläuterung

1) Die Datenmenge, die durch die substructure-reference$_1$ referiert wird, wird unter Beibehaltung ihrer logischen Struktur kopiert. Die substructure-reference$_2$ referiert danach die erzeugte Kopie.

2) Falls substructure-reference$_2$ zuvor eine Datenmenge referiert, so wird zunächst implizit ein DESTROY substructure-reference$_2$ ausgeführt.

Beispiel

```
*SUBSYSTEM SSS;
*MODUL MMM;
     MMM : PROC(BANKS);
     DCL STOFFDATEN BANK VARIABLE INIT(BANKS(1));
     DCL PROFILE BANK VARIABLE INIT(BANKS(2));
     DCL BANKS(*)BANK;
```

DECLARE
BANK

Syntax

a) Als Level-1-Variable
DCL bank [(dimension)] BANK VARIABLE
[INIT(bankreference)] [storageclass] ;

b) Als Substructure-Variable
DCL 1 .....,
level bank [(dimension)] BANK VARIABLE,.....

c) Als Parameter
DCL bank [(dimension)]BANK;

Erläuterung

1) Als storageclass sind BASED, AUTOMATIC (Standardwert) und STATIC INTERNAL erlaubt.

2) Die Beziehung zwischen der BANK bank und dem zugehörigen PL/1-FILE muß im Main-Modul hergestellt werden. Dies kann vom Subsystem-Entwickler in PLS oder vom Subsystem-Anwender (z.B. mit Subsystem DABAL) bewerkstelligt werden.

Beispiel

```
DECLARE BILD DESCRIPTOR;

DECLARE 1 SEITE,
        2 ZEILE BIN FIXED,
        2 INHALT(N) DESCRIPTOR INTT((N)NULL( ));

RESET BILD;
DO I = 1 TO N;
   RESET INHALT(I);
END;
```

DECLARE
DESCRIPTOR

Syntax

DECLARE .....name.....DESCRIPTOR.....;

Erläuterung

1) DESCRIPTOR kann als Datentypdeklaration in gleicher Weise auftreten wie das Attribut POINTER.

2) Die Attribute EXTERNAL und CONTROLLED sind nicht erlaubt.

3) Siehe RESET basedescriptor

4) Vor der ersten Verwendung muß jeder DESCRIPTOR mit der PL/1-Builtin-Function NULL ( ) initialisiert werden. Im allgemeinen übernimmt PLR dies. Nur bei Feldern vor Deskriptoren vom BASED-Typ muß der Anwender dafür sorgen.

Beispiel

```
DCL ELEMENT DYNAMIC;

DCL 1 PUNKT DYNAMIC;
        2 X(3) BIN FLOAT,
        2 TYP BIN FIXED,
        2 NAME CHAR(32) VARYING;

DCL ZEILE(1000) CHAR(80) DYNAMIC(BUCH.SEITE);
```

DECLARE
DYNAMIC
element

Syntax

DECLARE [1] dde [(dim)] DYNAMIC (basedeskriptorreference)
sonstige Attribute ;

Erläuterung

1) Mit Ausnahme der REFER-Option (die hier nicht erlaubt ist) sind genau die Attribute erlaubt, die für PL/1-Daten der Speicherklasse BASED erlaubt sind.
Insbesondere dürfen in der Dimensionangabe dim nur Konstante benutzt werden.

2) Wenn hinter dem DYNAMIC-Attribut ein Basedescriptor referiert wird, so wird dieser bei jeder Benutzung des Dynamic Arrays als "assozierter Basedescriptor" verwendet (siehe DYNAMIC element-reference).

Beispiel

```
DECLARE MODUL DYNAMIC ENTRY;

DECLARE SUB DYNAMIC ENTRY((*,*)BIN FLOAT(21))
    RETURNS(CHAR(12));

DECLARE SUB CHAR(6) INIT('SUB3');
DECLARE MOD CHAR(6) INIT('ABCDEF');
DECLARE (SUB1 ENTRYNAME(SUB),SUB2) DYNAMIC MODULE(MOD);
  Ein Aufruf von SUB1 würde bewirken, daß die
  externe Prozedur SUB3 des Moduls ABCDEF des
  gerade aktiven Subsystems zur Ausführung
  kommt.

DCL PROG ENTRY(POOL) DYNAMIC;

DCL (X, B ENTRY) DYNAMIC;
    X ist ein Dynamic Array Element ohne assoziierten
    Basedescriptor, B ist ein Dynamic Entry.

DCL 1 STRUCTURE AUTOMATIC,
       2 SUB_PROG ENTRY DYNAMIC ENTRYNAME('XYZ'),
       2 DATEN(N,N) BIN FLOAT(21);

DCL    1 S1  BASES(Q),
       2 SUBMOD  ENTRY(PTR) MODULE(MOD);
```

DECLARE
ENTRY
DYNAMIC
Konstante

Syntax

```
DECLARE name ENTRY [(parameter-descriptor-list)]
     [RETURNS(return-attribut)]
     DYNAMIC [storageclass]
     [MODULE(m)] [ENTRYNAME(e)] ;
```

Erläuterung

1) Die Attribute können in beliebiger Reihenfolge stehen. Die Deklaration kann auch in eine STRUCTURE eingebettet sein.

2) Als return-attribut ist zulässig, was in PL/1 erlaubt ist.

3) Für storageclass ist AUTOMATIC (Standardwert) und BASED erlaubt.

4) Standardwert für den Modulname m (CHAR(6)) ist der Entryname e.

5) Standardwert für den Entrynamen e (CHAR(6)) ist 'name'.

6) Man beachte, daß Namen von Dynamic-Entry-Konstanten und Dynamic-Entry-Variablen auch ohne Qualifikation eindeutig sein müssen.

Beispiel

```
DCL VSUB ENTRY DYNAMIC VARIABLE
              INIT(MODUL);

DCL FELD_SUB(4) ENTRY(BIN FLOAT(21) (X,X))
              RETURNS(CHAR(12)) DYNAMIC
              INIT((4) SUB);

A : PROC(PROG);
DCL PROG ENTRY(CHAR(*)) DYNAMIC;
DCL PROG1 ENTRY(CHAR(*)) DYNAMIC VARIABLE;
PROG1 = PROG;
----
-----
END A;
DCL 1 STRUCT BASED(P),
 2  ENT ENTRY VARIABLE DYNAMIC,
 2  NEXT POINTER;
```

DECLARE
ENTRY
DYNAMIC
Variable

Syntax

a) DECLARE name VARIABLE ENTRY [(parameter-descriptorlist)]
[RETURNS(return-attribute)] [storageclass] [alignment]
[INITIAL(dynamic-entry-reference)] DYNAMIC;

b) DECLARE name (dimension) VARIABLE DYNAMIC
ENTRY [(parameter-descriptorlist)][RETURNS(return-attribute)]
[INITIAL(dynamic-entry-reference)] [storageclass];

c) DECLARE name [(dimension)] DYNAMIC ENTRY [VARIABLE]
[(parameter-descriptorlist)] [RETURNS(return-attribute)];

Erläuterung

1) Es wird deklariert:
in Form a eine DYNAMIC-ENTRY-Variable,
in Form b ein Feld-solcher-Variablen,
in Form c ein DYNAMIC-ENTRY-Parameter oder ein Parameterfeld.

2) Die Attribute können in beliebiger Reihenfolge stehen.
Die Deklaration kann auch in eine STRUCTURE eingebettet sein.

3) Es sind alle in PLR erlaubten storageclass-Attribute erlaubt.

4) Man beachte, daß Namen von Dynamic-Entry-Konstanten und Dynamic-Entry-Variablen auch ohne Qualifikation eindeutig sein müssen.

Beispiel

siehe Beispiel zu PROCEDURE REGENT.
Der Aufruf der dort definierten Prozedur könnte folgendermaßen geschehen:

```
XX : PROC;
DCL Y DESCRIPTOR;
DCL U DYNAMIC(Y);
DCL UP ENTRY(DESCRIPTOR,) REGENT;
DEFINE Y,(2,10) STEP(10) AS(V);
CALL UP(Y, MAX);
    MAX SEI IM SUBSYSTEMCOMMON
---------
---------
RESET Y;
END XX;
```

DECLARE
ENTRY
REGENT

Syntax

DECLARE entry ENTRY [(parameter-descriptor-list)]
REGENT [EXTERNAL] [andere-optionen];

Erläuterung

1) Wenn eine externe REGENT-Prozedur aufgerufen werden soll, so muß sie mit der Option REGENT deklariert sein.

2) Wenn die Prozedur als DYNAMIC ENTRY deklariert ist, erübrigt sich die Option REGENT.

Beispiel

```
SUBSYSTEM SSS;
MODUL MMM;
    MMM : PROC(POOLS);
    DCL POOLS(*) POOL;
    DCL (POOL1 INIT(POOLS(1)),
         POOL2 INIT(POOLS(2)) POOL;
```

DECLARE
POOL

Syntax

a) Als Variable

DCL [level] pool [(dimension)] POOL
[storageclass] [INIT(poolreference)] ;

b) Als Parameter

DCL pool [(dimension)] POOL;

Erläuterung

1) Als storageclass sind: BASED, AUTOMATIC (Standardwert) und STATIC INTERNAL erlaubt.

Beispiel

```
DCL A  DESCRIPTOR;
DCL X(3) DYNAMIC(V);
DCL 1 Y DYNAMIC, 2 Z CHAR(80), 2 F FILE;

DEFINE A,(2,2) DESCRIPTOR;
DO J = 1 TO 2;
DO I = 1 TO 2;
DEFINE A(I,J), (I,J) AS(X);
END;
END;

DEFINE A(1,2,1,4),(3) AS(Y);
DEFINE A(2),(5,5) STEP(K) PRIORITY(R >=S);
.
.
.
RESET A;
```

DEFINE

Syntax

DEFINE substructure-reference, (indexlist) [type] [STEP (step)] [priority];
d-reference = basedescriptorreference | subarrayreference
typ = DESCRIPTOR | AS (dynamicelementname)
priority = HIGH | LOW | PRIORITY(p)

Erläuterung

1) Falls die betreffende substructure-reference bereits eine Datenmenge referierte, so wird diese alte Referenz gelöscht (Implizites DESTROY). Daten, auf die dadurch nicht mehr zugegriffen werden kann, werden vernichtet.

2) Die Indexliste definiert die Maximalzahl und die Schachteltiefe der neu definierten Datenteilbäume.

3) Wenn typ nicht angegeben ist, wird DESCRIPTOR angenommen.

4) Bei der Option DESCRIPTOR enthält das unterste Niveau der Datenteilbäume Deskriptoren, bei der Option AS( ) dagegen Dynamic Array Elemente mit einer Struktur, wie sie durch die zugehörige DYNAMIC-Deklaration vereinbart wurde.

5) In der STEP-Option kann ein Ausdruck angegeben werden, der einen Wert von 0 oder > 0 ergeben muß. Standard ist 0. Die Stepoption bezieht sich stets auf alle Indizes der Indexliste.

6) Fehlt die Prioritätsangabe so wird LOW angenommen. Der Prioritätsausdruck p muß entweder '1'B (d.h. HIGH) oder '0'B (d.h. LOW) ergeben.

Beispiel

```
DCL (A,B) DESCRIPTOR;
DCL X DYNAMIC;
DEFINE A,(10) DESCRIPTOR;
DO J = 1 TO 10;
DEFINE A(J),(2*J) STEP(J) PRIORITY(J>5) AS(X);
END;
DEFINE B, (2) DESCRIPTOR;
DO K = 1 TO 2;
DEFINE B(K), LIKE A;
END;
Die letzten 3 Anweisungen bewirken
dasselbe wie
DO K = 1 TO 2;
  DEFINE B(K), (10) DESCRIPTOR;
  DO I = 1 TO 10;
  DEFINE B(K,J), (2*J) STEP(J) PRIORITY(J>5) AS(X);
  END;
END;
.
.
RESET A;
RESET B;
```

DEFINE
LIKE

Syntax

DEFINE substructure-reference$_1$, LIKE substructure-reference$_2$.

Erläuterung

1) Die Struktur des Teilbaumes, der durch substructure-reference$_2$ angegeben ist, wird auf substructure-reference$_1$ übertragen. Die Wirkung ist dieselbe, wie wenn alle DEFINE Anweisungen, die den von substructure-reference$_2$ referierten Datenbestand definieren, in gleicher Reihenfolge für substructure-reference$_1$ ausgeführt würden.

Beispiel

```
DCL A DESCRIPTOR;
DCL X(3) DYNAMIC(A);
DEFINE A,(10) DESCRIPTOR;
DO I = 1 TO 10;
DEFINE A(I),(I,2) AS(X);
END;

DO I = 1 TO 3;
X(4,2,1) (I) = I
END;

DESTROY A(4,2);
```

Die oben implizit enthaltene Anweisung DEFINE A(4,2), (2) AS(X); wird rückgängig gemacht. Der Wert X(4,2,1)(1) bis X(4,2,1)(3) existieren nicht mehr.

```
DESTROY A(5);
```

Die Anweisungen DO I = 1 TO 5; DEFINE A(5,I), (2) AS(X); END; sowie DEFINE A(5),(5) DESCRIPTOR; werden rückgängig gemacht.

```
DESTROY A;
```

Die Anweisung DEFINE A,(10) DESCRIPTOR wird rückgängig gemacht.

```
RESET A;
```

Syntax

DESTROY substructure-reference;

Erläuterung

1) Die DESTROY-Anweisung macht die zugehörige (evtl. implizite) DEFINE-Anweisung rückgängig.

2) Daten, die in den durch substructure-reference referierten Datenbeständen enthalten waren und nun nicht mehr referiert werden können, werden vernichtet.

3) Datenbestände, die aufgrund von CONNECT-Anweisungen über mehrere substructure-referencen referiert werden können, werden nicht vernichtet; lediglich die Zugriffsmöglichkeit über die im DESTROY angegebene Referenz wird beseitigt.

Beispiel

```
DCL 1 A DYNAMIC(TOP),
    2 B(20,40),
    2 C POOL;

DCL (TOP(3)) DESCRIPTOR;
DCL U DYNAMIC BIN FIXED;
DCL 1 V(20) DYNAMIC, 2 BIT BIT, 2 F FILE;
```

Gültige Dynamic-Element-Referenzen

a) TOP(1) (I,4,3) -> A.C
   TOP(2) (4,2,I) -> A
   TOP(TOP (2) -> U) (L,2,3) -> V

b) B(10,20) (20,10)

ungültige Dynamic-Element-Referenzen

```
    U(3,4,7)      (kein assozierter Basedescriptor)
    DCL A INIT(TOP (2,3)->U);     (Deklaration)
    TOP(2) -> V(3).F    (ungenügende Qualifikation)
```

dynamic
element
reference

Syntax

a) Voll qualifizierte Form:
basedescriptorreference (dds-indexlist) → element
[(indexlist)] [.substructure]

b) Implizit qualifizierte Form:
ba) element (dds-indexlist) [.element-substructure]
bb) element (dds-indexlist$_1$) (indexlist) [.element-substructure]

Erläuterung

1) Zu a) Die nach dem Qualifikator stehende Bezeichnung ist formal identisch mit der Referenz einer PL/1-Struktur, eines PL/1-Feldes oder einer PL/1-Variablen, die jedoch als BASED statt als DYNAMIC vereinbart wurde.

Zu b) Diese Form ist nur gültig, wenn bei der DYNAMIC Deklaration ein assoziierter Basedeskriptor angegeben wurde.

2) Die Verwendung von Dynamic-Element-Referenzen ist nicht erlaubt

in der ATTRIBUTE-Anweisung
in DO-Anweisungen
in Formaten
in Deklarationen
im RETURN (...)
in PUT-DATA-Listen
in den Funktionen ADDR und UNSPEC
im logischen Ausdruck einer IF-Anweisung

3) Im Gegensatz zu einer Substructure-Referenz darf hier die dds-Indexliste nicht leer sein. Die Anzahl der angegebenen Indices muß ausreichen, um genau ein DDE-Element zu erreichen.

Beispiel

```
CALL VSUB;


TEXT = FELD_SUB(I) (DATEN);

PUT EDIT(FELD_SUB(3) (DATEN)) (SKIP,A(12));

DCL U_P ENTRY(CHAR(20)) EXTERNAL;
CALL U_P(SUB);

DCL A ENTRY(ENTRY DYNAMIC) DYNAMIC;
DCL B ENTRY DYNAMIC;
CALL A(B);
```

DYNAMIC
ENTRY
Referenz

Syntax

Eine ausreichend qualifizierte und indizierte Dynamic-Entry-Bezeichnung

Erläuterung

1) Die Anwendung von Dynamic-Entry-Funktionen in Ausdrücken und Argumenten geschieht in gleicher Weise wie die Anwendung von PL/1-Funktionsprozeduren.

2) Dynamic-Entry-Subroutinen werden in gleicher Weise durch eine CALL-Anweisung aufgerufen, wie PL/1-Subroutinen. Anstelle des CALL kann auch LINK stehen.

Beispiel

```
DCL A DESCRIPTOR;
DCL 1 PUNKT BASED(P), 2 FARBE BIN FIXED(15), 2 X(3);
DCL 1 PUN DYNAMIC(A), 2 N, 2 Y(3);
DCL (P) POINTER;
DCL 1 FELD(32000) BASED(P) LIKE PUNKT;
DEFINE A,(100,100,100) AS(PUN);
DO I = 1 TO 100;
DO I = 1 TO 100;
  FIX A(I,J) SET(P) ON(1);
  DO K = 1 TO 100;
     FELD(K).FARBE =.....
     FELD(K).X = .......;
     Die Felder A(I,J) und das Feld FELD überlagern
     sich. Der Zugriff ist sehr effektiv.
  END;
  LOOSE A(I,J)
END;
END;
   FIX A(I,J) SET(P) ON(50);
   PUT LIST(PUNKT);
   Das Feld A(I,J,50) und die Struktur
   PUNKT überlagern sich.

   LOOSE A(I,J);
RESET A;
```

FIX
SET
pointer

Syntax

FIX substructure-reference SET(pointerreference) [ON(index)]
[READ] ;

Erläuterung

1) Der Datenvektor, der über die Referenz referiert wird, wird im Arbeitsspeicher bis zum zugehörigen LOOSE (oder DESTROY) festgelegt. Die pointerreference erhält als Wert die Adresse des index-ten Datenelementes dieses Vektors.

2) Die Subarrayreferenz darf nur Dynamic Elemente enthalten, also keine weiteren Deskriptoren.

3) Wenn die ON-Option fehlt, wird ON(1) angenommen.

4) Die READ-Option erhöht die Effektivität, wenn die betreffenden Daten nicht verändert werden. Werden sie trotz Angabe der READ-Option verändert, so führt dies zu unvorhergesehenen Fehlern.

5) Siehe LOOSE;

Beispiel

PLR-Programm:

```
*SUBSYSTEM SSS
*MODULE MMM
     MMM : PROC;
     INITIATE BANK(BANK1) FREELIST(256) SPACE(128);
     END MMM;
```

Subsystem-Common

```
ENTER PLS;
SUBSYSTEM 'SSS';
   DCL 1;
     2 ....
     2 BANK1 BANK;
     ..........
END DATA;
CLAUSE INIT;
LINK MMM;
END CLAUSE;
END PLS;
```

Job-Control-Language

```
    //BANK1 DD DSN = DATEI.PRIVAT,DISP = SHR
```

Bei Eintritt in das Subsystem SSS mittels ENTER SSS; wird auf dem Dataset DATEI.PRIVAT eine REGENT-Datenbank initialisiert.

INITIATE
BANK

Syntax

```
INITIATE BANK (bankreference) [bankattributes] [password-attributes];
bankattributes  =  FREELIST (freelist) |
                   DELETELIST (deletelist) | NODELETELIST |
                   LISTPAGE (listpage) |
                   SPACE (space) |
                   TITLE(bankname)
password-attributes =  siehe INITIATE BANK PASS
```

Erläuterung

1) Eine REGENT-Datenbank wird auf dem Datenträger, der durch bankreference referiert wird, initialisiert. Die Beziehung zwischen Datenträger und bankreference wird auf folgende Weise hergestellt:

   bankreference .....bank-Konstante
   bank-Konstante PL/1-file
   PL/1-file.....DDname
   DDname.....Dataset im Sinne des Betriebssystems

2) Zur Erläuterung der bankattributes siehe die Beschreibung in Kap. 4 des Handbuchs. Standardwerte sind:

freelist	64
deletelist	16
listpage	16
space	16

Beispiel

```
INITIATE BANK(BANK1) PASSWORD(ALL = 'MEIER');
```

Passwort MEIER erlaubt alle Operationen

```
INITIATE BANK(BANK1);
```

Es wird keine Passwortprüfung vorgenommen

```
INITIATE BANK(BANK1) PASSWORD(ALL = 'MEIER',CP,C = 'MUELLER');
```

Zum Ändern von Passworten der Datenbank und aller darin enthaltenen Pools dient das Passwort MUELLER, alle übrigen Operationen verlangen das Passwort MEIER.

```
INITIATE BANK(BANK1) PASSWORD(CP,C = 'MUELLER',
                              W,WP,U,UP = 'MEIER');
```

Zum Ändern von Passworten in der Bank ist das Passwort MUELLER erforderlich. Zum Schreiben und Ändern von Daten oder Eröffnen neuer Pools wird das Passwort MEIER benötigt. Nur zum Lesen kann die Bank nicht geöffnet werden.

INITIATE
BANK
PASSWORD

Syntax

INITIATE BANK(bankreference) [bankattributes] [password-attributes];
bankattributes siehe INITIATE BANK
password-attributes = PASSWORD(list) | NOPASS | NOPASSWORD

list::= passtype [,passtype] *= expression
[,passtype [,passtype] * = expression] *

passtype = ALL | ALL PRIV | ALL NOPRIV |
R | READ | W | WRITE | U | UPDATE |
C | CHANGE PASSWORD |
RP | READ PRIV | WP | WRITE PRIV |
UP | UPDATE PRIV |
CP | CHANGE PASSWORD PRIV

Erläuterung

1) Standard ist NOPASS oder NOPASSWORD

2) Zur Erläuterung der Passworttypen siehe die Beschreibung in Kap. 4 des Handbuches

3) Der Passwortausdruck expression muß in eine Zeichenkette der Länge 8 konvertierbar sein.

4) Die Passworttypen müssen genau in der hier festgelegten Form angegeben werden. Z.B. ist PRIV READ nicht erlaubt.

Beispiel

```
DCL  BANK1 BANK;
DCL DATEI(4) POOL;
INITIATE BANK(BANK1) PASSWORDS(ALL='1', U,UP='2',W='3',WP='4');
OPEN BANK(BANK1) UPDATE PASSWORD('3');
INITIATE POOL(DATEI(1)) IN(BANK1) TITLE('D1')
     CATALOG
     PASSWORD(ALL ='X',U,W ='Y');
OPEN POOL(DATEI(1) TITLE('D1') IN(BANK1);
INITIATE POOL(DATEI(2)) IN(DATEI(1)) TITLE('O2')
     SPACE(20) CATALOG;
OPEN POOL(DATEI(2)) IN(BANK(2);
DO I = 3,4;
INITIATE POOL(DATEI(I)) IN(DATEI(2)) SPACE(10000) TITLE('O2')
     DATA;
END;

CLOSE POOL(DATEI(1));
     DATEI(3), DATEI(4), DATEI(2) werden implizit geschlossen
CLOSE BANK(BANK1);
```

INITIATE
POOL

Syntax

INITIATE POOL(poolreference) TITLE(poolname) IN(bank-or-poolreference)
[SPACE(space)] [PASSWORD(list)] [type] ;

type = CATALOG | DATA

Erläuterung

1) Es wird in bank-or-poolreference ein neuer Pool namens poolname eingerichtet. Ein Pool gleichen Namens darf nicht schon vorhanden sein

2) space gibt bei CATALOG die maximale Zahl der Einträge, bei DATA den Speicherbereich in Bytes an. Standardwert ist 16.

3) Für die Passwortliste list gelten die gleichen Regeln wie für INITIATE BANK

4) Standardwert für type ist DATA

Beispiel

```
DCL (A,B)DYNAMIC ENTRY;
LOAD A;
DO J = 1 TO 1000;
LOAD B;
CALL A;
CALL B;
END;
```

Anmerkung:

Die Anweisung LOAD B; hat in der Schleife nur beim ersten Durchlaufen eine Wirkung

LOAD
dynamic
entry

Syntax

LOAD dynamic-entry-reference;

Erläuterung

1) Der durch die dynamic-entry-reference spezifizierte dynamic entry wird in den Arbeitsspeicher geladen. Weitere Aufrufe dieses dynamic-entry sind genauso effektiv wie Aufrufe normaler PL/1-Prozeduren.

2) Die Wirkung der LOAD-Anweisung wird durch eine RELEASE-Anweisung rückgängig gemacht.

3) Mehrfaches LOAD auf denselben dynamic entry hat keine andere Wirkung als ein einfaches LOAD.

Beispiel

Siehe Beispiele zu FIX SET pointer.

```
DCL A DESCRIPTOR;
DCL X(32000) BASED(P);
DCL P1 POINTER;
DCL Y DYNAMIC(A);
.....
.....
FIX A(5,3,2,7) SET(P) ON(1);
FIX A(5,3,2,7) SET(P1) ON(2);
DO I = 1 TO 10;
X(I) = P1-> X(I);
END;
```

Dies entspricht folgenden Operationen:

```
Y(5,3,2,7,1) = Y(5,3,2,7,2);  bis
Y(5,3,2,7,10) = Y(5,3,2,7,11);
LOOSE A(5,3,2,7);
```

P und P1 sind jetzt undefiniert. Z.B. wäre X(3) = P1->X(3) jetzt ein Fehler.

```
RESET A;
```

LOOSE

Syntax

LOOSE substructure-reference;

Erläuterung

1) LOOSE macht die Wirkung von FIX SET rückgängig

2) Siehe FIX SET pointer

3) Auch wenn für einen Dynamic-Array-Element-Vektor mehrere FIX Operationen ausgeführt wurden, so genügt eine einzige LOOSE-Operation, um diesen Vektor wieder für Reorganisationen der Dynamic Array-Verwaltung verfügbar zu machen. Alle Pointer, die durch FIX auf diesen Datenvektor gesetzt worden waren, sind nach dem LOOSE undefiniert. Ihre Verwendung nach LOOSE kann unvorhersehbare Fehler hervorrufen.

Beispiele

Siehe Kap.6.8, Seite 6-41 bis 6-51

```
MESSAGE
MESSAGE ACTIVE
MESSAGE COUNT
MESSAGE DUMP
MESSAGE INACTIVE
MESSAGE RESET
```

Erläuterung

1) Die Syntax und Wirkung dieser Befehle ist in PLR gleich wie in allen Subsystemsprachen und in PLS. Siehe Kap.6.8 (Seite 6-41 bis 6-51).

Beispiel

```
OPEN BANK(BANK1);
```

BANK1 wird für INPUT eröffnet mit 2 Puffern und Standardpasswort.

```
CLOSE BANK(BANK1);
OPEN BANK(BANK1) UPDATE BUFNO(4) PASSWORD('MUELLER');
CLOSE BANK(BANK1);
```

OPEN
BANK

Syntax

OPEN BANK(bankreference) [PASSWORD(expression$_1$)][BUFNO(pufferzahl)]
[mode] ;

Erläuterung

1) mode kann sein INPUT OUTPUT UPDATE. Standard ist INPUT

2) Falls die Datenbank mit Passwortoption initialisiert war, muß der Passwortausdruck expression$_1$ mit den eingetragenen Passworten identisch sein.

mode	erforderliches Passwort
INPUT	RP | R | WP | W | UP | U
OUTPUT	WP | W | UP | U
UPDATE	UP | U

3) Standardpasswort besteht aus 8 Blanks (' ').

4) Standardwert für die Pufferzahl ist 2.

Beispiel

```
DCL BANK BANK;
DCL DATEI(4) POOL;

OPEN BANK(BANK) OUTPUT PASSWORD('X');
X sei das Passwort für WRITE-Operationen
OPEN POOL(DATEI(1)) IN(BANK) TITLE('ABC') INPUT DELETE CATALOG;
Fehler, da BANK nicht mit UPDATE eröffnet

OPEN POOL(DATEI(2)) IN(BANK) TITLE('123') CATALOG OUTPUT PASSWORD(
U sei Passwort für UPDATE PRIV
Alle folgenden OPEN innerhalb von DATEI(2)
erfordern kein Passwort mehr.

OPEN POOL(DATEI(3)) IN(DATEI(2)) TITLE('X') INPUT DATA;
OPEN POOL(DATEI(4)) IN(DATEI(3)) TITLE('X');

CLOSE POOL(DATEI(2));
DATEI(3) und DATEI(4) werden implizit geschlossen
CLOSE BANK(BANK);
```

OPEN
POOL

Syntax

OPEN POOL(poolreference) TITLE(poolname) IN(bank-or-poolreference)
[opendisp][closedisp][mode] [PASSWORD(password)] [type];

opendisp::= OLD | MOD
closedisp::= KEEP | DELETE | RELEASE
mode::= INPUT | OUTPUT | UPDATE

Erläuterung

1) Es wird ein in bank-or-poolreference bestehender Pool namens poolname eröffnet.

2) Standardwert für opendisp ist OLD.

3) MOD ist nur im Zusammenhang mit DATA erlaubt.

4) Standardwert für closedisp ist KEEP

5) DELETE erfordert, daß bank-or-poolreference mit UPDATE eröffnet ist.

6) OUTPUT (bzw. UPDATE) erfordert, daß bank-or-poolreference mit OUTPUT oder UPDATE (bzw. UPDATE) eröffnet ist.

7) Standardwert für mode ist INPUT.

8) Standardwert für type ist DATA.

Beispiel

```
DCL 1 STRUCTURE(5),
      2 B BANK,
      2 D DESCRIPTOR,
      2 ER ENTRY REGENT,
      2 ED ENTRY DYNAMIC VARIABLE,
      2 P POOL;

DCL SUB1 ENTRY( ) BANK,ENTRY,ENTRY DYNAMIC);
DCL SUB2 ENTRY(1,2 (BANK,DESCRIPTOR,ENTRY,ENTRY DYNAMIC,POOL))
         DYNAMIC;
CALL SUB1(B,ER(2),ED(4));
CALL SUB2(STRUCTURE);
```

parameter
descriptor list

Syntax

DCL.....ENTRY(parameter descriptor [,parameter descriptor]*)..

Erläuterung

1) Als parameter descriptor sind alle Bezeichnungen zugelassen, die in PL/1 zugelassen sind

2) zusätzlich sind in der Deklaration von
   ENTRY(.....) DYNAMIC und
   ENTRY(.....) REGENT
   zugelassen:
   BANK
   DESCRIPTOR
   POOL
   ENTRY DYNAMIC

3) Es wird empfohlen, die parameter descriptor list in allen ENTRY-Deklaration vollständig anzugeben.

Beispiel

```
A:PROC(PP);
   DCL (P,PP(*)) POINTER;
   DCL 1 DATEI BASED(P),
       2 TITEL CHAR(16),
       2 POOLX POOL;

   DCL POOL POOL;
   DCL B ENTRY(POOL) REGENT;
   DCL POOL1 POOL INIT(PP(4) -> POOLX);
   P = PP(1);
   OPEN POOL(PP(2) -> DATEI.POOLX) IN(POOLX) CATALOG;
   OPEN POOL(PP(3) -> POOLX) IN(PP(2) -> POOLX) DATA;
   CALL B(POOL1);
   POOL = PP(4) -> POOLX;
   CALL B(POOL);
END A;
```

Syntax

Eine poolreference sieht aus wie eine POINTER-Referenz.

Erläuterung

1) Eine poolreference kann nur stehen in der INIT-Option von DECLARE POOL, als aktueller Parameter und in den Anweisungen

OPEN POOL
CLOSE POOL
RETRIEVE POOL
ATTRIBUTE
STORE POOL
SET POOL

Beispiel

```
DCL X(3) DYNAMIC(A);
DCL A DESCRIPTOR;
DEFINE A,(10)DESCRIPTOR;
DO I = 1 TO 10;
DEFINE A(I),(I) AS(X);
REDEFINE A(10), STEP(10);
     A(10) erhält STEP(10) statt STEP(0)
REDEFINE A(10), HIGH;
     A(10) erhält Priorität HIGH statt LOW
X(10,20).X = 15.;
     X(10) wird auf A(10,20) erweitert
     Die drei Variablen
       A(10,20) -> X(1)
       A(10,20) -> X(2)
       A(10,20) -> X(3)
     erhalten den Wert 15. zugewiesen
     A wird UNRELEASED
     A(10) wird UNRELEASED
REDEFINE A(10), REDUCE;
              Die Daten, die über A(10,11) bis A(10,20)
              referiert werden konnten, entfallen.
REDEFINE A(10), RELEASED;
              A(10) ist wieder RELEASED
              RESET A;
```

REDEFINE

Syntax

REDEFINE substructure-reference, option [option] *;

option:: = REDUCE [(number)] | STEP(step) | RELEASED | priority
priority:: = HIGH | LOW | PRIORITY(p)

Erläuterungen

1) REDUCE ohne number-Angabe reduziert die Zahl der durch Basedeskriptorreferenz oder Subarrayreferenz referierten Teilbäume durch mehrfache Reduktion um die gültige Schrittweite step auf den kleinsten positiven Wert.

2) REDUCE mit number-Angabe reduziert die Länge auf den Wert number.

3) Die STEP-Option setzt einen neuen Wert für step.

4) Die RELEASED-Option setzt den Status auf RELEASED.

5) Die priority-Option setzt eine neue Priorität fest.

Beispiel

```
DCL (A,B) DYNAMIC ENTRY;

LOAD A;
DO WHILE(.....);
CALL A;
END;
RELEASE A;
LOAD B;
DO J = 1 TO 500;
CALL B;
END;
RELEASE B;
```

RELEASE
dynamic
entry

Syntax

RELEASE dynamic-entry-reference;

Erläuterung

1) siehe LOAD dynamic entry.

2) Mehrfaches RELEASE ist ohne Wirkung.

3) Wenn vergessen wird, ein dynamic entry nach einem LOAD wieder zu RELEASEn, so bleibt der betreffende Modul bis zum Ende des REGENT-Laufs im Arbeitsspeicher.

Beispiel

```
REORG HIGH RELEASED
                Daten mit Prioritätsstatus HIGH UNRELEASED
                sollen nicht reorganisiert werden.
REORG DA SIZE(40000);
                Der Pool wird auf 40000 Bytes gesetzt.
REORG HIGH RELEASED SIZE(10000);
REORG LOW RELEASED;
```

Syntax

REORG [sizeoption] [leveloption];

```
    sizeoption::= SIZE(poolsize)
   leveloption::= release priority
       release::= RELEASED | UNRELEASED
      priority::= HIGH | LOW | PRIORITY(p)
```

Erläuterung

1) sizeoption und leveloption können auch einzeln oder in umgekehrter Reihenfolge angegeben werden. Dasselbe gilt für release und priority (Standard: UNRELEASED, LOW).

2) poolsize gibt die neu festzulegende Poolgröße der dynamischen Datenstrukturen in Bytes an. Ist diese kleiner als die aktuelle Größe, so wird sofort reorganisiert.

3) Über die Leveloption kann angegeben werden, bis zu welchem Prioritätsstatus der dynamischen Datenfelder bei dieser und folgenden Reorganisationen reorganisiert werden soll.

LOW	RELEASED
LOW	UNRELEASED
HIGH	RELEASED
HIGH	UNRELEASED

Beispiel

```
DCL POOL POOL;
OPEN POOL (POOL) NEW UPDATE.......;
DCL N BIN FIXED(15);

DO I = 1 TO 4;     /* Spaltenindex  */
DO J = 1 TO 4;     /* Zeilenindex   */
   N = 10 *I + J;
   STORE POOL(POOL) FROM(N);
END; END;
/* Speichert eine 4 mal 4-Matrix, deren Elemente Zahlenfolge N
   als zweistellige Integerkonstante enthalten, zeilenweise ab */
SET POOL (POOL) NEXT(0);
DO I = 1 TO 16;
   RETRIEVE POOL(POOL) TO(N);
   PUT EDIT(N) (SKIP, 16 F(3));
END;
Die Zahlenfolge wird gelesen
/* 11 12 13 14 21  22 23 24 31 32 33 34 41 42 43 44  */
/* Lesen dieser Matrix in M */
DCL M(4,4) BIN FIXED(15);
RETRIEVE POOL(POOL) TO(M) FROM(0);
IJ   = MATRIX(2,3)  /* liefert IJ  : 23 */

MATRIX : PROC(I,J) RETURNS(BIN FIXED(15));
DCL M BIN FIXED(15);
RETRIEVE POOL(POOL) TO(N) FROM(2*(I-1)* 4 + (J-1)));
RETURN(N);
```

RETRIEVE
POOL
data

Syntax

RETRIEVE POOL(poolreference) TO(data) [SET(key)] ;

Erläuterung

1) Aus dem Pool werden beginnend an der aktuellen Position des Positionszeigers Daten gelesen und in dem durch data referierten Datenfeld (Datenelement, Feld, Struktur oder Datenfeld einer dynamischen Datenstruktur) gespeichert.

2) Der Positionszeiger rückt um die Länge der übertragenen Daten vor.

3) Mit der SET-Klausel wird der Wert des Positionszeiger zu Beginn des gerade gelesenen Records zurückgeliefert.

Beispiel

```
.......
DCL A DESCRIPTOR;
DCL B DYNAMIC A;
DEFINE A,(10) AS(B) STEP(1);
DCL X;
DCL KEY BIN FIXED(31);
J = 0;
RETRIEVE POOL(P) FROM(KEY) TO(X) NEXT(KEY);
DO WHILE(X<0);
J = J + 1;
B(J) = X;
RETRIEVE POOL(P) FROM(KEY) TO(X) NEXT(KEY);
END;
```

Erläuterung

Es wird (ohne Benutzung oder Veränderung des Positionszeigers) aus Pool P eine Folge von X-Werten der Reihe nach gelesen und in der DDS B gespeichert.

RETRIEVE
POOL
data FROM

Syntax

RETRIEVE POOL(poolreference) TO(data)
  FROM(key_1) [NEXT(key_2)] ;
  key_1:: = pooloffset | pooloffset, indexlist | indexlist

Erläuterung

1) Aus dem Pool werden beginnend an der Position key_1 Daten gelesen und in dem durch data referierten Datenfeld (Datenelement Feld, Struktur oder Datenfeld einer dynamischen Datenstruktur) gespeichert.

2) Der Positionszeiger im Pool wird durch die Anweisung nicht verändert.

3) Mit der NEXT-Klausel wird die unmittelbar auf die übertragenen Daten folgende Position zurückgeliefert.

Beispiel

```
siehe STORE POOL dynamic array
OPEN POOL(DA_POOL)....INPUT.....;
RETRIEVE POOL(DA_POOL) TO(TOP) FROM(KEY2);
DO I = 1 TO 3;
PUT SKIP LIST(BLATT(I));
END;
ergibt 1 2 3
RETRIEVE POOL(DA_POOL) TO(N) FROM(KEY1,1,1,3);
ergibt N : 33
RETRIEVE POOL(DA_POOL) TO(TOP) FROM(KEY1,1,1);
N = BLATT(3);
ergibt N : 33
```

RETRIEVE
POOL
dynamic data structure

Syntax

RETRIEVE POOL(poolreference) TO (substructure-reference)
[FROM(key_1)];

key_1:: = key | key, indexlist

Erläuterung

1) Die im Pool poolreference gespeicherte dynamische Datenstruktur wird in die substructure-reference kopiert. Sofern substrucutre-reference zuvor eine Datenmenge referierte, wird ein DESTROY substructure-reference implizit ausgeführt.

2) Es können Teilmengen des unter key gespeicherten dynamischen Datenbaumes dadurch referiert werden, daß in analoger Weise wie bei einer Subarrayreferenz eine Indexliste angegeben wird.

3) Die Angabe einer indlist ist nur zulässig, wenn die dynamische Datenstruktur mit STORE....DIRECT in den Pool gespeichert worden war.

Beispiel

```
BEGIN;
DCL A DESCRIPTOR;
DEFINE A,.....;
......
RESET A;
END;

A : PROC;
DCL X DESCRIPTOR;
DEFINE X,......;
......
RESET X;
RETURN;
......
END A;

BEGIN;
DCL Y DESCRIPTOR;
DEFINE Y,.....;
IF...THEN DO; RESET Y; GO TO OUT; END;
......
END;
OUT:......
```

RESET
basedescriptor

Syntax

RESET basedescriptor-reference[()];

Erläuterung

1) Wenn in einem Block ein Basedescriptor deklariert wurde, mit NULL initialisiert und vor Verlassen des Blocks in einer DEFINE-Anweisung benutzt wurde, so muß vor Verlassen des Blocks ein RESET ausgeführt werden. Dies gilt auch, wenn das DEFINE z.B. in einem gerufenen Unterprogramm ausgeführt wurde.

Beispiel

```
DCL POOL POOL;
OPEN BANK(POOL) IN(....) OLD DATA;

DCL X;
DO I = 1 TO 100;
SET POOL(POOL) NEXT((I-1) * 4 *10 + 1);
Der Zeiger wird auf das 1.,41.,81. etc. Byte gesetzt.
RETRIEVE POOL(POOL) TO(X);
Es wird die 1.,10.,20 etc. Zahl aus der Datei POOL nach X gelesen,
sofern die Zahlen alle die Länge von 4 Bytes haben.
......
........
END;

DCL CARD_POOL POOL;
OPEN POOL(CARD_POOL) IN.......;
DCL CARD CHAR(80);
SET POOL(CARD_POOL) NEXT(9 * 80);
DO I = 1 TO 20;
   RETRIEVE POOL(CARD_POOL) TO(CARD);
   PUT EDIT(CARD) (SKIP,A);
END;
Es werden die Karten 10 bis 29 ausgegeben.
```

SET
POOL
NEXT

Syntax

```
SET POOL(poolreference) NEXT(expression);
```

Erläuterung

1) Im eröffneten Pool poolreference vom Typ DATA erfolgt der nächste sequentielle Zugriff beim Pooloffset expression. Direkter Zugriff ist davon nicht berührt.

2) expression wird als BIN FIXED(31,0) ausgewertet.

Beispiel

```
DCL POOL POOL;
OPEN POOL (POOL).....OUTPUT NEW;
DCL CARD CHAR(80) INIT('KARTE1');
DCL OFF BIN FIXED(31);
STORE POOL(POOL) FROM(CARD) SET(OFFS);
    /* OFFS :  0  */
STORE POOL(POOL) FROM(CARD) SET(OFFS);
    /* OFFS : 80  */
STORE POOL(POOL) FROM(CARD) TO(160);
DCL NAK BIN FIXED(31);
ATTRIBUTE POOL ACT(NAK);
/*  NAK  :  240  */
STORE POOL(POOL) FROM(CARD) SET(OFFS);
    /*  OFFS : 160  */
CARD = 'KARTE2';
STORE POOL(POOL) FROM(CARD) TO(80);
    /*  Überschreiben der 2. Karte   */
```

STORE
POOL
data

Syntax

STORE POOL(poolreference) FROM(data) [SET(key)] ;

Erläuterung

1) Die durch data bezeichneten Daten (Datenelement, Feld, Struktur oder Datenfeld einer dynamischen Datenstruktur) werden in den Pool geschrieben. Der Pool muß mit DATA OUTPUT oder DATA UPDATE eröffnet sein.

2) Der Positionszeiger im Pool wird um die Recordlänge der übertragenen Daten hochgezählt.

3) Mit der SET-Klausel wird die Position des gerade abgespeicherten Records zurückgeliefert.

Beispiel

```
DCL 1 STRUCTURE, 2......;
DCL KEY(N) BIN FIXED(31);
DO I = 1 TO N;
/* BERECHNE STRUCTURE */
STORE POOL(P) FROM(STRUCTURE) SET(KEY(I));
END;
------
------
RETRIEVE POOL(P) TO(STRUCTURE) FROM(KEY(K));
/* DIE K-TE STRUCTURE WIRD GELESEN */
-------
STORE POOL(P) FROM(STRUCTURE) TO(KEY(K));
/* DIE K-TE STRUCTURE WIRD ÜBERSPEICHERT */
```

Beispiel

```
DCL L BIN FIXED(31);
DCL A DESCRIPTOR;
DCL B DYNAMIC(A);
--------
STORE POOL(P) DIRECT FROM(A) SET(L);
STORE POOL(P) TO(L,5,3,2) FROM(B(5,3,3));
/* DER GESPEICHERTE WERT VON B(5,3,2) WIRD ÜBERSPEICHERT */
```

STORE
POOL
TO data

Syntax

STORE POOL(poolreference) FROM(data) TO(key_1) [NEXT(key_2)];
key_1::= pooloffset |
pooloffset, indexlist

Erläuterung

1) Die durch data bezeichneten Daten (Datenelement, Feld, Struktur oder Datenelement einer dynamischen Datenstruktur) werden in den Pool geschrieben. Der Pool muß mit DATA OUTPUT oder DATA UPDATE eröffnet sein.

2) Wenn die Option (pooloffset,indexlist) in der TO-Klausel verwendet wird, wird auf ein Datenelement einer im Pool bereits gespeicherten dynamischen Datenstruktur zugegriffen und ihm der Wert von data zugewiesen. Diese Form führt zu Fehlern, wenn bei pooloffset kein Descriptor einer dynamischen Datenstruktur gespeichert wurde. Die NEXT-Option ist dabei nicht erlaubt.

3) Der Positionszeiger im Pool wird bei dieser Anweisung nicht verändert.

4) Mit der NEXT-Klausel wird die auf die übertragenen Daten folgende Position im Pool zurückgeliefert.

Beispiel

```
DCL TOP DESCRIPTOR;
DCL BLATT BIN FIXED(15) DYNAMIC(TOP);
DEFINE TOP,(5,4,3) DESCRIPTOR;
N=1;
DO I = 1 TO 5;
DO 7 = 1 TO 4;
DO K = 1 TO 3;
BLATT(I,J,K)  = N;
N = N + 1;
END; END; END;
DCL (KEY1,KEY) BIN FIXED(31);
DCL DA_POOL POOL;
OPEN POOL(DA_POOL).....OUTPUT KEEP DATA;
STORE POOL(DA_POOL) FROM(TOP) DIRECT SET(KEY1);
STORE POOL(DA_POOL) FROM(TOP(1,1)) DIRECT SET(KEY2);
CLOSE POOL(DA_POOL);
N = 33;
STORE POOL(DA_POOL) FROM(N) TO(KEY1,1,1,3).
RESET TOP;
/* siehe RETRIEVE POOL dynamic array */
```

STORE
POOL
dynamic data structure

Syntax

STORE POOL(poolreference) FROM(substructure-reference)
[DIRECT] [SET(key)] ;

Erläuterung

1) Die durch substructure-reference referierte dynamische Datenstruktur oder Teildatenstruktur wird in den Pool geschrieben. Der Pool muß mit DATA UPDATE oder DATA OUTPUT eröffnet sein.

2) Die Option DIRECT ermöglicht einen späteren indexsequentiellen Zugriff auf Subarrays (siehe RETRIEVE FROM bzw. STORE TO, das bedeutet aber geringere Effektivität beim STORE bei gleichen Speicherplatzanforderungen.

3) Fehlt die TO-Klausel, so wird sequentiell geschrieben, d.h. beginnend beim aktuellen Positionszeiger. Der Positionszeiger wird um die Länge der zu speichernden Daten hochgezählt.

4) Mit der SET-Klausel wird die Anfangsposition der Dynamischen Datenstruktur zurückgeliefert. Dieser Pooloffset kann z.B. bei einem späteren RETRIEVE benutzt werden, die Dynamische Datenstruktur im Pool zu lokalisieren.

Beispiel

```
DCL POOL POOL;
OPEN POOL (POOL).....OUTPUT NEW;
DCL CARD CHAR(80) INIT('KARTE1');
DCL OFF BIN FIXED(31);
STORE POOL(POOL) FROM(CARD) SET(OFFS);
    /* OFFS : 0  */
STORE POOL(POOL) FROM(CARD) SET(OFFS);
    /* OFFS : 80  */
STORE POOL(POOL) FROM(CARD) TO(160);
DCL NAK BIN FIXED(31);
ATTRIBUTE POOL ACT(NAK);
/*  NAK  :  240 */
STORE POOL(POOL) FROM(CARD) SET(OFFS);
    /*  OFFS : 160  */
CARD = 'KARTE2';
STORE POOL(POOL) FROM(CARD) TO(80);
    /* Überschreiben der 2. Karte  */
```

STORE
POOL
PL/1 data

Syntax

STORE POOL(poolreference) FROM(pl/1data)
[TO(key_1)] [SET(key_2)] ;

key_1::= pooloffset |
pooloffset,indexlist |
,indexlist

Erläuterung

1) Die durch pl/1data bezeichneten Daten (Datenelement, Feld, Struktur) werden in den Pool geschrieben. Der Pool muß mit DATA OUTPUT oder DATA UPDATE eröffnet sein.

2) Wenn die Option pooloffset,indexlist) in der TO-Klausel verwendet wird, wird indexsequentiell auf ein im Pool gespeichertes Subarrayblattelement mit Hilfe der indlist zugegriffen und ihm der Wert pl/1data zugewiesen. Diese Form führt zu Fehler, wennn bei pooloffset kein Descriptor einer dynamischen Datenstruktur gespeichert wurde.

3) Fehlt die TO-Klausel oder der pooloffset in der TO-Klausel, wird sequentiell geschrieben, d.h. hinter die zuvor geschriebenen Daten. Dabei wird der next-I/O-Zeiger um die Recordlänge hochgezählt.

4) Wird die TO-Klausel verwendet, so wird der Record unter dem Pooloffset key_1 abgelegt. Der next-I/O-Zeiger wird nicht beeinflußt, wohl aber unter Umständen der Füllstand. (Es können Daten überschrieben werden!).

5) Mit der SET-Klausel wird der Pooloffset des gerade abgespeicherter Records zurückgeliefert. Er ist bei TO(key_1) gleich key_1.

Beispiel

Es sei vereinbart:

```
DCL A, B(K) DESCRIPTOR;
DCL N  DYNAMIC;
```

Dann sind

```
A
A( )
A(3)
A(I,J)
A(2,A(1,I) ⟶ N)
B(4)( )
B(4)(B(3)(2,1) ⟶ N)
```

syntaktisch richtige Substructure-Referenzen.

.
.
.

```
RESET A;
```

substructure
reference

Syntax

basedescriptor-reference [(dds-indexlist)] ;

Erläuterung

1) Eine Substructure-Referenz bezeichnet stets eine Teilmenge der über einen Basedeskriptor referierten Daten. Die Spezifikation des betreffenden Teilbaumes ist in Form einer dds-Indexliste in der Gestalt

   $(index_1, index_2, ...)$

   an die Basedeskriptor-referenz anzufügen.

2) Wenn die gesamte dynamische Datenstruktur (also nicht eine echte Teilmenge) referiert werden soll, so ist dds-indexlist leer.

3) Eine leere Indexliste muß vorhanden sein, wenn die Basisdeskriptorreferenz mit einer Indizierung endet. In allen anderen Fällen kann sie entfallen.

4) Substructure-Referenzen können nur in den Formen auftreten, die in diesem Handbuch explizit angegeben sind. Sie können z.B. nicht an die Stelle von PL/1-Ausdrücken treten.

Beispiel

```
DCL (A,B) DESCRIPTOR;
DCL X(3) DYNAMIC(B);
DEFINE A,(10) DESCRIPTOR;
DO I = 1 TO 10;
DEFINE A(I),(I) AS(X);
END;

DO J = 1 TO 10;
SWITCH A(J), B;
DO I = 1 TO J;
   B(I) -> X(*)  = 15.;
/* oder X(I) (*)  = 15.;  */
END;
SWITCH A(J), B;
END;
```

Alle Variablen von A(1,1) -> X(1) bis A(10,10) -> X(3) erhalten den Wert 15. zugewiesen.

```
RESET A;
RESET B;
```

SWITCH

Syntax

SWITCH substructure-reference$_1$, substructure-reference$_2$;

Erläuterung

1) Alle Datenbestände, die zuvor über substructure-reference$_1$ referierbar waren, sind nun über substructure-reference$_2$ referierbar und umgekehrt.

KAPITEL 11

PLS - HANDBUCH

In alphabetischer Reihenfolge:

- PLS-Anweisung
- Anweisungen in Statement- und Clause-Definitionen
- PLS-Funktionen

Beispiel

```
DCL  A  BIT(5);
ACTIVE  A,I1;
A='01011'B;
EXEC  CALL  SUB(A);;
I1=100;
EXEC;
  J1=I1+2;
END EXEC;
```

Erläuterung

Die Variablen A und I1 werden aktiviert, in den EXEC-Gruppen werden sie durch ihren Wert ersetzt. A als BIT-Variable ist nicht standardmäßig aktiv, I1 ist nicht aktiv, da nicht deklariert.
Die Anweisung EXEC CALL SUB(A);; generiert also den PL/1-Text:

```
CALL  SUB(01011);
```

Die zeite EXEC-Gruppe erzeugt:

```
J1=100+2;
```

ACTIVE

PLS-Anweisung in Definitionen

Syntax

ACTIVE name [,name] *;

Erläuterung

Die ACTIVE-Anweisung ist keine ausführbare Anweisung, sondern eine Deklaration, die innerhalb des PL/1-Blockes gültig ist, in dem sie steht. Die Variablennamen "name" werden, wenn sie in einer EXEC-Gruppe angetroffen werden, durch den Wert der Variablen dieses Namens ersetzt. Mit der ACTIVE-Anweisung können Variable aktiviert werden, die nicht standardmäßig aktiv sind, wie etwa Bitketten. Auch nicht explizit deklarierte Variable oder solche, denen durch eine DEFAULT-Anweisung Attribute zugewiesen werden, können nur durch die ACTIVE-Anweisung zu Ersetzungsvariablen werden.

Beispiel

```
IF BIDENTIFIER('JOI') THEN DØ;
     /* JOINTS */
     END;
ELSE IF BID('ELE',SKIP) THEN DØ;
     /* ELEMENTS */
     END;
ELSE IF BID('SUB',NOSKIP) THEN DØ;

     IF BID('SUBSTRU') THEN DØ;
        /* SUBSTRUCTURE */
        END;
     ELSE DØ;
          /* SUBELEMENT */
          END;
     END;
```

Erläuterung

Es soll festgestellt werden, ob die nächste Benennung in der Eingabe mit 'JOI', 'ELE', oder 'SUB' anfängt. Im Falle 'SUB' wird die Bennennung nicht übergangen (wegen NOSKIP) und es kann weiter geprüft werden, ob sie mit 'SUBSTRU' anfängt.

BIDENTIFIER

PLS-Funktion

Syntax

BIDENTIFIER (xyz [,SKIP | NOSKIP])

Erläuterung

Die Funktion BIDENTIFIER (Begin of IDENTIFIER) liefert einen logischen Wert (BIT(1)) zurück. Sie stellt fest, ob die Zeichen "xyz" den Anfang der nächsten Benennung (Identifier) in der Eingabe darstellen. Eine Bennenung ist eine Folge von Zeichen, bestehend aus Buchstaben, Ziffern und Zeichen $, #, @, -
Die Zeichenfolge muß mit einem Buchstaben oder #, $, @ beginnen. Benennungen oder Identifier sind also gültige PL/1-Variablennamen. Wird die Benennung gefunden, wird '1'B zurückgelifert und falls "SKIP" angegeben wurde, übergangen. Ist eine Benennung mit Anfang "xyz" nicht in der Eingabe vorhanden, wird '0'B zurückgegeben, der Eingabezeiger wird nicht verändert.

Beispiel

Eingabe : '??AXB!!++ XYZ//;'
BWORD('?') liefert '1'B, Eingabezeiger steht danach auf 'XYZ'.

BWORD('??AXB',NOSKIP) liefert '1'B, Eingabezeiger steht weiterhin auf '??'.

BWO('?? AXB') liefert '0'B wegen des Leerzeichens.

BWO('?') & BWO('X') liefert zweimal '1'B, Eingabezeiger steht auf dem Semikolon.

```
IF  BWORD('?',NOSKIP) | BWO('!',NOSKIP)
   |BWORD('/',NOS) THEN ...
```

liefert '1'B | '0'B | '0'B, der Eingabezeiger steht weiterhin auf '??'.

PLS-Funktion

Syntax

BWORD (xyz [,SKIP | NOSKIP])

Erläuterung

Die Funktion BWORD (Begin of WORD) liefert einen logischen Wert (BIT(1)) zurück. Sie stellt fest, ob die Zeichen "xyz" als nächstes in der Eingabe vorhanden sind. Alle Zeichen außer Leerzeichen und Semikolon sind gültig. Werden die Zeichen "xyz" gefunden, wird '1'B zurückgegeben und bei SKIP das gesamte Wort in der Eingabe übergangen.
Ein Wort ist in diesem Sinne eine Folge von Zeichen außer Leerzeichen und Semikolon, abgeschlossen durch Leerzeichen oder Semikolon.

Beispiel

```
CLAUSE 'CLAUSE_NAME_1';
  /*Clause-Definition*/
END CLAUSE;
CLA 'CLAUSE2';
  /*Clause-Definition          */
  /*Aufruf von CLAUSE_NAME_1: */
  LINK CLAUSE_NAME_1;
END CLA;
```

PLS-Anweisung

Syntax

CLAUSE name;

Erläuterung

Die CLAUSE-Anweisung ist der Anfang einer CLAUSE-Definition. Eine CLAUSE kann einen Teil einer POL-Anweisung abarbeiten. Sie wird aus STATEMENT- und CLAUSE-Anweisungen aufgerufen. "name" ist der Name der CLAUSE. Er dient zur Identifikation, z.B. beim Löschen einer CLAUSE und zum Aufruf. "name" ist eine Zeichenkette, eingeschlossen in Apostrophe, maximal 32 Zeichen lang. Sie darf keine Leerzeichen enthalten.

Beispiel

```
CLAUSE END;
   PUT LIST('SUBSYSTEM SUB1: END OF TRANSLATION');
   EXEC PUT LIST('SUBSYSTEM SUB1: END OF EXECUTION');;
   EXEC;
      CLOSE FILE (#F1);
      FREE #D1;
   END EXEC;
END CLAUSE;
```

Erläuterung

Wenn das Subsystem SUB1 durch: 'END SUB1;' abgeschlossen wird, wird die End-Clause aktiv. Sowohl zur Übersetzungszeit als auch zur Ausführungszeit wird eine Meldung ausgegeben. Der File #F1 wird geschlossen, die Datenstruktur #D1 wird freigegeben. Folgende PL/1-Anweisungen werden durch obige End-Clause erzeugt:

```
PUT LIST('SUBSYSTEM SUB1: END OF EXECUTION');
CLOSE FILE(#F1);
FREE #D1;
```

Beispiel

Die END-Subsystem-Anweisung kann eine LEAVE-OPTION enthalten:

```
END SUB  LEAVE.... ;
```

Die END-CLAUSE arbeitet das LEAVE ab:

```
CLAUSE END;
  IF ID('LEAVE') THEN DØ;
    .
    .
  END;
END CLAUSE;
```

CLAUSE
END

PLS-Anweisung

Syntax

CLAUSE END;

Erläuterung

Die CLAUSE END - Anweisung definiert eine END-CLAUSE, die nur beim Subsystem-Abschluß aktiv wird. Sie wird bei der END-Subsystem-Anweisung aufgerufen. Sie wird jedoch nicht aufgerufen am Ende einer externen subsystemspezifischen POL-Prozedur. Die END-CLAUSE kann z.B. zum Schließen von Files oder zum Freigeben von allokiertem Speicherplatz benutzt werden.

Die END-CLAUSE kann außerdem zusätzliche Angaben auf der END-Subsystem-Anweisung abarbeiten.

Beispiel

```
CLAUSE INITIAL;
   PUT LIST('SUBSYSTEM SUB1 STARTS TRANSLATION');
   EXEC PUT LIST('SUBSYSTEM SUB1 STARTS EXECUTION');;
   EXEC;  /* INITIALISE COMMON  */
      SUB1.#COMVAR1=SQRT(NEXT_EX);
      SUB1.#COMVAR2=0;
   END EXEC;
END CLAUSE;
```

Erläuterung

Wenn das Subsystem SUB1 aufgerufen wird, z.B. durch: 'ENTER SUB1 2*N;' , wird die Initialisierungs-Clause aktiv. Sowohl zur Übersetzungszeit als auch zur Ausführungszeit wird eine Meldung ausgegeben. Die COMMON-Variable #COMVAR1 wird initialisiert mit SQRT(NEXT_EX), also mit der Wurzel des Ausdrucks auf der ENTER-Anweisung. Folgende PL/1-Anweisungen werden für obiges Beispiel erzeugt:

```
PUT LIST('SUBSYSTEM SUB1 STARTS EXECUTION');
          SUB1.#COMVAR1=SQRT(2*N);
          SUB1.#COMVAR2=0;
```

CLAUSE
INITIAL

PLS-Anweisung

Syntax

CLAUSE INITIAL;

Erläuterung

Die INITIAL CLAUSE-Anweisung definiert eine Initialisierungs-CLAUSE, die nur beim Subsystemstart aktiv wird. Sie kann zum Initialisieren des Subsystem-Common und zum Abarbeiten des Rests der "ENTER subsystemname"-Anweisung dienen. Sie wird nicht aufgerufen am Anfang einer externen subsystemspezifischen POL-Prozedur, kann also nicht zum Initialisieren der Makrozeit-Datenstruktur benutzt werden.

Beispiel

```
COMPRESS;

COM;
```

COMPRESS

PLS-Anweisung

Syntax

COMPRESS;

Erläuterung

Die PLS-Systembibliotheken für die Anweisungsdefinitionen und die Tabellen sind Partitioned Data Sets (PDS) des OS/360. Sie müssen von Zeit zu Zeit komprimiert werden, um die Lücken zu schließen und ein Überlaufen zu verhindern. PLS komprimiert die Bibliotheken dann automatisch am Subsystemende, wenn der freie Platz weniger als 10% des Gesamtplatzes des Datasets ausmacht. Durch die COMPRESS-Anweisung kann unabhängig davon ein Komprimieren erzwungen werden.

Anmerkung: Wenn das Betriebssystem den dynamischen Aufruf von IEBCOPY nicht erlaubt, so ist das Komprimieren in einem eigenen Step durch // EXEC PGM=IEBCOPY auszuführen.

Beispiel

```
CPARMS  'MACRO,LIST,XREF,A,AG';

CPAR 'STORAGE';

DCL CCC CHAR(50) VARYING;
CCC = 'XREF'|| SUBSTR(PARM,I,J);
CPARMS CCC;
```

CPARMS

PLS-Anweisung

Syntax

CPARMS expression;

Erläuterung

Die Statement- und Clause-Definitionsprogramme benutzen zur Erzeugung eines Objektmoduls einer Treiberroutine den PL/1-Optimizing Compiler des OS/370. Mit der CPARMS-Anweisung können die Compiler-Parameter geändert werden. "expression" ist ein Zeichenkettenausdruck von maximal 92 Zeichen Länge.
Default-Parameter für den Compiler:

'INCLUDE,NA,NAG,NOP,NSTG,SMSG,NX'.

Beispiel

```
DATA 'PROCEDURES';
  #SUB1_R1: PROC;
      .
      .
      .
    END #SUB1_R1;
  #SUB1_R2: PROC;
      .
      .
      .
    END #SUB1_R2;
END DATA;

DATASTRUCTURE  'D1';
   DCL (SUB1_I,SUB1_J,SUB1_K) BIN FIXED;
   DCL 1 #SUB1_BASED BASED(#SUB1_P),
          2 #SUB1_NEXT PTR INIT(NULL()),
          2 #SUB1_PREVIOUS PTR,
          2 #SUB1_DATA PTR INIT(NULL());
   DCL #SUB1_CHAR CHAR(128) VAR;
   CALL #SUB1_INIT(SUB1_I, #SUB1_CHAR);
END DAT;
```

Erläuterung

Im ersten Beispiel werden 2 interne Prozeduren definiert. Im zweiten Beispiel sind die Variablen SUB1_I, SUB1_J und SUB1_K für den Anwender zugänglich, sie müssen im Subsystem-Handbuch erläutert werden. Alle anderen Namen sind durch # geschützt.

DATASTRUCTURE

PLS-Anweisung

Syntax

DATASTRUCTURE name;

Erläuterung

Die DATASTRUCTURE-Anweisung dient dazu, Subsystem-Datenstrukturen zu definieren. Die PL/1-Deklarationen der gewünschten Datenstrukturen werden angegeben. Diese Deklarationen werden beim Übersetzen von POL-Anweisungen nach dem Eröffnen eines Subsystems (ENTER subname;) in das erzeugte PL/1-Programm eingeschoben und stehen somit für die Dauer der Subsystem-Anwendung zur Verfügung. Beliebig viele Subsystem-Datendeklarationen können angegeben werden. Jede Folge von Deklarationen wird durch einen Namen identifiziert; mit diesem Namen kann die Deklaration wieder gelöscht werden. 'name' ist eine Zeichenkette mit maximal 32 Zeichen ohne Leerzeichen. Die Datenstrukturen werden auch in jedes subsystemspezifische externe POL-Programm eingefügt. Außer Deklarationen kann eine Datenstruktur auch PL/1-Anweisungen enthalten, z.B. interne Unterprogramme, die für die Subsystemausführung erforderlich sind. Die in den Deklarationen verwendeten Namen, auf die der Subsystemanwender nicht zugreifen darf, müssen geschützt werden. Dies geschieht dadurch, daß sie mit dem "Protection character" # beginnen.

Beispiel

```
DATA COM;
     DCL 1,
        2 (#SUB1, #SUB2, #SUB3) ENTRY(PTR,CHAR(*)) DYNAMIC,
        2 (#EXT EXTERNAL, #DYN DYNAMIC) ENTRY (PTR,BIN,FILE),
        2 #BAS DESCRIPTOR,
        2 #BANK BANK;
END DATA;

DATASTRUCTURE  COMMON;
     DCL 1,
        2 #FILE1 FILE INIT(SYSPRINT),
        2 #I BIN FIXED(15) INIT(0),
        2 #E1 ENTRY DYNAMIC,
        2 USER_TEXT CHAR(32) VARYING;
END DATASTRUCTURE;
```

Erläuterung

Im Subsystem SUB1 können die Common-Variablen im zweiten Beispiel wie folgt referiert werden:

In PLR-Moduln:

```
OPEN FILE(#FILE1);
CLOSE FILE(SUB1.#FILE1)
I=I+1;
CALL #E1;;
```

In Statement- und Clause-Definitionen:

```
EXEC OPEN FILE( #FILE1);;
EXEC SUB1. #I=NEXT_INTEGER;;
EXEC LINK #E1;;
```

In POL-Programmen sind die Namen, die durch # geschützt sind, nicht zugänglich. Die Verwendung von Namen, die mit dem Protection character beginnen, ist dem Anwendungsprogrammierer (ebenso wie Namen, die mit QQ beginnen) grundsätzlich verboten. In POL-Programmen kann also lediglich der ungeschützte Name USER_TEXT verwendet werden:

```
USER_TEXT='PROBLEM NR.1';
PUT LIST(USER_TEXT);
```

DATASTRUCTURE
COMMON

PLS-Anweisung

Syntax

DATASTRUCTURE COMMON;

Erläuterung

Eine ausgezeichnete Datenstruktur für jedes Subsystem ist der Subsystem-COMMON. Eine Deklaration in Form einer einzigen PL/1-"Structure" wird benutzt, um alle diejenigen Subsystem-Daten aufzunehmen, die nicht nur in der POL, sondern auch in allen Subsystem-Moduln ansprechbar sein sollen. Auch alle externen subsystemspezifischen POL-Routinen können auf den Subsystem-Common zugreifen.
Der Subsystem-COMMON muß deklariert werden, bevor er in einem Modul benutzt werden kann. Wird während der Subsystem-Entwicklung der Common geändert, müssen alle Module neu übersetzt werden. Module, die eine alte Version des COMMON enthalten, erzeugen bei ihrem Aufruf die Fehlermeldung

COMMON NOT UP TO DATE IN PROCEDURE name.

"name" ist der Name der Prozedur mit dem alten COMMON. Die Nachricht ist vom LEVEL ERROR, durch MESSAGE INACTIVE ERROR kann Sie unterdrückt werden. Man kann so für begrenzte Zeit mit einer alten COMMON-Version ohne Neuladen der Module arbeiten.
Am Beginn der Subsystem-COMMON-Definition muß eine Level-1 "structure"-Deklaration folgen. Der Name der "major structure", also der äußersten Strukturebene, wird nicht angegeben; er wird gleich dem Subsystemnamen gesetzt.
Beispiel: DCL 1, 2 A BIN FIXED(15), 2 B CHAR(10);.

DATASTRUCTURE
COMMON
(2)

Zulässige Deklarationsattribute sind alle diejenigen, die auch in einer PL/1-BASED-Struktur ohne REFER-OPTION zulässig sein, also insbesondere keine variablen Dimensionierungen und Längen, keine Storage-Class-Attribute. Zusätzlich sind die REGENT-Attribute DYNAMIC ENTRY, BANK, POOL und DESCRIPTOR zulässig. Für jeden zu dem Subsystem gehörenden auf der PLS-Ebene dynamisch aufrufbaren Modul (also für jeden Modul, dessen Name in einer EXEC LINK-Anweisung in einer STATEMENT- oder CLAUSE-Definition erscheint) ist im Subsystem-Common oder einer anderen Datenstruktur eine "DECLARE... ENTRY(...)DYNAMIC"-Deklaration erforderlich. Alle Argumente sind in der DYNAMIC ENTRY-Deklaration zu deklarieren.

Der Subsystem-Common wird beim Subsystem-Start angelegt und initialisiert. Die Initialisierung ist möglich durch Angabe von INIT-Optionen in der Deklaration oder durch Anweisungen, die in einer "INITIAL CLAUSE" stehen. Alle externen POL-Prozeduren können auf den Common zugreifen. Dies erfolgt über einen Pointer, der in der REGENT-Datenstruktur (QQ als erstes Argument!) übergeben wird. Die COMMON-Deklaration wird von PLS in der Bibliothek REGENT. PLSTTRAN.DATA gespeichert. Der Name des Members lautet prCOMM, "pr" ist der zweibuchstabige Subsystemprefix, der von PLS beim Initialisieren des Subsystems erzeugt wird, er kann durch die LIST SUBSYSTEMS-Anweisung ausgedruckt werden. In alle PLR-Module und Routinen wird der Subsystem-COMMON automatisch eingefügt. Als Name der COMMON-Struktur wird der Subsystemname verwendet. In PLR können die COMMON-Elemente durch subsystemname.elementname referiert werden. Namen, die der Anwender des Subsystems nicht verwenden soll, müssen durch den "Protection character"# geschützt werden, sie müssen mit diesem Zeichen beginnen.

Beispiel

```
DESTROY CLAUSE 'C1';
DEST CLA 'X1', 'X2', 'X3';
DCL CLAUSES(8) CHAR(32) VARYING
    INIT ('CC1', 'CC2', 'CC3', 'CC4','CC5', 'CC6',
          'CC7', 'CC8');
DO I=1 TO 8;
    DESTROY CLAUSE CLAUSES(I);
    END;
```

DESTROY
CLAUSE

PLS-Anweisung

Syntax

DESTROY CLAUSE name [,name]* ;

Erläuterung

DESTROY CLAUSE zerstört die aufgeführten Clauses mit dem Namen "name". "name" ist ein Zeichenkettenausdruck mit maximal 32 Zeichen Länge. Er darf keine Leerzeichen enthalten (auch nicht am Schluß).

Beispiel

```
DESTROY DATASTRUCTURE 'D1', 'D2';
DEST DATA 'DATA05';
DO I=N TO M;
    DEST DATA 'DD' || NAME(I);
    END;

DEST DATA COMMON;
```

DESTROY
DATASTRUCTURE

PLS-Anweisung

Syntax

DESTROY DATASTRUCTURE name [,name]* ;

Erläuterung

DESTROY DATASTRUCTURE löscht die aufgeführten Subsysteme-Datenstrukturen mit dem Namen 'name'. 'name' ist ein Zeichenkettenausdruck mit maximal 32 Zeichen. Er darf keine Leerzeichen enthalten.

DESTROY DATASTRUCTURE COMMON; löscht den Subsystem-COMMON.

Beispiel

```
SUBSYSTEM 'SUB1';
DESTROY END CLAUSE;
SUBSYSTEM 'SUB2';
DEST END;
```

Erläuterung

Die End-Clauses der Subsysteme SUB1 und SUB2 werden zerstört.

DESTROY
END CLAUSE

PLS-Anweisung

Syntax

DESTROY END [CLAUSE];

Erläuterung

DESTROY END CLAUSE löscht die End-Clause des gerade behandelten Subsystems.

Beispiel

```
SUBSYSTEM 'SUB1';
DESTROY INITIAL CLAUSE;
SUBSYSTEM 'SUB2';
DEST INIT;
```

Erläuterung

Die Initialisierungs-Clauses der Subsysteme SUB1 und SUB2 werden zerstört.

PLS-Anweisung

Syntax

DESTROY INITIAL [CLAUSE] ;

Erläuterung

DESTROY INITIAL CLAUSE löscht die Initialisierungs-Clause des gerade behandelten Subsystems.

Beispiel

```
DESTROY MACROTIME DATASTRUCTURE;
DEST MAC DAT;
DES MAC;
```

DESTROY
MACROTIME
DATASTRUCTURE

PLS-Anweisung

Syntax

DESTROY MACROTIME [DATASTRUCTURE] ;

Erläuterung

DESTROY MACROTIME DATASTRUCTURE löscht die Makrozeit-Datenstruktur des gerade behandelten Subsystems.

Beispiel

```
DESTROY STATEMENT 'PLOT', 'PRINT';
DEST STAT 'GET', 'PUT', 'READ', 'WRITE';
DEST STAT NAME 'GOTO', DATATYPE ASS, N'STOP', D INT;
DO I=1 TO N;
   DEST STAT NAME STATNAME(I);
   END;
```

DESTROY
STATEMENT

PLS-Anweisung

Syntax

DESTROY STATEMENT stat [,stat]* ;

stat::= NAME name
| DATATYPE type

Erläuterung

DESTROY STATEMENT löscht die angeführten POL-Anweisungen. 'name' ist das Schlüsselwort der POL-Anweisungen, die mit einem festen Schlüsselwort beginnen. 'name' darf keine Leerzeichen enthalten (auch nicht am Ende). 'type' ist der Typ der Datentyp-Anweisungen. die mit einem bestimmten Datentyp beginnen. 'type' kann REAL, INTEGER, OPERATOR, IDENTIFIER, STRING, BITSTRING, ASSIGNMENT oder %ASSIGNMENT sein. DESTROY STATEMENT ASSIGNMENT schaltet die PL/1-Zuweisung aus. Auch andere PL/1-Anweisungen können ausgeschaltet werden. DEST STAT %AS schaltet die Makroprozessor-Zuweisung aus.

Beispiel

```
DESTROY SUBSYSTEM 'SUB1' KEY'KEY1';
DEST SUB 'SUB2' KEY 'SUB2KEY';
DESTROY 'SUB3' 'KEY3';
DES 'SUB4';
```

DESTROY
SUBSYSTEM

PLS-Anweisung

Syntax

DESTROY [SUBSYSTEM] name [[KEY] key] [NOMEMBERS];

Erläuterung

DESTROY SUBSYSTEM löscht ein Subsystem mit dem Namen 'name', falls der richtige Schlüssel 'key' angegeben wird. Defaultwert für 'key' ist (wie bei der INITIATE SUBSYSTEM-Anweisung) der Nullstring ". 'name' und 'key' sind Zeichenketten von maximal 32 Zeichen Länge, sie dürfen keine Leerzeichen enthalten. Alle zum Subsystem gehörenden Zuweisungs- und Datendefinitionen, alle Tabellen und Module, die sich auf REGENT-Bibliotheken befinden, werden gelöscht. Subsystem-Datenbanken werden nicht gelöscht, ihre Verwaltung obliegt der Eigenverantwortung der Subsystemanwender. Subsystem-Module auf der Bibliothek REGENT.MODS werden nicht gelöscht, wenn NOMEMBERS angegeben wird. Routinen auf der Routinenbibliothek REGENT.LOAD werden in keinem Fall gelöscht.

Beispiel

```
CLAUSE 'C1';
  '
  '
  '
END CLAUSE;

CLA 'C2';
  '
  '
  '
END CLA;

END CLAUSE;
```

Erläuterung

Die letzte END CLAUSE-Anweisung ist ungültig, da sie nicht am Ende einer CLAUSE-Definition steht. Daher wird die letzte Anweisung als normales END-statement interpretiert.

END
CLAUSE

PLS-Anwiesung

Syntax

END CLAUSE;

Erläuterung

END CLAUSE schließt eine CLAUSE-Definition ab. Sie ist nur gültig, wenn vorher eine CLAUSE-Anweisung die CLAUSE-Definition eröffnet hat.

Beispiel

```
DATASTRUCTURE  'D1';
  '
  '
  '
END DATASTRUCTURE;
DATA COMMON;
  '
  '
  '
END DATA;

END DAT;
```

Erläuterung

Die letzte END DAT - Anweisung ist ungültig, da sie nicht am Ende einer Datenstruktur-Definition steht. Daher wird die letzte Anweisung als normales END-statement interpretiert.

END
DATASTRUCTURE

PLS-Anweisung

Syntax

END DATASTRUCTURE;

Erläuterung

END DATASTRUCTURE schließt eine Datenstruktur-Definition ab. Sie ist nur gültig, wenn vorher eine DATASTRUCTURE-Anweisung die Datenstruktur-Definition eröffnet hat.

Beispiel

```
EXEC;
   statement1;
   statement2;
    '
    '
    '
END EXEC;

EXECUTE;
    '
    '
    '
END EXECUTE;


END EXEC;
```

Erläuterung

Die letzte END EXEC-Anweisung ist ungültig, da sie nicht am Ende einer EXEC-Gruppe steht. Die Anweisung wird ignoriert.

END
EXECUTE

PLS-Anweisung in Definitionen

Syntax

END EXECUTE;

Erläuterung

END EXEC schließt eine EXEC-Gruppe in STATEMENT- und CLAUSE-Definitionen ab. Die Anweisung ist nur gültig, wenn vorher eine EXEC-Gruppe durch 'EXEC;' eröffnet wurde.

Beispiel

```
MACROTIME DATASTRUCTURE 'MD1';
    DCL 1 A,
          2 L BIN FIXED(15) INIT(0),
          2 P PTR INIT(NULL()),
          2 C CHAR(8) INIT('SUB1');
END MACROTIME DATASTRUCTURE;
MACRO DATA 'MD2';
   :
END MACRO DATA;
MAC DAT 'MD3';
   :
END MAC;

END MAC DAT;
```

Erläuterung

Die letzte END-Anweisung ist fehlerhaft, da sie nicht am Ende einer Makrozeit-Datenstruktur-Definition steht. Die Anweisung wird von PLS als PL/1-END-Anweisung interpretiert und von PL/1 als fehlerhaft beanstandet.

END
MACROTIME
DATASTRUCTURE

PLS-Anweisung

Syntax

END MACROTIME [DATASTRUCTURE] ;

Erläuterung

Diese Anweisung schließt die Definition einer Makrozeit-Datenstruktur ab. Sie ist nur gültig, wenn vorher durch 'MACRO DATA' die Definition einer Makrozeit-Datenstruktur eröffnet wurde.

Ein Subsystem kann nur eine einzige Makrozeit-Datenstruktur besitzen.

Beispiel

```
STATEMENT 'ST1';
  '
  '
  '
END STATEMENT;
STA DATATYPE INTEGER;
  '
  '
  '
END STA;
END STAT;
```

Erläuterung

Die letzte END STAT-Anweisung ist ungültig, da sie nicht am Ende einer STATEMENT-Definition steht. Daher wird die letzte Anweisung als normales END-statement interpretiert.

END
STATEMENT

PLS-Anweisung

Syntax

END STATEMENT;

Erläuterung

END STATEMENT schließt eine STATEMENT-Definition ab. Sie ist nur gültig, wenn vorher eine STATEMENT-Anweisung die STATEMENT-Definition eröffnet hat.

Beispiel

1.
```
EXECUTE PUT LIST('STEP1') SKIP;;
EXEC;
   PUT LIST('STEP2') SKIP;
END EXEC;
```

2. Eine Liste von Werten und deren Anzahl soll an eine Routine übergeben werden. POL-Anweisung:

RECHNE x_1, y_1 ... , x_i, y_i ... ; x_i, y_i Realzahlen

Generierte Anweisungen:

```
BEGIN;
X(1) = x1;
Y(1) = y1;
X(2) =
      :
      :
      :
DCL (X(N), Y(N)) DECIMAL FLOAT;
CALL RECHNE (X,Y,N);
END;
```

Statementdefinition:

```
STATEMENT 'RECHNE'; DCL I BIN FIXED(15);
EXEC BEGIN;;
I=0;
DO WHILE (TYP¬ = 10 & TYP¬ = 15);
I=I+1;
EXEC X(I) = NEXT_REAL;;
SKIP(',');
EXEC Y(I) = NEXT_REAL;;
SKIP(',');
END;
EXEC;
DCL (X(I), Y(I)) DECIMAL FLOAT;
CALL RECHNE (X,Y,I);
END;
END EXEC;
END STATEMENT;
```

PLS-Anweisung in Definitionen

Syntax

EXECUTE exectext ;

Erläuterung

Die Anweisungen EXEC und END EXEC umschließen die bei der Abarbeitung der POL-Anweisung zu generierenden PL/1-Anweisungen. Während der Übersetzungsphase des PLS-Übersetzers wird eine solche "EXEC-Gruppe" nicht ausgeführt, sondern in das erzeugte PL/1-Programm kopiert. Erst zur Ausführungszeit des generierten Programms werden die zwischen EXEC und END EXEC stehenden Anweisungen aktiv. Soll nur eine Anweisung erzeugt werden, kann sie auch zwischen "EXEC" und ";" stehen:

```
EXEC PUT LIST('BEISPIEL')SKIP;;
```

Dabei ist zu beachten, daß sowohl das Semikolon für die EXEC-Anweisung als auch das für die zu generierende Anweisung stehen muß. Mit Hilfe der Kurzform der EXEC-Anweisung lassen sich auch Teile von Anweisungen generieren: "EXEC MULT (A,B);" erzeugt den Teil einer PL/1-Anweisung "MULT(A,B)" (ohne Semikolon).

In einer EXEC-Gruppe können beliebige PL/1-Anweisungen stehen. Es ist darauf zu achten, daß das erzeugte Programm auf jeden Fall syntaktisch richtig wird. Labels vor Anweisungen sind problematisch, da bei mehrmaliger Anwendung der gleichen POL-Anwendung das Label mehrfach deklariert wäre. Man kann entweder jedesmal ein neues Label mit Hilfe einer Makrozeit-Variablen generieren oder die Gruppe von Anweisungen, die das Label enthält, zwischen BEGIN; oder END; setzen.

Werden durch eine POL-Anweisung mehrere PL/1-Anweisungen erzeugt, so ist es erforderlich, sie als DO-Gruppe zu generieren.

Beispiel: Die POL-Anweisung "DRUCKE ALLES;" soll die Anweisungsfolge "PUT EDIT (HEADLINE) (SKIP,A); CALL PRINT(3); erzeugen. Wenn der POL-Programmierer nun schreibt: "IF X >=0 THEN DRUCKE ALLES;" und die erzeugten PL/1-Anweisungen stehen nicht zwischen DO und END, ergeben sich Resultate, die der POL-Programierer nicht erwartet

Erläuterung

Durch EXEC BEGIN;; wird 'BEGIN;' erzeugt, danach werden in einer Schleife die Zuweisungen

```
X(1) = ... Y(1) = ...
X(2) = ... Y(2) = ...      usw.
```

erzeugt. Am Ende wird die Deklaration für X und Y und der Aufruf auf RECHNE generiert.

Dieses Beispiel zeigt auch, daß die generierten PL/1-Anweisungen in bestimmter Weise variiert werden können. In dem Beispiel wird im generierten Text der Name "I" durch den Wert der Variablen I ersetzt, ebenso wird der Name "NEXT_REAL" durch den Wert der PLS-Funktion NEXT_REAL ersetzt. Variable, deren Namen in der EXEC-Gruppe durch ihren Wert ersetzt werden, heißen "Ersetzungsvariable", ihr Wert heißt "Ersetzungswert". ("Replacement variable", "replacement value").
Eine Ersetzung findet nur für aktive Variable statt. Aktiv sind neben den PLS-Funktionen alle einfachen arithmetischen oder Zeichenkettenvariablen (mit den Attributen BINARY, DECIMAL, PICTURE oder CHARACTER) innerhalb des Blocks, in dem sie durch eine DECLARE-Anweisung deklariert sind. Im obigen Beispiel sind also die Variablen I (da als BINARY deklariert) und NEXT_REAL (da PLS-Funktion) aktiv. Mit Hilfe von ACTIVE und UNACTIVE-Anweisungen können unaktive Variable aktiviert und aktive desaktiviert werden. Felder von Variablen oder Feldelemente dürfen nicht als aktive Variable benützt werden.

(im Beispiel würde also die zweite Anweisung, CALL PRINT(3);, nicht in der THEN-Clause stehen). Falls erforderlich, kann auch "BEGIN;" und "END;" die erzeugten Anweisungen umschließen. Dies ist immer dann nötig, wenn DECLARE-Anweisungen generiert werden, um doppelte Deklarationen zu vermeiden.

Innerhalb des durch eine EXEC-Gruppe generierten PL/1-Textes werden aktive Ersetzungsvariable und PLS-Funktionsaufrufe durch ihren Wert ersetzt. Ersetzungsvariable sind alle in der STATEMENT- oder CLAUSE-Definition deklarierten BINARY, DECIMAL, CHARACTER und PICTURE-Variablen.

Beispiele

```
1.    DATASTRUCTURE 'DYN_ENT'; / Deklariere Dynamic Entries*/
         DCL 1,
            2 (A,B,C) DYNAMIC ENTRY(BIN FIXED(15)) MODULE('MOD1'),
            2 D ENTRY DYNAMIC ENTRYNAME('ENTRYD') MODULE('MM');
      END DATA;
      STAT 'EVAL.UIERE';
      EXEC LINK A(3);;          /*Rufe A dynamisch  */
      EXEC LINK B(I);;          /*Rufe B dynamisch  */
      EXEC;
      LINK C (111);             /*Rufe C dynamisch  */
      LINK D;                   /*Rufe D dynamisch  */
      END EXEC;
      END STAT;
```

Erläuterung

Die DYNAMIC-ENTRY-Konstanten A,B,C und D werden in der Datenstruktur 'DYN_ENT' deklariert und werden in der Statement-Definition für Evaluierte in EXEC-Gruppen angesprochen. Der Modul MOD1 wurde im Modulgenerator erzeugt, die xModule-Anweisung lautet:

```
xMODULE MOD1 ENTRIES(A,B,C);
```

```
2.    STAT 'NAM.REAL';
      EXEC LINK NAMREAL (NEXT_STRING, NEXT_REAL);;
      END STAT;
```

Erläuterung

Hier werden die beiden Argumente für die dynamisch aufzurufende Routine NAMREAL aus der POL-Anweisung mittels der PLS-Funktionen NEXT_STRING und NEXT_REAL gewonnen. "NAMREAL 'ABC' 1.5;" würde also übersetzt in: "LINK NAMREAL ('ABC',1.5)".

EXECUTE
LINK

PLS-Anweisung in Definitionen

Syntax

EXECUTE LINK name [(argumente)] ;;

Erläuterung

Innerhalb einer EXEC-Gruppe ist auch die Nicht-PL/1-Anweisung LINK zulässig. Diese Anweisung dient dazu, zur Ausführungszeit des POL-Programms Entries in PLR-Moduln dynamisch aufzurufen. 'name' muß der Name einer DYNAMIC-ENTRY-Konstanten oder DYNAMIC-ENTRY-Variablen sein. Das dynamische Laden und Ausführen des Moduls wird von der REGENT-Modulverwaltung RMM vorgenommen. Der Modul muß daher ein vom REGENT-Modulgenerator erzeugter Modul sein.'argumente' sind die an die Routine zu übergebenden Argumente. Ein Entry, der in einem Subsystem durch EXEC LINK aufgerufen wird, muß in einer Subsystem-Datenstruktur als DYNAMIC ENTRY mit allen Argumenten (falls vorhanden) deklariert sein.

Beispiel

```
EXECUTE MESSAGE TEXT('START OF SUB1');;

EXEC MESSAGE SEVERE TEXT('0017 NOT ENOUGH SPACE ON BANK');;

EXEC;
   MESSAGE TEXT('0099 OC1 IN SUB2') T;
   STOP;
END EXEC;

EXEC MESS E TEXT('0026'|| ERRTEXT);;
```

EXECUTE
MESSAGE

PLS-Anweisung in Definitionen

Syntax

EXECUTE MESSAGE [level] [text] ;;

level::= D | DEBUG
| I | INFORMATIVE
| W | WARNING
| E | ERROR
| S | SEVERE
| T | TERMINAL

text::= TEXT (expression)

Standardwerte: I, TEXT('')

Erläuterung

In einer EXEC-Gruppe kann auch eine MESSAGE-Anweisung stehen. Hier ist nur eine Nachrichtenerzeugung, nicht die Nachrichtensteuerung (z.B. MESSAGE ACTIVE) erlaubt. Die Syntax der MESSAGE-Anweisung entspricht der Syntax der PLR-Anweisung und der System-Anweisung MESSAGE. (Siehe Kap.6.8, Seite 6-41)'expression' muß ein Zeichenkettenausdruck sein.

Beispiel

```
FILE;
FILE ALL;
FILE ON LIBRARY ('DD1');
FILE ('A', ('B', 'C'), 'D');
FILE ((NAME 'X1', DATA INT, DATA REAL, MODUL 'X2', NAME 'X3'))
     LIB 'PLSLIB1';
```

Erläuterung:

Bei 1) und 2) werden alle bisher definierten CLAUSES und STATEMENTS auf die Bibliothek REGENT.PLSTRAN.MODS geladen.
Bei 3) werden die Treibermodule auf eine Bibliothek mit dem DD-Namen geladen.
Bei 4) werden nur die Statements A, B, C und D geladen, dabei werden B und C in einen Modul zusammengebunden.
Bei 5) wird ein Treibermodul erzeugt mit den Statements mit den Namen X1 und X3, mit den Datentyp-Statements, die mit REAL- oder INTEGER-Zahlen anfangen und mit dem bereits erzeugten Treibermodul X2. Der Modul wird auf die Bibliothek mit dem DD-Namen PLSLIB1 geladen.

FILE

PLS-Anweisung

Syntax

```
FILE   [ALL  |  (gruppe  [ ,gruppe]* )
       [ON][LIBRARY      ddname ] ;

gruppe::=      name | (name  [,name]* )

name::=        DATATYPE dtype | NAME  rname
             | MODUL   modulname
```

Erläuterung:

Die File-Anweisung ist normalerweise unnötig. Mit ihr kann das Binden der STATEMENT- und CAUSE-Treiber-Module gesteuert werden. Die PLS-Anweisungen STATEMENT und CLAUSE erzeugen Treiberroutinen auf temporären Dateien (es sind dies Members auf einer Objektmodul-Bibliothek). Die FILE-Anweisung dient zum permanenten Speichern der Treibermodule (als Lademodule). Durch FILE werden also die Treiberroutinen zu dynamisch aufrufbaren Moduln zusammengebunden. Die einfachste und normalerweise angewandte Form der Anweisung ist: FILE;. Es werden alle bis dahin seit der letzten FILE-Anweisung definierten STATEMENT- und CLAUSE-Routinen jede für sich zu einem Modul gebunden und in der Datei REGENT.PLSTRAN.MODS angelegt. Die FILE-Anweisung ermöglicht es aber auch, verschiedene Statement- und Clause-Treiberroutinen und schon fertig gebundene Module zu einem einzigen Modul zusammenzubinden. Dies ist dann effektiver, wenn die Routinen meist zusammen benutzt werden, da dann nur einmal der Modul von der Platte geladen werden muß.

Bedeutung der Optionen:

ALL: Jede bisher definierte Statement- oder Clause-Routine wird in je einem Modul permanent gespeichert. All ist default.

gruppe: Besteht eine "gruppe" aus einem Namen, wird die Definition dieses Namens zu einem Treibermodul gebunden, besteht die "gruppe" aus einer eingeklammerten Liste von Namen, werden die in der Liste aufgeführten Definitionen zusammen in einen Modul gebunden.

FILE
(2)

name: - dtype - rname - modulname	Name einer Statement oder Clause-Definition (Schlüsselwort: Name, default), einer Datentyp-Anweisungs-Definition (Schlüsselwort DATATYPE) oder eines Moduls, der sich bereits in der Datei REGENT.PLSTRAN.MODS befindet (Schlüsselwort MODUL). Namen jeweils in Apostrophe eingeschlossen.
ddname	DD-Name der Bibliothek, auf die die erzeugten Treibermodule gespeichert werden sollen. Normalerweise ist die Datei REGENT.PLSTRAN.MODS über den DD-Namen 'PLSLIB' ansprechbar, dies ist auch der Defaultwert für "ddname". Der Name muß in Apostrophe eingeschlossen sein.

Wird keine FILE-Anweisung in einem PLS-Lauf angetroffen, dann wird vor Ausführung der END PLS-Anweisung automatisch die Erzeugung der bis dahin definierten Treiberroutinen (STATEMENT AND CLAUSES) veranlaßt. Diese automatische Treibermodul-Generierung kann durch die Anweisung NOFLE; verhindert werden.

Beispiel

```
Eingabe:  'PATTERN ??? -.- X123.456///'

FIND ('PATT')
FIND ('?')
FIND ('-.-')                    } liefern '1'B
FIND ('///')
FIND ('X123.456')
```

FIND ('NIX') liefert '0'B, da die Zeichenkette 'NIX' fehlt.

FIND ('123') liefert '0'B, da '123' nicht am Anfang eines Elements steht. Das Element ist hier der Identifier 'X123'.

```
STA 'PUT';
    DCL C(3) CHAR(8) VARYING
        INIT ('DISPLAY', 'PLOT', 'MOVIE');
    DO I=1 TO 3;
        IF FIND(C(I)) THEN GO TO POL;
        END;
    PLI;
POL: ---
END STA;
```

Dieses PUT-Statement wird dann als PL/1-Statement interpretiert, wenn weder 'DISPLAY' noch 'PLOT' oder 'MODULE' darin vorkommen.

PLS-Funktion

Syntax

FIND (xyz)

Erläuterung

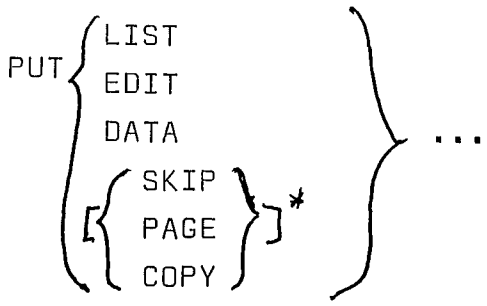
FIND ist eine logische Funktion, die den Wert '0'B oder '1'B zurückliefert. Sie liefert '1'B, falls in der behandelten POL-Anweisung bis zum Semikolon die Zeichen 'xyz' der Beginn eines Elements sind. Die Zeichenfolge 'xyz' wird also nur am Beginn einer Benennung, einer Konstanten, eines Operators oder eines Begrenzers wie ,,), (,; gesucht.
Wird die Zeichenfolge 'xyz' in der Anweisung nicht gefunden, wird '0'B zurückgeliefert. Die FIND-Funktion ist nur in STATEMENT- und CLAUSE-Definitionen gültig.

Beispiel

```
STA 'PUT';
    IF IDENTIFIER ('LIST') THEN DO;
        :
        :
        :
       END;
    ELSE IF ID('EDIT') THEN DO;
        :
        :
        :
       END;
    ELSE IF ID('DATA') THEN DO;
        :
        :
        :
       END;
    DCL C(3) CHAR(8) VARYING INIT
       ('SKIP','PAGE','COPY');
    DO WHILE (TYPE¬=10);
    DO I=1 TO 3;
       IF ID(C(I)) THEN DO;
        :
        :
        :
        :
       END;
       END;
END STA;
```

Erläuterung

Die Anweisungsdefinition arbeitet eine Anweisung ab mit der Syntax:



PLS-Funktion

Syntax

IDENTIFIER (xyz [,SKIP | NOSKIP])

Erläuterung

Die Funktion IDENTIFIER liefert einen logischen Wert (BIT(1)) zurück. Sie stellt fest, ob die Zeichen "xyz" die nächste Benennung (Identifier) in der Eingabe darstellen. Eine Benennung ist eine Folge von Zeichen, bestehend aus Buchstaben, Ziffern und den Zeichen $, #, @, _. Die Zeichenfolge muß mit einem Buchstaben oder #, $, @ beginnen. Benennungen oder Identifier sind also gültige PL/1-Variablennamen. Wird die Bennnung gefunden, wird '1'B zurückgeliefert und falls "SKIP" angegeben wurde, übergangen. Ist eine Benennung 'xyz' nicht in der Eingabe vorhanden, wird '0'B zurückgegeben, der Eingabezeiger wird nicht verändert.

Beispiel

```
INITIATE SUBSYSTEM 'GRA.PHIC'KEY'GRAKEY';
INIT 'S2' 'KEY2';
INI SUB 'SUB1' KEY 'S1KEY';
```

PLS-Anweisung

Syntax

INITIATE [SUBSYSTEM] name [[KEY] key];

Erläuterung

Die INITIATE-Anweisung dient dazu, ein neues Subsystem zu initialisieren. Es wird der Name des Subsystems und ein Schlüsselwort angegeben. Dieses Schlüsselwort muß stets angegeben werden, wenn das Subsystem erweitert, geändert oder gelöscht werden soll.

'name' ist eine Zeichenkette mit maximal 32 Zeichen, die den Namen des Subsystems darstellt. 'name' darf kein Leerzeichen enthalten, auch nicht am Ende. Soll der Name des Subsystems abkürzbar sein, ist nach den signifikanten ersten Buchstaben ein "." in den Namen einzufügen. Soll ein Punkt Bestandteil des Namens sein, müssen 2 Punkte geschrieben werden. Beispiel: 'NAM.E', das Subsystem heißt NAME, es kann zu NAM abgekürzt werden. 'S..1.22', das Subsystem heißt S.122 und kann S.1 abgekürzt werden.

'key' ist eine Zeichenkette von maximal 32 Zeichen und bezeichnet den Schlüssel für das Subsystem. 'key' darf keine Leerzeichen enthalten. Defaultwert für 'key', falls nicht angegeben, ist der Nullstring.

Ist ein Subsystem mit gleichem Namen und gleichem Schlüssel schon vorhanden, so wird zuerst das alte Subsystem zerstört und anschliessend neu initialisiert. Stimmt dagegen nur der Name und nicht der Schlüssel, erfolgt eine Fehlermeldung und keine Initialisierung.

Anmerkung: Die Zerstörung des alten Subsystems wird in der nächsten REGENT-Version unterbleiben.

Beispiel

```
IF ISEXPRESSION THEN A=NEXT_EX;
ELSE MESSAGE SEVERE TEXT ('EXPRESSION MISSING');
```

Eingabe:	Resultat von ISEX:
A+B	'1'B
A+B-	'0'B
A+B-C	'1'B
NAME_OF_VARIABLE	'1'B
./	'0'B
+3	'1'B

ISEXPRESSION

PLS-Funktion

Syntax

ISEXPRESSION

Erläuterung

ISEXPRESSION ist eine logische Funktion, die nur innerhalb STATEMENT- und CLAUSE-Definitionen gültig ist. Sie liefert '1'B falls als nächstes in der Eingabe ein arithmetischer, logischer oder Zeichenketten-Ausdruck steht. Sonst ist das Ergebnis '0'B.

Beispiel

Clause definiert durch:

```
CLAUSE 'WINKEL';
```

Aufruf:

```
LINK WINKEL;
```

LINK

PLS-Anweisung in Definitionen

Syntax

LINK cname;

Erläuterung

Die LINK-Anweisung dient zum Aufrufen von CLAUSES. 'cname' ist der in der CLAUSE-Anweisung definierte Name der Clause, 'cname' darf keine Leerzeichen enthalten und darf höchstens 32 Zeichen lang sein.

Beispiel

```
LIST MEMBERS OF SUBSYSTEM 'REMAC';
LIST MEMS SUB'SUB2';
LIST MEM 'SUB3';
```

LIST
MEMBERS

PLS-Anweisung

Syntax

LIST MEMBERS [OF] [SUBSYSTEM] subname;

Erläuterung

Alle zu dem Subsystem 'subname' gehörenden Partitioned-Dataset-Members, die sich auf den REGENT-Systembibliotheken befinden, werden namentlich in alphabetischer Reihenfolge aufgeführt. Es sind dies im einzelnen:

- auf dem PDS REGENT.MODS:
  PLR-Module

- auf dem PDS REGENT.PLSTRAN.MODS:
  Anweisungstreibermodule

- auf dem PDS REGENT.PLSTRAN.DATA:
  Subsystemtabellen und Datenstruktur-Definitionen.

Beispiel

```
LIST STATEMENTS OF SUBSYSTEM 'PLS';
LIST STATS 'GRAPHIC';
LIS STA 'SMAC';
```

LIST
STATEMENTS

PLS-Anweisung

Syntax

LIST STATEMENTS [OF] [SUBSYSTEM] subname;

Erläuterung

Die gültigen PL/1-Anweisungen, die POL-Anweisungen und die Datenstrukturen eines Subsystems mit dem Namen 'subname' werden namentlich aufgeführt.

Format der Liste:

```
**************************************
*                                    *
*      SUBSYSTEM - INFORMATION       *
*                                    *
**************************************
```

NAME OF SUBSYSTEM	PREFIX	CREATION DATE	SUBSYSTEM LIST	ABB-REV. ?	NUMBER OF STATEMENTS	NUMBER OF CLAUSES	NUMBER OF DATA-DECL.
PLS	PL	01.05.73	PLPLS	NO	79	0	2

NAME OF STATEMENT	CHARACTERISTIC	MODUL
%AS	PL/1-STATEMENT	
ASS	PL/1-STATEMENT	
%ACT	PL/1-STATEMENT,ALIAS	
%ACTIVATE	PL/1-STATEMENT	
%CONTROL	PL/1-STATEMENT	
%DCL	PL/1-STATEMENT,ALIAS	
%DEACT	PL/1-STATEMENT,ALIAS	
%DEACTIVATE	PL/1-STATEMENT	
%DECLARE	PL/1-STATEMENT	
%DO	PL/1-STATEMENT	
%END	PL/1-STATEMENT	
%GO	PL/1-STATEMENT	
%GOTO	PL/1-STATEMENT,ALIAS	
%IF	PL/1-STATEMENT	
%INCLUDE	PL/1-STATEMENT	
%PAGE	PL/1-STATEMENT	
%SKIP	PL/1-STATEMENT	
ALLOC	PL/1-STATEMENT,ALIAS	
ALLOCATE	PL/1-STATEMENT	
BEGIN	PL/1-STATEMENT	
CALL	PL/1-STATEMENT	
CHECK	PL/1-STATEMENT	
CLAUSE	POL-STATEMENT,ABBREV.	PLCLAU
CLOSE	PL/1-STATEMENT	

LIST
STATEMENTS
(2)

Bedeutung der Spalten:

Der Kopf der Liste besteht aus den Angaben, die das betreffende Subsystem beschreiben, siehe Anweisung LIST SUBSYSTEMS.

NAME OF STATEMENT — Name der POL- oder PL/1-Anweisung oder der Datendefinition. Die Reihenfolge der Liste ist:

- Preprocessorzuweisung und PL/1-Zuweisung (%AS und ASS)
- Datentyp-POL-Anweisungen, falls definiert (REAL,INIT,etc)
- PL/1-Preprocessor-Anweisungen
- PL/1- und POL-Anweisungen

Sind die Namen abkürzbar, so sind die signifikanten Buchstaben unterstrichen.

CHARACTERISTIC — beschreibt die Art der Anweisungen.

MODUL — Name des zugehöirigen Members in einere PLS-Bibliothek. Bei POL-Anweisungen und Systemanweisungen ist dies der Name des Treiberroutinen-Moduls. Bei Datendefinitionen ist es der Membername in der PLS-Databibliothek (REGENT.PLSTRAN.DATA).

Beispiel

```
LIST SUBSYSTEMS;

LIST SUBS;

LIST SUB;
```

LIST
SUBSYSTEMS

Syntax

LIST SUBSYSTEMS;

Erläuterung

Alle im System vorhandenen REGENT-Subsysteme werden namentlich aufgeführt.

Format der Subsystem-Liste:

LIST OF REGENT-SUBSYSTEMS

NAME OF SUBSYSTEM	PRE FIX	CREATION DATE	SUB-SYSTEM LIST	ABB-REV. ?	NUMBER OF STATE-MENTS	NUMBER OF CLAU-SES	NUMBER OF DATA-DECL.
PLS	PL	01.05.73	PLPLS	NO	70	0	2
REGENT	QQ	01.05.73	PLREG	NO	61	0	1
TEST	TE	27.06.74	PLTEST	NO	65	1	1
VEKTOR	VE	27.06.74	PLVEKT	YES	66	1	1

Bedeutung der Spalten

NAME OF SUBSYSTEM — Subsystemname in voller Länge; wenn der Name abkürzbar ist, sind die signifikanten Zeichen unterstrichen.

PREFIX — Subsystem-Prefix. Dieser aus zwei Zeichen bestehende Prefix wird benutzt, um die Treibermodule, die Datenstrukturen und die von PLS verwalteten Listen, die zu einem Subsystem gehören, eindeutig zu kennzeichnen. Die Namen aller zu einem Subsystem gehörenden Members in den PLS-Bibliotheken fangen mit dem Prefix an. Auch die Namen der PLR-Module beginnen mit dem Subsystem-Prefix.

11-78

LIST
SUBSYSTEMS
(2)

CREATION DATE	Datum der Initialisierung des Subsystems
SUBSYSTEM LIST	Name der Tabelle der POL-Anweisungen und Datenstrukturen des Subsystems in der Bibliothek REGENT.PLSTRAN.DATA
ABBREV.?	YES oder NO, je nachdem ob der Subsystemname abkürzbar ist. Bei YES sind die signifikanten Buchstaben des Namens unterstrichen.
NUMBER OF STATEMENTS	Anzahl der POL und PL/1-Anweisungen, die zum Subsystem gehören, die Anzahl ist 61.
NUMBER OF CLAUSES	Anzahl der Clauses des Subsystems.
NUMBER OF DATA-DECLARATIONS	Anzahl der Datenstruktur-Definitionen.

Beispiel

```
   LPARMS 'LIST,RENT,MAP,XREF';
DCL LINPARAMETER CHAR(13) INIT('LIST,MAP,REUS,');
   LPAR LINPARAMETER || 'SIZE(100K,32K)';
```

PLS-Anweisung

Syntax

LPARMS expression;

Erläuterung

Die FILE-Routine benutzt zur Erzeugung von Treibermoduln aus STATEMENT- und CLAUSE-Definitionen den OS/360 Linkage Editor. Mit der Anweisung LPARMS können die Linkage-Editor-Anweisungen geändert werden. 'expression' muß ein Zeichenketten-Ausdruck von maximaler Länge 95 sein. Defaultwert für die Parameter:
'REUS,SIZE=(108K,26K),DCBS'.

Beispiel

```
MACROTIME DATASTRUCTURE;
DCL 1 SUBMAC BASED,
      2 NAME CHAR(8) INIT(''),
      2 LISTKOPF PTR INIT(NULL()),
      2 LIST_LAENGE BIN FIXED(15) INIT(0),
      2 NEXT_ELEMENT BIN FIXED(15) INIT(1);
END MACROTIME DATASTRUCTURE;
MAC;
DCL 1 WERTE,
      2 X(1000) CHAR(1) INIT((1000)(1)'');
END MAC;
MAC DAT;
DCL 1 POINTERS BASED,
      2 P(12) PTR INIT((12)NULL());
END MAC;
```

Erläuterung

Nach BASED darf keine POINTER-Variable angegeben werden. Wird das BASED-Attribut weggelassen, wird es von PLS eingefügt.

MACROTIME
DATASTRUCTURE

PLS-Anweisung

Syntax

MACROTIME DATASTRUCTURE;

Erläuterung

Die Anweisungs-Treiber-Routinen, die dazu dienen, eine bestimmte Anweisung zu expandieren, sind als unabhängige Module in einer Bibliothek gespeichert und werden bei Bedarf dynamisch in den Arbeitsspeicher geladen. Sie können daher untereinander nicht über globale interne oder auch externe Variable kommunizieren. Mit der Anweisung "MACROTIME DATASTRUCTURE" wird eine Übersetzungszeit-Datenstruktur deklariert, die den Treiberroutinen eine Kommunikation untereinander ermöglicht. Die hier deklarierten Variablen sind den globalen Makrozeit-Variablen der OS/360-Assembler Macrolanguage oder in PL/1-Macrotime-Procedures deklarierten Variablen vergleichbar. Die Übersetzungs-Zeit-Datenstruktur ist eine BASED PL/1-Struktur mit festen Längen und Dimensionen. Sie wird bei der Übersetzung der ENTER-Anweisung angelegt. Ein Pointer auf die Struktur wird an alle Treiberroutinen übergeben, so daß auf die globalen Übersetzungszeitvariablen in allen Routinen zugegriffen werden kann.
Sollen Variable initialisiert werden, so muß dies durch das INITIAL-Attribut in der Deklaration erfolgen.
Zur Übersetzungszeit können zum Abspeichern von Werten verkettete Listen (linked lists) verwendet werden. Der Listenkopf muß dann in der Übersetzungszeit-Datenstruktur gespeichert sein, damit alle Treiberroutinen auf die Listen zugreifen können. Mit Hilfe von verketteten Listen können so auch zur Übersetzungszeit Daten mit variablem Speicherplatzbedarf verwendet werden.

Wie die Subsystem-Datenstrukturen muß auch die Übersetzungszeit-Datenstruktur deklariert werden, bevor sie in einer STATEMENT- oder CLAUSE-Definition benutzt wird. Erweiterungen am Ende der Struktur können vorgenommen werden, ohne daß alle Treiberroutinen neu übersetzt werden müssen. Wird jedoch die Übersetzungszeit-Datenstruktur abgeändert, so müssen alle STATEMENT- und CLAUSE-Definitionen wiederholt werden, die auf die Struktur zugreifen.

Beispiel

```
STATEMENT 'MESSTEST';
   DCL TEXT CHAR(100) VAR,
       I PIC '999' INIT(0),
       J PIC '99';
   DO WHILE(TYPE¬=10);
       I=I+1;
       TEXT=NEXT_STRING;
       J=NEXT_INTEGER;
       TEXT=SUBSTR(TEXT,2,LENGTH(TEXT)-2);
       /* APOSTROPHE  WEGNEHMEN */
       IF J=0 THEN
       MESSAGE I TEXT ('TEST'||I||TEXT);
       ELSE IF J=4 THEN
       MESS W TEXT ('TEST'||I||TEXT);
       ELSE IF J=8 THEN
       MESSAGE ERROR TEXT ('TEST'||I||TEXT);
       ELSE MESS S TEXT ('TEST'||I||TEXT);
       END;
END STATEMENT;
```

Erläuterung:

Folgendes Statement wird definiert:

MESSTEST 'text' { 0 / 4 / 8 / 12 } ,

Durch seine Anwendung wird bei der REGENT-Übersetzung (im P-Step) die Nachricht:

TEST { 0 / 4 / 8 / 12 } text

auf den Druck-Ausgabe-File ausgedruckt.

MESSAGE
in Definitionen

PLS-Anweisung

Syntax

```
MESSAGE [level] [text] ;

level::=   D |DEBUG
         | I |INFORMATIVE
         | W |WARNING
         | E |ERROR
         | S |SEVERE
         | T |TERMINAL

 text::=   TEXT (expression)
 Standardwerte:  I,TEXT('')
```

Erläuterung

Während sich MESSAGE-Anweisungen in PLR-Programmen und die Systemanweisung MESSAGE in POL-Programmen ebenso wie EXEC MESSAGE in Definitionen zur Ausführungszeit eines Subsystems auswirken, dient die MESSAGE-Anweisung in STATEMENT- oder CLAUSE-Definitionen der Erzeugung von Nachrichten zur Übersetzungszeit der Subsystem-POL. Im übrigen ist die Syntax der Anweisung die gleiche wie bei den anderen MESSAGE-Statements. (Siehe Kap.6.8, Seite 6-41 bis 6-51).

Beispiel

```
NEXT_BITSTRING(SKIP)
NEXT_BIT(NOSKIP)
NEXT_BITS
```

Eingabe:	NEXT_BIT liefert:
'0'B&'1'B	'0'B
'1011101'B	'1011101'B
'123111'B	''B und Fehlermeldung, da kein Bitstring
'1010101'+3	''B und Fehlermeldung, da die Zeichenkette keine Bitkette ist, B fehlt.
'101 101'B	''B und Fehlermeldung, da kein Bitstring wegen Leerzeichen
(15)'0'B	(15)'0'B Wiederholungsfaktoren werden erkannt.

NEXT_BITSTRING

PLS-Funktion

Syntax

NEXT_BITSTRING [(SKIP | NOSKIP)]

Erläuterung

Die Funktion NEXT_BITSTRING ist nur in STATEMENT- oder CLAUSE-Definitionen gültig. Sie liefert eine Zeichenkette variabler Länge zurück (CHARACTER(250) VARYING), die aus der nächsten in der Eingabe vorkommender Bitkette besteht. Ist eine Bitkette nicht vorhanden, erfolgt eine Fehlermeldung, die Funktion liefert ''B zurück. Die Bitkette, die geliefert wird, enthält die Apostrophe und das B.

Bei SKIP wird der Eingabezeiger hinter die Bitkette gesetzt, bei NOSKIP bleibt der Eingabezeiger unverändert stehen und der Bitstring kann noch einmal gelesen werden.

Beispiel

```
NEXT_EXPRESSION(SKIP)
NEXT_EX(NOSKIP)
NEXT_EX
```

Eingabe:	NEXT_EX liefert:
3↓4-N;	3↓4-N
A A1 A2	A
3 4 -5	3
4 -5 3	4 -5
X↓A×C(5);	X↓A×C(5)
'A'||(3)'B'?	'A'||(3)'B'
X&Y&¬'110'B	X&Y&¬'110'B
4↓7-;	Fehler, Ausdrücke müssen in sich abgeschlossen sein
(3↓A A	Fehler
A&¬↓-↓1=5;	A&¬↓-↓1=5 gültiger PL/1-Ausdruck

NEXT_EXPRESSION

PLS-Funktion

Syntax

NEXT_EXPRESSION [(SKIP | NOSKIP)]

Erläuterung

Die Funktion NEXT_EXPRESSION ist nur in STATEMENT- oder CLAUSE-Definitionen gültig. Sie liefert eine Zeichenkette variabler Länge zurück (CHARACTER(250) VARYING), die aus dem nächsten in der Eingabe vorkommenden logischen, arithmetischen oder Zeichenketten-Ausdruck besteht. Ist ein Ausdruck nicht vorhanden, erfolgt eine Fehlermeldung, die Funktion liefert " zurück.

Bei SKIP wird der Eingabezeiger hinter den Ausdruck gesetzt, bei NOSKIP bleibt der Eingabezeiger unverändert stehen, und der Ausdruck kann noch einmal gelesen werden.

Beispiel

```
NEXT_IDENTIFIER(NOSKIP)
NEXT_IDENT(SKIP)
NEXT_ID
```

Eingabe:	NEXT_ID liefert:
A+B×C	A
A.B.C	A
A(5) B	A
#Ø_123+4	#Ø_123
'X'	Fehler, keine Benennung
ABCDEFGHIJK+B	ABCDEFGHIJK
!A	Fehler

Fehlermöglichkeit

```
DCL I BIN FIXED(15) INIT(1);
EXEC NEXT_ID(I)=NEXT_EX;;
```

Erwartet wird, daß bei Eingabe steht:

```
X  3.14
```

erzeugt werden soll:

```
X(1)=3.14;
```

Jedoch bringt PLS die Fehlernachricht
PLS 182, IN PLS-FUNKTION NEXT_ID ARGUMENTS MISSPELLED. Der Grund ist, daß PLS nach NEXT_ID (entweder SKIP oder NOSKIP erwartet (statt I).

Abhilfe:

```
EXEC NEXT_ID(SKIP)(I)=NEXT_EX;;
```

NEXT_IDENTIFIER

PLS-Funktion

Syntax

NEXT_IDENTIFIER [(SKIP | NOSKIP)]

Erläuterung

Die Funktion NEXT_IDENTIFIER ist nur in STATEMENT- oder CLAUSE-Definitionen gültig. Sie liefert eine Zeichenkette variabler Länge zurück (CHARACTER(250) VARYING), die aus der nächsten in der Eingabe vorkommenden Benennung (Identifier) besteht. Ist eine Benennung nicht vorhanden, erfolgt eine Fehlermeldung, die Funktion liefert '' zurück. Für die Benennungen gelten die gleichen Syntaxregeln wie für Namen in PL/1.

Bei SKIP wird der Eingabezeiger hinter den Identifier gesetzt, bei NOSKIP bleibt der Eingabezeiger unverändert stehen, und die Benennung kann noch einmal gelesen werden.

Beispiel

```
NEXT_ITEM (I,SKIP)
NEXT_ITEM (J,NOSKIP)
NEXT_ITEM (K)
```

Eingabe:	NEXT_ITEM liefert (Zeichen,Typ):
A1+B2	A1,5
33×48	33,1
1.68E-8I+1	1.68E-8I,6
0.0010B	0.0010B,2
8.3-2.5	8.3,2
101BBB	101B,1
'XYZ''W' A	'XYZ''W',3
(18)' ';	(18)' '
'010101'B	'010101'B,4
'010101'X	'010101',3
'020101'B	'020101',3
NAME.A1	NAME,5
8I.J	8I,6
¬=3	¬=,7
>>	>,7
×××××	××,7
(3+4)	(,8
)-	),9
;	;,10
+23-24	+23,11
-110B+2	-110B,11
+101.1B;	+101.1B,12
-1.01E-5B×3	-1.01E-5B,12
,3,4	,,13
%%%	%,13
!?!	!,14
End of file bei SYSIN	,15

NEXT_ITEM

PLS-Funktion

Syntax

NEXT_ITEM (type [,SKIP | NOSKIP])

Erläuterung

Die Funktion NEXT_ITEM ist nur in STATEMENT- oder CLAUSE-Definitionen gültig. Sie liefert eine Zeichenkette variabler Länge zurück (CHARACTER(250) VARYING), die aus dem nächsten in der Eingabe vorkommenden Element besteht. Der Typ des Elements wird in der BIN FIXED(15)-Variablen 'type' zurückgeliefert, die Werte haben die gleiche Bedeutung wie bei der PLS-Funktion "TYPE".

Bei SKIP wird der Eingabezeiger hinter das Element gesetzt, bei NOSKIP bleibt der Eingabezeiger unverändert stehen, und das Element kann noch einmal gelesen werden.

Beispiel

```
NEXT_N (I,NOSKIP)
NEXT_N (10,SKIP)
NEXT_N (80)
```

Eingabe:

```
STRING HAT LAENGE 80 UND HEISST
'      1       2      3      4      5
       6       7      8      ';
```

NEXT_N(80) liefert:

```
 STRING HAT LAENGE 80 UND HEISST
'      1       2      3      4      5
       6       7
```

Danach steht der Eingabezeiger zwischen 7 und 8, also mitten in einer Zeichenkette. Falls nun nicht der Rest der Zeichenkette bis zum Apostroph auch noch mit NEXT_N abgearbeitet wird, gerät der Übersetzer "außer Tritt" und interpretiert alle Nicht-Strings als Strings und umgekehrt. NEXT_N ist also nur in Sonderfällen sinnvoll, z.B. für das Abarbeiten formatierter Eingabe.

NEXT_N

PLS-Funktion

Syntax

NEXT_N (n [,SKIP | NOSKIP])

Erläuterung

Die Funktion NEXT_N ist nur in STATEMENT- oder CLAUSE-Definitionen gültig. Sie liefert eine Zeichenkette variabler Länge zurück (CHARACTER(250) VARYING), die aus den nächsten in der Eingabe vorkommenden n Zeichen besteht. 'n' ist BIN FIXED(15) und muß 250 sein. Sind bis zum Ende der Eingabedatei weniger als n Zeichen vorhanden, erfolgt eine Fehlermeldung, es werden dann nur die vorhandenen Zeichen geliefert.

Bei SKIP wird der Eingabezeiger hinter die n Zeichen gesetzt, bei NOSKIP bleibt der Eingabezeiger unverändert stehen, und die Eingabe kann noch einmal gelesen werden.

Beispiel

```
NEXT_OPERATOR(NOSKIP)
NEXT_OPERATOR(SKIP)
NEXT_OP
```

Eingabe:	NEXT_OP liefert:
+-+	+
×××××	××
----	-
-5	-
+16-20	+
||||	||
>=	>=
-&-	-
|&	|
¬=	¬=
×/	×
ABC	Fehler, kein Operator
??	" " "
1-2	" " "
(())	" " "
,,;	" " "

NEXT_OPERATOR

PLS-Funktion

Syntax

NEXT_OPERATOR [(SKIP | NOSKIP)]

Erläuterung

Die Funktion NEXT_OPERATOR ist nur in STATEMENT- oder CLAUSE-Definitionen gültig. Sie liefert eine Zeichenkette variabler Länge zurück (CHARACTER(250) VARYING), die aus dem nächsten in der Eingabe vorkommenden Operator besteht. Ist ein Operator nicht vorhanden, erfolgt eine Fehlermeldung, die Funktion liefert '' zurück.

Bei SKIP wird der Eingabezeiger hinter den Operator gesetzt, bei NOSKIP bleibt der Eingabezeiger unverändert stehen, und der Operator kann noch einmal gelesen werden.

Beispiel

```
NEXT_REAL(NOSKIP)
NEXT_REAL(SKI)
NEXT_REAL
```

Eingabe:	NEXT_REAL liefert:
1.0-2.0	1.0
11+22	11
-8+9	-8
+28.3-2.0	+28.3
1.01B-3	1.01B
101B	101B
2.01B-3	2.01
1.01E-3B;	1.01E-3B
1.02E-3B,	1.02E-3
1.789E+28-2.5	1.789E+28
X-2	0,Fehler
'21'	0,Fehler
'101'B	0,Fehler

NEXT_REAL

PLS-Funktion

Syntax

NEXT_REAL [(SKIP | NOSKIP)]

Erläuterung

Die Funktion NEXT_REAL ist nur in STATEMENT- oder CLAUSE-Definitionen gültig. Sie liefert eine Zeichenkette variabler Länge zurück (CHARACTER(250) VARYING), die aus der nächsten in der Eingabe vorkommenden DECIMAL oder BINARY FLOAT-Konstanten besteht. Ist eine Real-Konstante nicht vorhanden, erfolgt eine Fehlermeldung, die Funktion liefert 0 zurück. Die Real-Konstante kann ein Vorzeichen + oder - besitzen.

Bei SKIP wird der Eingabezeiger hinter die Realzahl gesetzt, bei NOSKIP bleibt der Eingabezeiger unverändert stehen, und die Eingabe kann noch einmal gelesen werden.

Beispiel

```
NEXT_STRING(SKIP)
NEXT_STR(NOSKIP)
NEXT_STRING
```

Eingabe:	NEXT_STRING liefert:
'XYZ';	'XYZ'
'AB1''23',	'AB1''23'
(10)'S' 'T'	(10)'S'
'10101'B	'', Fehler, da Bitkette
'10102'B	'10102'
'10101'X	'10101'

NEXT_STRING

PLS-Funktion

Syntax

NEXT_STRING [(SKIP | NOSKIP)]

Erläuterung

Die Funktion NEXT_STRING ist nur in STATEMENT- oder CLAUSE-Definitionen gültig. Sie liefert eine Zeichenkette variabler Länge zurück (CHARACTER(250) VARYING), die aus der nächsten in der Eingabe vorkommenden Zeichenkette besteht. Ist eine Zeichenkette nicht vorhanden, erfolgt eine Fehlermeldung, die Funktion liefert '' zurück. Die die Zeichenkette umschließenden Apostrophe werden mit zurückgegeben.

Bei SKIP wird der Eingabezeiger hinter die Zeichenkette gesetzt, bei NOSKIP bleibt der Eingabezeiger unverändert stehen, und der String kann noch einmal gelesen werden.

Beispiel

```
NEXT_WORD(SKIP)
NEXT_WORD(NOSKIP)
NEXT_WORD
```

Eingabe:	NEXT_WORD liefert:
3 4 5	3
AX+B×3;	AX+B×3
AX + B	AX
+++++A++++;	+++++A++++
;	; und Fehler

NEXT_WORD

PLS-Funktion

Syntax

NEXT_WORD [(SKIP | NOSKIP)]

Erläuterung

Die Funktion NEXT_WORD ist nur in STATEMENT- oder CLAUSE-Definitionen gültig. Sie liefert eine Zeichenkette variabler Länge zurück (CHARACTER(250) VARYING), die aus den nächsten in der Eingabe vorkommenden Zeichen bis zum nächsten Leerzeichen oder Semikolon besteht. Ist das nächste Zeichen in der Eingabe ein Semikolon, so wird ein Semikolon zurückgeliefert, gleichzeitig erfolgt eine Fehlermeldung.

Bei SKIP wird der Eingabezeiger hinter die gelieferten Zeichen gesetzt, bei NOSKIP bleibt der Eingabezeiger unverändert stehen, und die Zeichen können noch einmal gelesen werden.

Beispiel

```
NOCOMPRESS;

NOCO;
```

NOCOMPRESS

PLS-Anweisung

Syntax

NOCOMPRESS;

Erläuterung

Die Systembibliotheken REGENT.PLSTRAN.MODS und REGENT.PLSTRAN.DATA sind Partitionen Data Sets (PDS) des OS/360. Diese Bibliotheken enthalten die Anweisungs-Treiber-Module, Tabellen und Datenstruktur-Deklarationen. Während der Subsystem-Entwicklung müssen sie von Zeit zu Zeit komprimiert werden. Dies geschieht am Ende des Subsystems PLS immer dann, wenn weniger als 10% freier Platz auf den Bibliotheken ist. Die Anweisung NOCOMPRESS verhindert dagegen das Komprimieren auf jeden Fall.

Anmerkung: Wenn das Betriebssystem den dynamischen Aufruf von IEBCOPY nicht erlaubt, so ist das Komprimieren in einem eigenen Step durch // EXEC PGM=IEBCOPY auszuführen.

Beispiele:

```
NOFILE;
```

Erläuterung:

Die definierten Statements and Clauses werden nicht auf die Bibliothek geladen.

PLS-Anweisung

Syntax

NOFILE;

Erläuterung:

Wurden in einer Anwendung des Subsystems PLS STATEMENTS und / oder Clauses definiert und steht nirgends die FILE;-Anweisung, so generiert PLS automatisch eine FILE;-Anweisung vor der END PLS;-Anweisung. Soll dieses automatische File zu Testzwecken unterbleiben, kann dies durch die Anweisung NOFILE; erreicht werden.

Beispiel

```
CALL QQNEW_STATEMENT;
CALL PL_PREFIXES(P);
CALL PL_LABELS(L);
IF PL_ASSTEST THEN PUT LIST('ASSIGNMENT');
   ELSE PUT LIST('NO ASSIGNMENT');
```

PL_ASSTEST

PLS-Funktion

Syntax

PL_ASSTEST

Erläuterung

PL_ASSTEST ist eine logische Funktion, die feststellt, ob eine Anweisung eine Zuweisung ist. Sie wird dann benötigt, wenn eine STATEMENT-Definition mehrere POL-Anweisungen abarbeiten soll. Sie muß am Anfang einer Anweisung, nach Aufruf von QQNEW_STATEMENT, PL_PREFIXES und PL_LABELS aufgerufen werden.
PL_ASSTEST besitzt keine Argumente und liefert als Funktionswert einen BIT(1)-Wert. Ist die Anweisung eine Zuweisung, ist der Funktionswert '1'B, im anderen Fall '0'B.

Beispiel

```
DCL (P,L) CHAR(250) VARYING;
CALL QQNEW_STATEMENT;
CALL PL_PREFIXES (P);
CALL PL_LABELS (L);
```

Die Labels der Anweisung stehen jetzt in L.

PL_LABELS

PLS-Funktion

Syntax

```
CALL PL_LABELS (arg);
```

Erläuterung

PL_LABELS ist eine Prozedur mit einem CHARACTER(*) VARYING-Argument, die die vor einer Anweisung stehenden Marken (Labels) gewinnen kann. Alle Labels einschließlich aller Doppelpunkte werden in dem einen CHAR(*)VARYING Argument der Prozedur zurückgegeben, sind keine Labels vorhanden, wird der Nullstring geliefert. Die Prozedur PL_LABELS wird benötigt, wenn eine STATEMENT-Definition mehrere POL-Anweisungen abarbeiten soll. Sie muß am Anfang einer Anweisung nach dem Aufruf von QQNEW_STATEMENT und PL_PREFIXES aufgerufen werden.

Beispiel

```
DCL P CHAR(250) VARYING;
CALL QQNEW_STATEMENT;
CALL PL_PREFIXES (P);
```

Die Condition Prefixes der Anweisung stehen jetzt in P.

PL_PREFIXES

PLS-Funktion

Syntax

```
CALL PL_PREFIXES (arg);
```

Erläuterung

PL_PREFIXES ist eine Prozedur mit einem CHARACTER(*) VARYING-Argument, die die vor einer Anweisung stehenden Präfixe (Condition prefixes) abarbeitet. Die Präfixe einschließlich aller Klammern und Doppelpunkte werden in dem Argument zurückgegeben. Sind keine Präfixe vorhanden, wird der Nullstring zurückgegeben. Die Prozedur PL_PREFIXES wird benötigt, wenn eine STATEMENT-Definition mehrere POL-Anweisungen abarbeiten soll. Sie muß am Anfang einer Anweisung nach dem Aufruf von QQNEW_STATEMENT aufgerufen werden.

Beispiel

Die folgenden Anweisungsdefinitionen dienen dazu, die Anzahl der PUT-Anweisungen in einem Programm zu zählen und auf Verlangen auszudrucken.

```
MAC DATA;
DCL 1 ANZAHL ,
      2 PUT BIN FIXED(15) INIT(0);
END MAC DAT;
STAT 'PUT'; /* Definiere PUT-Anweisung */
PUT = PUT+1; /* Zähle PUT-Anweisung */
PLI;         /* Behandle Anweisung als PL/1-Anweisung */
END STAT;
STA 'DRU.CKE'; /* Drucke Anzahl */
PUT LIST(PUT) SKIP;
END STA;
```

Ausgabe der gerade behandelten Anweisung ohne RETURN:

```
STATEMENT 'PUT' LABELS TO L PREFIXES TO P;
 .
 .
 .
EXEC P||L||THIS_STATEMENT;
```

PLS-Anweisung in Definitionen

Syntax

PLI;

Erläuterung

Diese Anweisung teilt dem PLS-Übersetzer mit, daß er die Anweisung als System- oder PL/1-Anweisung behandeln soll. Man kann PL/1-Anweisungen auf bestimmte Eigenschaften und Optionen untersuchen, indem man eine POL-Anweisung gleichen Namens definiert. Soll der Übersetzer die betrachtete Anweisung als System- oder PL/1-Anweisung interpretieren, kann man dies durch die PLI-Anweisung erreichen. PLI impliziert ein RETURN-Statement. Ist dieses nicht erwünscht, so kann der Anwender das aktuelle Statement mit Hilfe der Funktionen PL_PREFIXES, PL_LABELS, THIS_STATEMENT verarbeiten.

Beispiel

In der Eingabe stehen nach der Anweisung DATEN Tripel von Werten, die jeweils auf die 3 Files QQFILE1 bis 3 verteilt werden:

```
DATEN 1, 2, 3, 7, 9, 11, 5, 8, 9, ...;
```

```
STA 'DATEN';
    OPEN FILE(QQFILE1) OUTPUT STREAM TITLE('SUB11'),
         FILE(QQFILE2) OUTPUT STREAM TITLE('SUB12'),
         FILE(QQFILE3) OUTPUT STREAM TITLE('SUB13');
    DO WHILE (TYPE¬=10&TYPE¬=15);
         X=NEXT_REAL; SKIP(',');
         Y=NEXT_REAL; SKIP(',');
         Z=NEXT_REAL; SKIP(',');
         PUT FILE(QQFILE1) LIST(X);
         PUT FILE(QQFILE2) LIST(Y);
         PUT FILE(QQFILE3) LIST(Z);
    END;
    CLOSE FILE(QQFILE1), FILE(QQFILE2), FILE(QQFILE3);
END STA;
```

Einlesen der Felder X,Y,Z zur Ausführungszeit:

```
GET FILE(SUB11) LIST(X);
GET FILE(SUB12) LIST(Y);
GET FILE(SUB13) LIST(Z);
```

QQFILE1
QQFILE2
QQFILE3

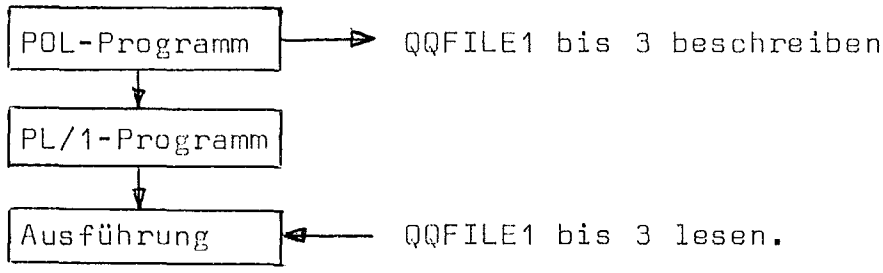
PLS-Variable in Definitionen

Syntax

QQFILE1
QQFILE2
QQFILE3

Erläuterung

QQFILE1 bis 3 sind drei FILE-Variable, die dazu dienen können, Daten von der Übersetzungszeit an die Ausführungszeit zu übergeben.



Die FILES sind ohne Attribute deklariert, alle Attribute können im OPEN angegeben werden. Um Verwechslungen bei gleichzeitiger Anwendung durch verschiedene Subsysteme zu vermeiden, sollten die Files mit TITLE eröffnet werden. Die DD-Namen sollten mit den ersten 7 Zeichen des Subsystemnamens beginnen.

Beispiel

```
I=QQLEFT_MARGIN;

ANWEISUNGS_LAENGE = QQRIGHT_MARGIN

   -QQLEFT_MARGIN+1;
```

QQLEFT_MARGIN

PLS-Variable in Definitionen

Syntax

QQLEFT_MARGIN

Erläuterung

Die Variable QQLEFT_MARGIN enthält den gültigen Wert des Compiler-Parameter LEFTMARGIN, also die Spaltenposition in den Eingaberecords, auf der der gültige Teil einer Anweisung beginnt. Dieser Wert ist notwendig bei der satzweisen (record-weisen) Verarbeitung formatgebundener Eingabe. Normalerweise hat QQLEFT_MARGIN den Wert 2.

Beispiel

```
STA 'RECORDS';
    OPEN FILE(QQFILE1) OUTPUT RECORD TITLE ('SUB11');
    BEGIN;
       DCL RECORD CHAR(QQLRECL) INIT('');
       DO WHILE(RECORD¬ ='××EOF'&TYPE¬ =15);
          RECORD=QQTHIS_RECORD('1'B);
          WRITE FILE(QQFILE1) FROM(RECORD);
          END;
    END;
    CLOSE FILE(QQFILE1);
END STA;
```

Erläuterung

Die RECORDS-Anweisung liest alle folgenden Sätze der Eingabe und schreibt sie auf den Kommunikations-FILE QQFILE1, solange bis der Satz '××EOF' die Verarbeitung beendet.

QQLRECL

PLS-Variable in Definitionen

Syntax

QQLRECL

Erläuterung

Die Variable QQLRECL enthält den jeweils gültigen Wert der Satzlänge (Logical Record Length, DCB-Parameter auf der DD-Karte LRECL, ENVIRONMENT-Parameter in der FILE-Deklaration RECSIZE) für die Eingabedatei, über die das POL-Programm eingelesen wird. Dieser Wert ist von Nutzen bei der satzweisen (record-weisen) Verarbeitung formatgebundener Eingabe. Für Karteneingabe hat QQLRECL den Wert 80.

Beispiel

```
NEXT_STATEMENT:
  DO WHILE(TYPE¬=10&TYPE¬=15);
     /* VERARBEITE 1 ANWEISUNG */
     END;
  CALL QQNEW_STATEMENT;
  GOTO NEXT_STATEMENT;
```

QQNEW_STATEMENT

PLS-Funktion

Syntax

```
CALL QQNEW_STATEMENT;
```

Erläuterung

QQNEW_STATEMENT ist eine Routine, die benötigt wird, wenn eine STATEMENT-Definition mehrere POL-Anweisungen abarbeiten soll. Wenn eine Anweisung bis zum Semikolon abgearbeitet wurde, muß QQNEW_STATEMENT aufgerufen werden, um die nächste Anweisung im Eingabepuffer zur Verfügung zu stellen. Wird der Aufruf vergessen, führt dies zur Fehlermeldung 'STATEMENT TOO LONG'.

Beispiel

Eingaberecord:

```
Position:  12345678901234567890123456789012345...
                        1         2         3

Text:          POS ID  'TEXT' , 12  4 2.3 ;
```

Anweisungsdefinition:

```
STA 'POS';
  DO WHILE(TYPE¬=10&TYPE¬=15);
     PUT LIST(QQNEXTPOS);
     SKIP;
     END;
END STA;
```

Die ausgedruckten Werte von QQNEXTPOS werden nacheinander
7, 11, 18, 20, 24, 26, 30 sein.

QQNEXTPOS

PLS-Variable in Definitionen

Syntax

QQNEXTPOS

Erläuterung

Die Variable QQNEXTPOS enthält jeweils die Spaltenposition des nächsten in der Eingabe enthaltenen Elements. Dieser Wert ist von Nutzen bei der Verarbeitung formatgebundener Eingabe.

Beispiel

```
READ:
    STAT=THIS_STATEMENT(SKIP);
    PUT FILE(QQFILE2) EDIT(STAT)
       (COLUMN(QQLEFT_MARGIN+QQRELPOS),A);
    GOTO READ;
```

QQRELPOS

PLS-Variable in Definitionen

Syntax

QQRELPOS

Erläuterung

Die Variable QQRELPOS enthält die Spalten-Position des Anfangs der gerade betrachteten Anweisung, bezogen auf den linken Rand QQLEFT_MARGIN. Die Kartenspalte des Statementanfangs ist QQLEFT_MARGIN + QQRELPOS. Der Wert von QQRELPOS ist von Nutzen bei der Verarbeitung formatgebundener Anweisungen.

Beispiel

```
I=QQRIGHT_MARGIN;

ANWEISUNGS_LAENGE = QQRIGHT_MARGIN
     -QQLEFT_MARGIN+1;
```

QQRIGHT_MARGIN

PLS-Variable in Definitionen

Syntax

QQRIGHT_MARGIN

Erläuterung

Die Variable QQRIGHT_MARGIN enthält den gültigen Wert des Compiler-Parameter RIGHTMARGIN, also die Spaltenposition in den Eingaberecords, auf der der gültige Teil einer Anweisung endet. Dieser Wert ist notwendig bei der satzweisen (record-weisen) Verarbeitung formatgebundener Eingabe. Normalerweise hat QQRIGHT_MARGIN den Wert 72.

Beispiel

```
      DCL TEXT CHAR(QQRIGHT_MARGIN-QQLEFT_MARGIN+1);
      DCL RECORD CHAR(QQLRECL) VARYING;
  READ:
      RECORD = QQTHIS_RECORD('1'B);
      TEXT = SUBSTR(RECORD,QQLEFT_MARGIN,
             QQRIGHT_MARGIN-QQLEFT_MARGIN+1);
             /* VERARBEITE TEXT */

      IF INDEX (TEXT,'END')=0 THEN GOTO READ;
```

Erläuterung

In diesem Beispiel wird satzweise gelesen. Jeder Satz wird in die Variable RECORD gelesen, die mit der aktuellen Satzlänge QQLRECL deklariert wurde. Aus dem Satz wird die interessierende Information von QQLEFT_MARGIN bis QQRIGHT_MARGIN nach TEXT gespeichert. Enthält TEXT das Wort 'END', wird aufgehört, sonst wird der nächste Satz gelesen.

QQTHIS_RECORD

PLS-Funktion

Syntax

QQTHIS_RECORD(bit)

Erläuterung

QQTHIS_RECORD liefert den gerade betrachteten logischen Satz (record) aus der Eingabedatei. Die Funktion hat ein BIT(1) Argument. Hat 'bit' den Wert '1'B, wird der Satz in der Eingabe übergangen (entspricht SKIP), sonst bleibt der Eingabezeiger stehen. Die Funktion ist nützlich zum satzweisen Verarbeiten formatgebundener Eingabe (z.B. Lesen von FORTRAN-Karten).

Beispiel

Übergehen des Restes einer Anweisung bis zum Semikolon im Fehlerfalle:

```
DO WHILE(TYPE¬=10&TYPE¬=15);
   SKIP;
   END;
```

Abfragen einer Alternative:

```
DO WHILE(TYPE =10&TYPE =15);
   IF  ID (Alternative 1) THEN DO;
     .
     .
     .
   END;
   ELSE IF ID (Alternative 2) THEN DO;
   usw., weitere Alternativen

   ELSE DO;
     MESSAGE ERROR TEXT ('FALSCHER PARAMETER');
     SKIP; / !!!!!!! /
     END;
   END;
```

Das SKIP; verhindert im Fehlerfalle eine endlose Schleife.

PLS-Anweisung in Definitionen

Syntax

```
SKIP;
```

Erläuterung

Diese Anweisung dient zum Übergehen eines Elements in der Eingabe, sie setzt den Eingabezeiger um ein Element nach rechts.

Beispiel

```
SKIP 23;
SKIP 1;
```

SKIP
n

PLS-Anweisung in Definitionen

Syntax

SKIP n;

Erläuterung

'n' ist eine Integerkonstante. Übergehe die nächsten n Zeichen. Sind bis zum End-of-File der Eingabedatei weniger als n Zeichen vorhanden, erfolgt eine Fehlermeldung, die Zeichen bis zum Dateiende werden übergangen.

Beispiel

```
SKIP(',');
SKIP('NA UND?');
DCL X CHAR(5);
X='xxxxx';
SKIP(X);
```

SKIP
(xyz)

PLS-Anweisung in Definitionen

Syntax

```
SKIP (xyz);
```

Erläuterung

Übergehe das nächste Wort in der Eingabe, falls es genau aus den Zeichen xyz besteht. 'xyz' ist ein Zeichenkettenausdruck, der keine führenden Leerzeichen enthalten darf.

Beispiel

```
SKIP BITSTRING;
SKIP BIT;
```

SKIP
BITSTRING

PLS-Anweisung in Definitionen

Syntax

SKIP BITSTRING;

Erläuterung

Übergehe das nächste Element, falls es eine Bitkette ist.

Beispiel

```
SKIP EXPRESSION;
SKIP EX;
IF ISEX THEN DO;
   CCC=NEXT_EX(NOSKIP);
   IF TYP=5 THEN III=NEXT_ID;
   SKIP EX;
   END;
```

SKIP
EXPRESSION

PLS-Anweisung in Definitionen

Syntax

SKIP EXPRESSION;

Erläuterung

Übergehe einen arithmetischen, logischen oder Zeichenkettenaus-durck, falls ein solcher als nächstes in der Eingabe vorhanden ist.

Beispiel

```
SKIP ID;
SKIP IDENTIFIER;
SKIP ID('DIES');
SKIP ID('IST');
SKIP ID('FUELLTEXT');
DCL A CHAR(32) VARYING;
A = NEXT_ID(NOSKIP);
IF A ='ENDE' THEN SKIP ID;
A = 'ABC';
SKIP IDENTIFIER(A);
```

SKIP
IDENTIFIER

PLS-Anweisung in Definitionen

Syntax

SKIP IDENTIFIER [(xyz)] ;

Erläuterung

SKIP IDENTIFIER: Übergehe das nächste Element in der Eingabe, falls es eine Benennung ist.

SKIP IDENTIFIER(xyz); Übergehe das nächste Element in der Eingabe, falls es genau die Benennung xyz ist. xyz muß ein Zeichenkettenausdruck sein.

Eine Benennung (oder ein Identifier) hat die Syntax aller Namen in PL/1. Sie muß mit einem Buchstaben oder $, #, @ beginnen, darf aus Buchstaben, Ziffern und den Zeichen $, #, @ und _ bestehen und kann maximal 32 Zeichen lang sein.

Beispiel

```
SKIP INTEGER;
SKIP INT;
```

SKIP
INTEGER

PLS-Anweisung in Definitionen

Syntax

SKIP INTEGER;

Erläuterung

Übergehe das nächste Element, falls es eine Integerkonstante ist. Die Konstante kann DECIMAL oder BINARY sein.

Beispiel

```
SKIP OPERATOR;
SKIP OP;
```

SKIP
OPERATOR

PLS-Anweisung in Definitionen

Syntax

SKIP OPERATOR;

Erläuterung

Übergehe das nächste Element, falls es ein Operator ist.
Operatoren sind: +, -, ×, /, &, |, ¬, ||, =, ¬=, <, <=, ¬<, >, >=, ¬>, **

Beispiel

```
SKIP REAL;
IF TYPE¬=5 THEN SKIP REAL;
```

SKIP
REAL

PLS-Anweisung in Definitionen

Syntax

```
SKIP REAL;
```

Erläuterung

Übergehe das nächste Element, falls es eine Realkonstante ist. Die Konstante kann DECIMAL oder BINARY, FIXED oder FLOAT sein. SKIP REAL schließt also SKIP INTEGER ein.

Beispiel

```
SKIP STRING;
SKIP STR;
```

SKIP
STRING

PLS-Anweisung in Definitionen

Syntax

SKIP STRING;

Erläuterung

Übergehe das nächste Element, falls es eine Zeichenkette ist.

Beispiel

```
SKIP WORD;
DO WHILE (TYPE¬=10&TYPE¬=15);
     SKIP WORD;
     END;
```

SKIP
WORD

PLS-Anweisung in Definitionen

Syntax

SKIP WORD;

Erläuterung

Übergehe das nächste Wort in der Eingabe (alle Zeichen bis zum nächsten Semikolon oder Leerzeichen).

Beispiel

```
SKIPB ('ABC');
SKIPB ('?');
DCL C(3)CHAR(8) VARYING
    INIT('CHAR1','C2','3');
DO I=1 TO 3;
    SKIPB (C(I));
END;
```

SKIPB
(xyz)

PLS-Anweisung in Definitionen

Syntax

SKIPB (xyz);

Erläuterung

SKIPB steht für 'SKIP BEGINN'. Übergehe das nächste Wort, falls es mit den Zeichen xyz beginnt. xyz muß ein Zeichenkettenausdruck sein. Der Rest des Wortes, bis zum nächsten Leerzeichen oder Semikolon wird übergangen.

Beispiel

```
SKIPB IDENTIFIER('NAME');
SKIPB ID('#Ø_123');
SKIPB ID('A');
DCL X CHAR(4) INIT('NAME');
DCL I PIC '99';
DO I=1 TO 10;
   SKIPB ID(X||I);
END;
```

SKIPB
IDENTIFIER

PLS-Anweisung in Definitionen

Syntax

SKIPB IDENTIFIER (xyz);

Erläuterung

Übergehe das nächste Element, falls es eine Benennung ist, die mit xyz beginnt. xyz muß ein Zeichenkettenausdruck sein.

Beispiel

```
STATEMENT NAME 'DRU.CKE' ALIAS 'PRINT'
    LABELS TO PRINTLAB PREFIXES TO PRINTPREF;
     .
     .
     .
END STATEMENT;
STA 'LIES' ALIAS 'HOLE', 'LESE';
     .
     .
     .
END STA;
STA DATENTYP INT ALIAS 'SET.ZE';
     .
     .
     .
END STA;
STAT D REAL;
     .
     .
     .
END STATEMENT;
```

PLS-Anweisung

Syntax

```
STATEMENT stat-id  [ALIAS aliaslist ]
[LABELS [TO] labelvar ] [PREFIXES [TO] prefixvar] ;
stat-id::=   [→NAME]  name | DATATYPE type
aliaslist::= aname  [,aname ]*
```

Erläuterung

NAME-DATATYPE: Im Normalfall beginnt eine POL-Anweisung mit einem Schlüsselwort, so wie auch z.B. alle PL/1-Anweisungen außer der Zuweisung und der Nullanweisung mit einem Schlüsselwort beginnen. Dieses Schlüsselwort, der Name der Anweisung, wird nach 'NAME' angegeben. 'stat-id' ist eine Zeichenkette mit maximal 32 Zeichen, eingeschlossen in Apostrophe. Sie darf keine Leerzeichen enthalten. Soll der Name der Anweisung abkürzbar sein, so muß nach den signifikanten Zeichen ein Punkt stehen. Soll der Name der Anweisung einen Punkt enthalten, müssen zwei Punkte angegeben werden.

Beispiele:
```
STATEMENT NAME 'DRU.CKE'...
STATEMENT 'LESE'..., STAT '../'... .
```

Es ist auch möglich, Anweisungen zu definieren, die nicht mit einem Schlüsselwort, sondern mit einem bestimmten Datentyp beginnen. Die möglichen Datentypen sind:

Bitstring	-	'type'	= BITSTRING	Bitkettenkonstante
Identifier	-	'type'	= IDENTIFIER	Benennungen (Namen)
Integerzahl	-	'type'	= INTEGER	alle BINARY oder DECIMAL FIXED-Konstanten
Operator	-	'type'	= OPERATOR	alle gültigen PL/1-Operatoren +, -, /, *, |, ||, =, &, >, usw.

STATEMENT
(2)

Realzahl	-	'type'	= REAL	Alle BINARY oder DECIMAL FLOAT-Konstanten
Characterstring	-	'type'	= STRING	alle Zeichenketten-Konstanten

Wenn also die Anweisungsdefinition: 'STAT DATATYPE INT ...' beginnt, so werden durch diese Definition alle Anweisungen behandelt, die mit einer ganzen Zahl beginnen. Die Datentyp-Anweisungen BIT, INT, OPE, REA und STR sind deshalb unproblematisch, weil es keine PL/1-Anweisungen gibt, die nicht mit einer Benennung (einem Identifier) beginnen. Mit dem Typ IDENTIFIER beginnt fast jede PL/1-Anweisung. Die Feststellung, um was für eine Anweisung es sich handelt, geschieht daher in folgender Reihenfolge:

1. Ist die Anweisung eine Zuweisung und ist die Zuweisung eine zum Subsystem gehörende Anweisung (nicht durch DESTROY D ASS ausgeschaltet)? Wenn ja, handelt es sich um eine Zuweisung.

2. Ist die Benennung am Anfang der Anweisung der Name einer PL/1 oder POL-Anweisung?
   Wenn ja, handelt es sich um eine Anweisung mit Namen.

3. Ist eine Datentyp-Anweisung mit 'type' = IDENTIFIER vorhanden?
   Wenn ja, handelt es sich um eine Datentyp-Anweisung.

4. Die Anweisung ist fehlerhaft.

Bei namentlichen POL-Anweisungen muß die Treiberroutine den Namen der Anweisung nicht mehr verarbeiten, der Eingabe-Zeiger steht rechts vom Namen der Anweisung. Bei Datentyp-Anweisungen steht der Eingabezeiger am Beginn der Anweisung, so daß die Treiberroutine den Wert des ersten Elements der Anweisung gewinnen kann.

STATEMENT
(3)

Alias: Eine Anweisung kann bis zu 21 verschiedene Namen haben, also außer dem Hauptnamen auch bis zu 20 Aliasnamen. Für 'aname' gelten die gleichen Regeln wie für den Namen der Anweisung (max.32 Zeichen lange Zeichenkette in Apostrophen, Abkürzung wird durch Punkt bezeichnet, keine Leerzeichen).

LABELS: Jede PL/1-Anweisung und jede POL-Anweisung kann mit Labels
PREFIXES: und Prefixes versehen sein. Im Normalfall, wenn die Angaben von "LABELS" und "PREFIXES" in der Anweisungs-Definition fehlt, werden die Prefixes und Labels vor die expandierte(n) Anweisung(en) gesetzt. Sollen jedoch der Treiberroutine, die die POL-Anweisung abarbeitet, die Labels und/oder Prefixes zur Verfügung gestellt werden, können mit "labelvar" und "prefixvar" zwei Variablennamen angegeben werden. In diese Variable speichert der PLS-Übersetzer dann die Labels bzw. Prefixes hintereinander, einschließlich aller Klammern und Doppelpunkte. Die Variablen "labelvar" und "prefixvar" werden in der STATEMENT-Definition als CHARACTER(250) VARYING automatisch deklariert.

Beispiel

```
SUBSYSTEM 'GRAPHIC' KEY 'GRAKEY';
SUB 'SUB1' KEY 'SKEY1';
SUB 'S2' 'KEY2';
DCL NAME CHAR(32) VARYING,
    KEY CHAR(7);
NAME = 'SUBNAME';
KEY = 'SUBKEY1';
SUBSYSTEM NAME KEY KEY;
```

SUBSYSTEM

PLS-Anweisung

Syntax

SUBSYSTEM name [[KEY] key] ;

Erläuterung

Die Subsystem-Anweisung dient dazu, anzugeben, zu welchem REGENT-Subsystem die folgenden Definitionen gehören sollen. Alle POL- und Datenstruktur-Definitionen bis zur nächsten SUBSYSTEM-Anweisung beziehen sich auf das angegebene Subsystem.

name — Subsystemname, ausgeschrieben oder falls abkürzbar, auch abgekürzt.

key — Schlüssel für das Subsystem. (Nullstring falls nicht angegeben).

name und key sind Zeichenkettenausdrücke. Sie dürfen keine Leerzeichen enthalten, auch nicht am Ende.

Beispiel

```
DCL X CHAR(500) VARYING;
X = THIS_STATEMENT;
IF INDEX(X,'JOINT')¬=0 THEN DO;
    .
    .
    .
   END;
X = THIS;
```

THIS_STATEMENT

PLS-Funktion

Syntax

THIS_STATEMENT

Erläuterung

Diese Funktion liefert als Zeichenkette (RETURNS (CHARACTER(3500) VARYING)) die aktuelle Anweisung vom Anfang bis einschließlich Semikolon, jedoch ohne Marken und Präfixe. Alle Leerzeichen und Kommentare innerhalb der Anweisung werden mit zurückgegeben. Der Eingabezeiger bleibt unverändert (kein SKIP).

Beispiel

Soll festgestellt werden, ob als nächstes in der Eingabe eine Integer- oder Realzahl mit oder ohne Vorzeichen steht, kann dies z.B. mit:

```
I = MOD(TYPE,10);
IF I=1 | I=2  THEN ...
```

abgefragt werden.

Abarbeiten einer Liste bis zum Anweisungsende:

```
DO WHILE (TYPE¬=10&TYPE¬=15);
  .
  .
  .
END;
```

Erläuterung:

Beim Abarbeiten einer Anweisung bis zum Ende sollte immmer außer auf Semikolon (TYPE=10) auch auf Eingabe-Ende (TYPE=15) abgefragt werden, falls bei der letzten Anweisung das Semikolon vergessen wird. Sonst ergibt sich ein endloser LOOP.

Beispiel:

```
DCL TYPMAX BIN FIXED(15);
IF TYPE=5 THEN.....
```

Erläuterung:

Hier wird TYPE nicht als
PLS-Funktion erkannt!

TYPE

PLS-Funktion

Syntax

TYPE

Erläuterung

Die Funktion TYPE gibt einen ganzzahligen Wert zwischen 1 und 15 zurück (RETURNS (BIN FIXED(15))). Sie liefert den Typ des nächsten in der Eingabe stehenden Elements zurück.

1 - Integerzahl, Binary oder Decimal Fixed-Konstante ohne Vorzeichen

2 - Realzahl, Binary oder Decimal-Float-Konstante ohne Vorzeichen

3 - Zeichenkette

4 - Bitkette

5 - Benennung (Identifier)

6 - Operator: +, -, *, /, &, |, ||, ¬, =, ¬=, >, >=, ¬>, <, <=, ¬<, **

8 - Klammer auf: (

9 - Klammer zu:)

10 - Semikolon: ;

11 - Wie 1, jedoch mit Vorzeichen + oder -

12 - Wie 2, jedoch mit Vorzeichen + oder -

13 - Anderes PL/1-Zeichen: ,,%,?

14 - Nicht - PL/1 - Zeichen, z.B. !

15 - End of File, Ende der Eingabe (EOF)

Achtung

Ist in der Statementdefinition eine Deklaration für eine Variable vorhanden, die mit TYP beginnt, wird die Funktion TYPE inaktiv! Dies gilt entsprechend für alle PLS-Funktionen.

Beispiel

```
DCL C CHAR;              /* C wäre standardmäßig aktiv */
UNACTIVE C;              /* C wird unaktiv in diesem Block */
EXEC;
CALL A(C);               /* Es wird "CALL A(C);" generiert */
END EXEC;
BEGIN;                   /* neuer Block */
ACTIVE C, I;             /* C ist aktiv in diesem Block, ebenso I */
C = 'XYZ';
EXEC;
CALL A(C);               /* Es wird "CALL A (XYZ);" generiert */
END EXEC;
I = 100;
EXEC PUT LIST(I);;       /* Es wird "PUT LIST(100);" generiert  */
  .
  .
  .
```

UNACTIVE

PLS-Anweisung in Definitionen

Syntax

UNACTIVE uname [,uname]* ;

Erläuterung

Ebenso wie ACTIVE ist UNACTIVE keine ausführbare Anweisung, sondern eine Deklaration. Die Variablen 'uname' sind in dem PL/1-Block, in dem die Anweisung steht, nicht aktiv. Es können standardmäßig aktive Variable somit von der Ersetzung in EXEC-Gruppen ausgeschlossen werden. Im äußersten Block einer Statement- oder Clause-Definition können auch PLS-Funktionen desaktiviert werden.

Beispiel

```
WORD ('xxx', NOSKIP)
WORD ('A+B')
DCL A CHAR(4) INIT('.///');
IF WORD(A) THEN GOTO L1;
IF WORD(',') THEN GOTO KOMMA;
```

WORD

PLS-Funktion

Syntax

WORD (xyz [,→SKIP | NOSKIP])

Erläuterung

WORD ist eine logische Funktion (RETURNS(BIT(1))), die dann '1'B liefert, wenn das nächste Wort in der Eingabe genau xyz ist. xyz muß ein Zeichenkettenausdruck sein.Ein Wort besteht aus allen Zeichen bis zum nächsten Leerzeichen oder Semikolon, es darf selbst keine Leerzeichen und kein Semikolon enthalten. Bei SKIP wird das Wort in der Eingabe übergangen, bei NOSKIP bleibt der Eingabezeiger unverändert.

KAPITEL 12

Ausgabe von Zeichnungen

1. Maschinenabhängigkeit

Die Ausgabe von Zeichnungen ist naturgemäß abhängig von der Art der vorhandenen Hardware und der dazugehörenden Software. Jedoch gibt es eine Reihe von Herstellern von Zeichenmaschinen, deren Grundsoftware weitgehend standardisiert ist. Diese Zeichenroutinen werden durch CALL aufgerufen. Namen und Argumentlisten sind einander angepaßt. Daher werden die Routinen der Calcomp-Basis-Software von REGENT unterstützt. Alle weiteren Plot-Routinen können ebenfalls benutzt werden. Dieses Kapitel bezieht sich in den Einzelheiten auf die REGENT-Version, die im KfK implementiert ist. Die Umstellung auf andere Anlagen ist jedoch unproblematisch.

2. Anwendung von Subsystemen mit Zeichenausgabe

Die Ausgabe von Zeichnungen bei der Subsystemanwendung wird durch den PLOT-Parameter der REGENT-Option auf der PROCEDURE-Anweisung des Anwenderprogramms gesteuert.

PLOT=CALCOMP oder PLOT

bedeutet Ausgabe auf Calcomp-Plotter.

$$\text{PLOT} = \left\{ \begin{array}{l} \text{XYNETICS} \\ \text{STATOS} \\ \text{TEKTRONIX} \end{array} \right\}$$

bedeutet Ausgabe auf Xynetics, Statos-Plotter oder Tektronix-Display. TEKTRONIX ist nur sinnvoll für Programme, die interaktiv auf einem Tektronix-Bildschirm laufen. Die Liste der Geräte kann erweitert werden.

NOPLOT

erlaubt keine Zeichenausgabe, ein Aufruf von Zeichenroutinen bewirkt eine Fehlermeldung. Alle Anwenderprogramme, die Zeichnungen erzeugen, müssen als letzte Anweisung

```
FINISH;
```

enthalten. Erst dann erfolgt der Abschluß der Plot-Dateien. Für die Zeichenausgabe muß außerdem Job-Control-Information bereitgestellt werden. Diese richtet sich nach den installationsabhängigen Erfordernissen des Betriebssystems. Benötigt wird eine PLOTTAPE-DD-Karte für Stapel-Jobs.

Die PLOTTAPE-DD-Karte hat folgendes Aussehen:

```
//G.PLOTTAPE DD UNIT=TAPEA, LABEL=(,NL),
// DCB=DEN=2, VOL=SER=αjob
```

α: T = Calcomp-Plot, Tusche
P = Calcomp-Plot, Kuli
S = Statos-Plot
X = Xynetics-Plot, Kuli
Y = Xynetics-Plot, Tusche

job: Die Buchstaben 4-8 des Jobnamens

! Beispiel

```
!
!       //IRE147P JOB (....)...
!       // EXEC  REGENT
!       //P.SYSIN DD *
!         PLOT: PROC OPTIONS(MAIN) REGENT(PLOT=STATOS);
!             Programm mit Zeichen-Anweisungen
!                   (z.B. GIPSY)
!         END PLOT;
!       //G.PLOTTAPE DD UNIT=TAPEA,LABEL=(,NL),
!       // DCB=DEN=2,VOL=SER=S147P,
```

Die Zeichnung selbst wird durch entsprechende POL-Anweisungen in den Subsxstemen bewirkt, z.B. durch die PLOT-Anweisung im Subsystem GIPSY.

3. Mehrere Schichten für die Graphik

REGENT unterstützt die graphische Datenverarbeitung auf drei Ebenen:

Ebene 1: REGENT bietet eine definierte Schnittstelle zur elementaren Treiber-Software der graphischen Ausgabegeräte. Ohne Änderung der Module kann der Subsystem-Anwender das für ihn geeignete graphische Ausgabegerät auswählen (PLOT=...).

Ebene 2: Als eigenständiges Subsystem erlaubt GIPSY die Beschreibung graphischer Aufgaben. Seine Fähigkeiten umfassen die Darstellung 2- und 3-dimensionaler Objekte: Strichgraphik, Darstellung von Wertefeldern, Behandlung von Körpern mit ebenen und gekrümmten Oberflächen. Zur Ausgabe von Zeichnungen bedient sich GIPSY der durch die Ebene 1 bereitgestellten Fähigkeiten.

Ebene 3: Andere REGENT-Subsysteme, die graphische Fähigkeiten beanspruchen, benutzen hierzu die Möglichkeiten des Subsystems GIPSY. Dies ist dadurch möglich, daß die Module eines Subsystems in der Sprache eines anderen Subsystems (hier: GIPSY) geschrieben werden können.

Der Anwender hat die Möglichkeit, auf alle Ebenen zuzugreifen, der Subsystemersteller arbeitet entweder mit Ebene 1 oder 2, um für sein Subsystem eine neue Ebene 3 zu schaffen (Abb. 12.1).

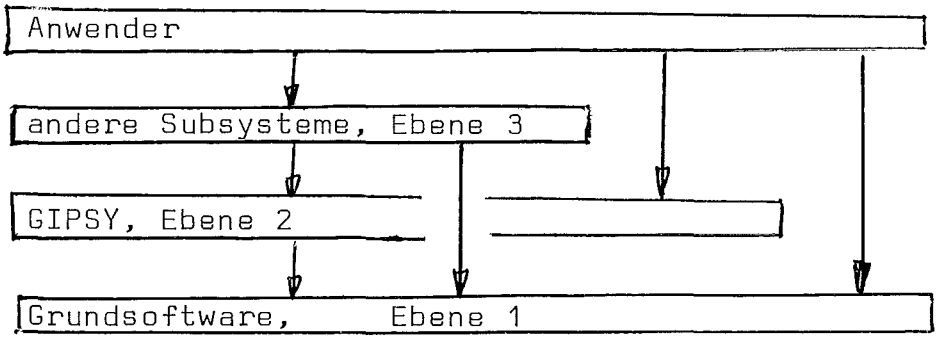


Abb. 12.1: Schichten für die Anwendung graphischer Fähigkeiten.

Beispiele

Fall 1: Der Anwender benutzt Ebene 3, andere Subsysteme.

```
P1: PROC OPTIONS(MAIN) REGENT(PLOT=STATOS);

ENTER YAPLOT;
  DIAGRAMME...              Zeichenanweisungen
  NIVEAU...                 des Subsystems
  PRESSURE                  YAPLOT.

END YAPLOT;
END P1;
```

Die YAPLOT-Anweisungen sind im betreffenden Subsystem-Handbuch beschrieben.

Fall 2: Der Anwender benutzt Ebene 2, GIPSY.

```
P2: PROC OPTIONS(MAIN) REGENT(PLOT=XYNETICS);

  ENTER GIPSY;

    DCL A SPACE(10);
    DCL B BALL;
    DCL C POLYGON(100);

    PLOT...

  END GIPSY;
END P2;
```

Die GIPSY-Anweisungen sind im GIPSY-Handbuch beschrieben.

Fall 3: Der Anwender benutzt Ebene 1, die Basis-Software.

Hierbei wird so programmiert wie in einem "normalen" PL/1-Programm mit PLOT-Aufrufen.

```
P3: PROC OPTIONS(MAIN) REGENT(PLOT);
DCL  PLOT ENTRY(DEC FLOAT(6), DEC FLOAT(6), BIN FIXED
(31)) EXT OPTIONS(FORTRAN);
```

```
CALL PLOT (X,Y,2);
```

Alle Plot-Routinen sind dabei verwendbar.

Fall 4: Ein Subsystementwickler benutzt GIPSY zur Erzeugung Plot-Moduls für Subsystem FLUXPLOT, Modulgenerator-Eingabe:

```
*SUBSYSTEM FLUXPLOT;
*MODULE FPLOT;
*PROCESS REGENT;
 FPLOT: PROC(...) REGENT(SUB=FLUXPLOT);

   ENTER GIPSY REENTER(...);
      GIPSY-Anweisungen
   END GIPSY LEAVE(...);

END FPLOT;
```

Fall 5: Ein Subsystementwickler benutzt die Basissoftware. Siehe dazu Kapitel 12.5.

Der Fall 6, GIPSY benutzt die Basissoftware, ist bereits fertig programmiert. Er entspricht genau dem Fall 5.

4. Die Schnittstelle zur Grundsoftware

Die folgenden 2 Kapitel sind nur für Subsystem-Entwickler wichtig.

In den Modulen der REGENT-Subsysteme müssen die Grundfähigkeiten zum Erzeugen graphischer Ausgabe, wie das Erzeugen von Strichen, Punktsymbolen und Texten, verfügbar gemacht werden. Folgende Forderungen müssen dabei erfüllt werden:

- ohne Neuübersetzen oder -binden von Modulen muß bei der Subsystem-Anwendung von einem auf das andere graphische Ausgabegerät umgeschaltet werden können.

- Die in Modulen vorhandene graphische Software muß, da die Module mehrfach dynamisch aufgerufen werden können (auch rekursiv), die REENTRANT-Eigenschaft besitzen.

- Bei der Übertragung von REGENT auf einen anderen Rechner mit anderer graphischer Software soll ein Neuübersetzen von Moduln nicht erforderlich sein, die Umstellung des REGENT-Kernes muß einfach möglich sein.

Um diese Forderungen zu erfüllen, wurde eine geräteunabhängige Schnittstelle für die Aufrufe der elementaren Grundsoftware bereitgestellt. Weil zum Zeitpunkt der Erstellung des REGENT-Systems die PLOT-Aufrufe der Firma Calcomp (sog. "Calcomp-Software") in großem Umfang angewendet wurden, wurde die Schnittstelle in Form von Aufrufen auf die Routinen PLOT, WHERE, FACTOR und PEN entsprechend den Calcomp-Konventionen bereitgestellt. Die höheren Funktionen (SYMBOL, NUMBER, LINE, AXIS, ARROW, LGAXS, etc.) greifen nur auf diese Schnittstelle zu. Sie sind geräteunabhängig und REENTRANT und können daher in die Module eingebunden werden. Die Schnittstelle besteht aus kleinen Routinen, die über eine Interface-Datenstruktur die Adresse der geräteabhängigen Treiberroutinen im REGENT-Kern gewinnen und in den Kern verzweigen. Bei der REGENT-Anwendung wird über einen Parameter

$$\text{PLOT} = \left\{ \begin{matrix} \text{STATOS} \\ \text{CALCOMP} \\ \text{XYNETICS} \\ \text{TEKTRONIX} \end{matrix} \right\} \text{ das graphische Ausgabegerät}$$

ausgewählt, entsprechend wird die dazugehörige Treibersoftware in den Mainmodul eingebunden. Die Abb. 12.2 zeigt den Zugriff auf die graphische Software aus einem Modul heraus.

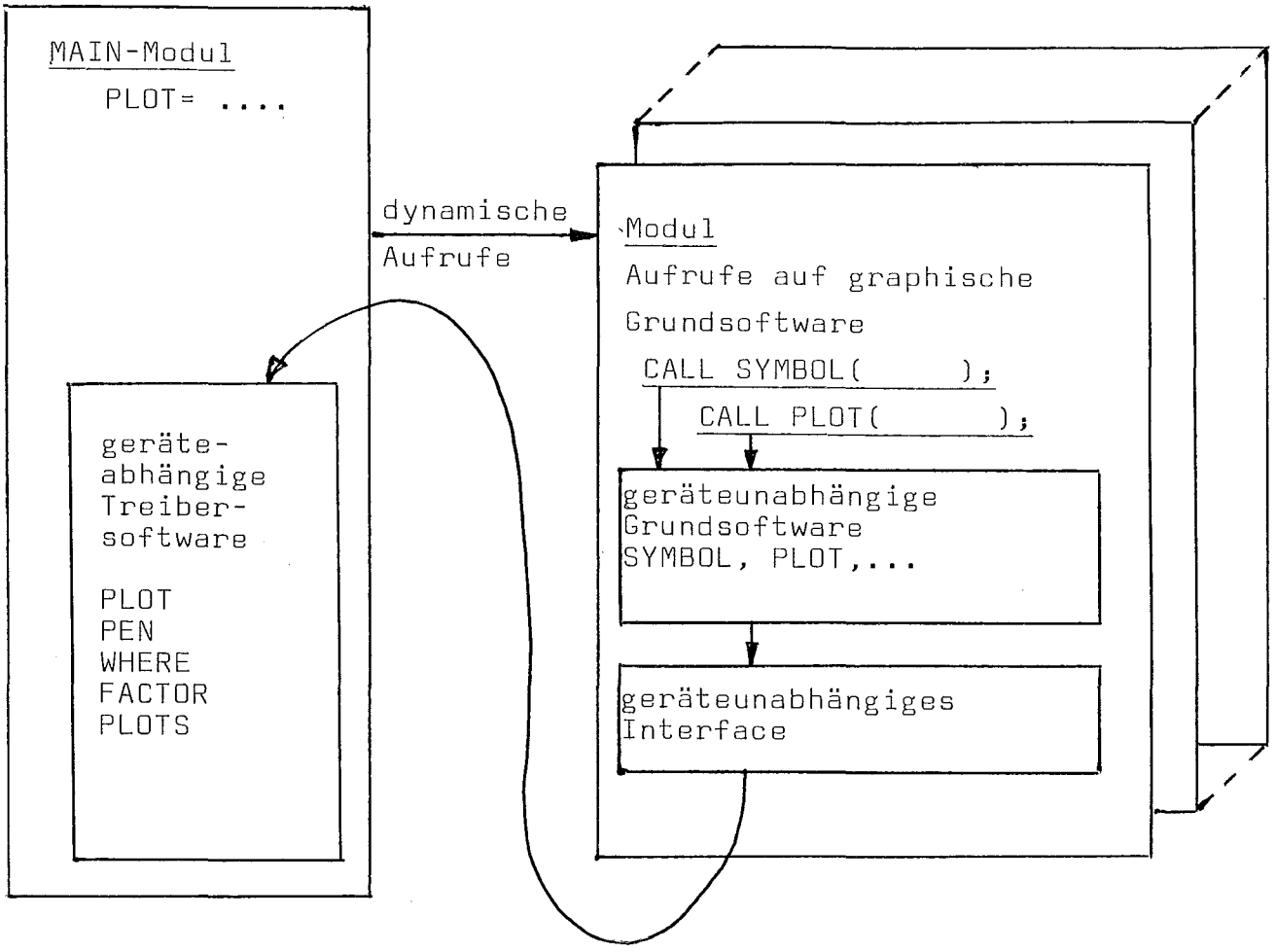


Abb. 12.2 Schnittstelle zur Treiber-Software graphischer Ausgabegeräte. Die Anwendung der Plot-Schnittstelle in Moduln wird im folgenden Kapitel beschrieben.

5. Zeichenprogramm-Aufrufe in Moduln

Die Software der Plotter-Hersteller (Calcomp, Statos) kann in REGENT-Moduln aufgerufen werden. Jeder Modul und jede Routine, die Zeichenausgabe erzeugt, muß die Anweisung:

```
!     %INCLUDE QQMAC(QQPLOT)
```

enthalten.

Zur Initialisierung der Zeichenausgabe darf nicht CALCIN und nicht PLOTS aufgerufen werden. Stattdessen ist in das Programm einzufügen:

```
!     %INCLUDE QQMAC(QQCALCIN);
```

Der Standardwert für die Plot-Pufferlänge (für PLOTS) ist 1000. Soll er verändert werden, ist vor dem %INCLUDE QQMAC(QQCALCIN) eine Zuweisung: QQLEN= Pufferlänge; zu kodieren.

Der Abschluß der Zeichenarbeiten darf nicht durch Aufruf von PLOTC(x,y,999) erfolgen, er wird von REGENT beim FINISH-Statement durchgeführt.
Nach der Initialisierung können die Plot-Routinen in allen Moduln, die %INCLUDE QQMAC(QQPLOT) enthalten, in der üblichen Weise benutzt werden:

- Deklaration der Routinen mit allen Parametern als ENTRY EXTERNAL OPTIONS (FORTRAN),
- Aufruf durch CALL.

Für die Routinen PLOTC, FACTOR und WHERE dürfen keine Deklarationen angegeben werden, diese werden durch QQPLOT eingefügt.

Für die Deklaration der Routinen LINE, SYMBOL, NUMBER, SCALE und AXIS gibt es %INCLUDE-Members, damit nicht jedesmal die volle Deklaration geschrieben werden muß.

Routine	Member für INCLUDE
LINE	QQLINE
SYMBOL	QQSYMBOL
NUMBER	QQNUMBER
SCALE	QQSCALE
AXIS	QQAXIS

Nach dem %INCLUDE auf QQPLOT sind folgende Werte zugreifbar, die Eigenschaften des Plotters beschreiben:

```
QQDEVICE (CHAR(8))       Plotter-Kennzeichnung: CALCOMP, STATOS,
                         XYNETICS oder TEKTRONIX
QQHEIGHT (DEC FLOAT(6)) Höhe des Zeichenpapiers in Meter
QQLENGTH (DEC FLOAT(6)) Länge (Breite) des Zeichenpapiers in Meter
```

! Beispiel

```
!
!     *MODUL PLOT;
!      PLOT: PROC(X,Y,N);
!      DCL (X,Y)(*)DEC FLOAT(6);
!      %INCLUDE QQMAC(QQPLOT);
!         /* Zeichenfähigkeiten werden benutzt */
!      QQLEN=2000;
!         /* Ändern der Pufferlänge */
!      %INCLUDE QQMAC(QQCALCIN);
!         /* Initialisieren der Plotroutinen */
!      %INCLUDE QQMAC(QQLINE,QQAXIS);
!         /* LINE und AXIS werden benutzt */
!      DCL SMOOT ENTRY(DEC FLOAT(6),
        DEC FLOAT(6), BIN FIXED(31)) OPTIONS (FORTRAN) EXTERNAL;
!         /* Auch SMOOT wird benötigt */
!      BREITE=QQHEIGHT;
!         /* Zeichenpapierhöhe */
!      CALL FACTOR(BREITE/100.);
!         /* FACTOR darf nicht deklariert werden */
!      CALL LINE(X,Y,N,...);
!      CALL AXIS(...);
!
!      CALL SMOOT(...);
!
!      END PLOT;
```

Anmerkung

Wegen der fehlenden Standardisierung graphischer Ausgabe ist dieser REGENT-Bereich nicht in Form einer PL/1-Spracherweiterung bereitgestellt worden. Es ist jedoch möglich, daß künftige REGENT-Versionen andere Mittel zur Zeichenausgabe bereitstellen.
Die weitaus meisten Anwendungen graphischer Ausgabe können mit dem Subsystem GIPSY durchgeführt werden.

KAPITEL 13

Ablauf- und Fehlernachrichten

13.1 Nachrichten des REGENT-Systems

Alle Nachrichten des REGENT-Systems besitzen einen Text, dessen erstes Wort eine Identifikation darstellt. Diese Identifikation besteht aus einer dreibuchstabigen Kennung und einer anschliessenden (je pro Kennung eindeutigen) dreistelligen Nummer. Die Kennungen bedeuten:

BNK	Datenbankverwaltung
DDS	Verwaltung dynamischer Datenstrukturen
MGN	Modul=/Routinen-generator
MSS	Nachrichtenverwaltung
PLR	PLR-Precompiler
POL	REGENT-Übersetzer
RMM	Modulverwaltung
UTI	Hilfsroutinen

Die Nachrichten sind im folgenden alphabetisch nach ihren Identifikationen geordnet.

BNK002 NAME OF BANK DOES NOT CORRESPOND WITH TITLE OPTION

Erläuterung
Der Name, der beim Öffnen der Bank angegeben wurde, stimmt nicht mit dem externen Namen der Bank überein, mit dem diese initialisiert wurde.

BNK003 NO ACCESS RIGHT FOR BANK OR POOL

Erläuterung
Das angegebene Paßwort berechtigt nicht, mit der Bank/Pool in dem gewünschten Mode (input/output/update) zu arbeiten.

BNK004 NO VALID FATHER REFERENCE

Erläuterung
Der Vater (Bank/Pool) ist keine gültige Referenz. Z.B. ist er nicht geöffnet bzw. bereits wieder geschlossen oder überhaupt kein Pool.

BNK005 SPECIFIED FATHER IS DATAPOOL

Erläuterung

Der Vater eines Pools kann immer nur die Bank selbst bzw. ein Catalogpool sein.

BNK006 INVALID POOL OPERATION

Erläuterung

Der Fehler kommt aus einem open statement, bei dem der mode des Vaters nicht output bzw. update ist. Z.B. muß bei open und delete der Vater update mode haben, bzw. bei open für output der Vater mindestens ebenfalls output.

BNK007 OPEN MODE DOES NOT CORRESPOND TO OPEN MODE OF FATHER

Erläuterung

Es wurde der Versuch gemacht, einen pool mit einem höheren mode als dem des Vaters zu öffnen.

BNK008 POOL ALREADY OPEN

Erläuterung

Der pool ist bereits geöffnet. Entweder ist der pool title falsch spezifiziert oder der Vater.

BNK009 MAXIMUM NUMBER OF ALLOWED POOLS IN FATHER CATALOG REACHED

Erläuterung

Selbsterklärend.

BNK010 NO SPACE FOR POOL AVAILABLE. CALL COMPRESS

Erläuterung

Beim Versuch, einen pool zu initialisieren, wurde festgestellt, daß die Bank auch nach Reorganisation der Freilistenverwaltung nicht mehr genügend Platz enthält bzw. zu stark fragmentiert ist. Die Initialisierung wurde nicht durchgeführt.
Aktion: COMPRESS oder weniger space.

BNK011 NO SPACE FOR REORGANISATION OF FREESPACE MANAGEMENT!
CALL COMPRESS

Erläuterung

Beim Initialisieren wurde mit FREELIST und DELETELIST sowie LISTPAGE die Maximalzahl der Einträge in die Freispeicherlisten festgelegt. Diese Anzahl ist erreicht und die Segmentierung der Bank zu stark, um Platz für eine Reorganisation zu erhalten.
Aktion: Compress oder Copy in eine andere Bank, deren Freispeicherbereich größer ist. (FREELIST, DELETELIST erhöhen.)

BNK012 DATAPOOL FULL

Erläuterung

Ein Datapool ist voll beschrieben. Er wurde z.B. mit mod output eröffnet.

BNK013 SPECIFIED POOL NOT FOUND IN FATHER CATALOG

Erläuterung

Der Vaterkatalog enthält keinen Pool mit dem im open statement spezifizierten Namen (title). Falscher Title oder Vater.

BNK014 MAXIMUM NUMBER OF GAPS IN FREESPACE MANAGEMENT REACHED.
CALL COMPRESS

Erläuterung

Die Maximalzahl der Freispeichereinträge wurde erreicht.
Aktion: siehe unter BNK 011.

BNK015 BANK ALREADY OPENED

Erläuterung

Selbsterklärend.

BNK020 BANK NOT OPEN

Erläuterung

Selbsterklärend.

BNK030 SPECIFIED POOL NOT OPEN

Erläuterung

Selbsterklärend.

BNK031 STORE-STRINGLENGTH <=0

Erläuterung

Die im STORE-Befehl angegebene Stringlänge ist <=0.

BNK032 STORE-POOL NOT OPENED

Erläuterung

Der im STORE-Statement referierte Pool ist nicht geöffnet.

BNK033 STORE-POOL NO DATAPOOL

Erläuterung

Der im STORE-Befehl referierte Pool hat nicht das Attribut DATA.

BNK034 NO ACCESS-RIGHTS FOR STORE

Erläuterung

Der im STORE-Statement referierte Pool wurde nicht mit STORE-Rechten geöffnet.

BNK035 OPEN-MODE IS INPUT

Erläuterung

Der im STORE-Statement referierte Pool wurde mit INPUT eröffnet, obgleich er durch ein STORE-Statement referiert wird.

BNK036 EOP-END OF POOL BY STORE SEQUENTIAL

Erläuterung

Bei einer sequentiellen STORE-Operation wurde das Poolende erreicht. Der Befehl wurde nicht ausgeführt.

BNK037 EOP-END OF POOL BY STORE DIRECT

Erläuterung

Bei einer direkten STORE-Operation wurde das Poolende erreicht. Der Befehl wurde nicht ausgeführt.

BNK041 RETRIEVE-STRINGLENGTH <=0

Erläuterung

Die im RETRIEVE-Befehl angegebene Stringlänge ist <=0.

BNK042 RETRIEVE-POOL NOT OPENED

Erläuterung

Der im RETRIEVE-Statement referierte Pool ist nicht geöffnet.

BNK043 RETRIEVE-POOL NO DATAPOOL

Erläuterung

Der im RETRIEVE-Statement referierte Pool ist kein Datapool.

BNK044 NO ACCESS RIGHTS FOR RETRIEVE

Erläuterung

Der im RETRIEVE-Statement referierte Pool wurde nicht mit RETRIEVE-Rechten geöffnet.

BNK046 EOP-END OF POOL BY RETRIEVE SEQUENTIAL

Erläuterung

Bei einer sequentiellen RETRIEVE-Operation wurde das Poolende erreicht. Der Befehl wurde nicht ausgeführt.

BNK047 EOP-END OF POOL BY RETRIEVE DIRECT

Erläuterung

Bei einer direkten RETRIEVE-Operation wurde das Poolende erreicht. Der Befehl wurde nicht ausgeführt.

BNK048 SET-POOL NO DATAPOOL

Erläuterung

Der im SET-Befehl referierte Pool ist kein Datapool.

BNK049 EOP-END OF POOL BY SET

Erläuterung

Mit dem SET-Befehl sollte der I/O-Zeiger des referierten Pools über die Poolgrenze hinauszeigen.

BNK050 REGENT-OPTION BANK MISSING

Erläuterung

In der REGENT-Option des PROCEDURE-Statements der MAIN-Procedure fehlt die Option BANK.

BNK051 REGENT-OPTIONS DA AND BANK MISSING

Erläuterung

In der REGENT-Option des PROCEDURE-Statements der MAIN-Prozedur fehlen die Optionen DA und BANK.

BNK070 NO ACCESS RIGHTS TO READ

Erläuterung

Das angegebene Paßwort genügt nicht, um die Bank zu lesen.

BNK071 NO ACCESS RIGHTS TO READ PRIVILEGED

Erläuterung

Wie unter BNK 070, nur privilegiertes Lesen.

BNK081 FROM-OR TO-VALUE <=0

Erläuterung

Die in der FROM- oder TO-Option anzugebenden Werte sind relative Pool-Offsets und müssen positiv sein.

BNK082 INDEX NR.i TOO GREAT OR <=0
INDEXLIST: $n_1 \ldots n_m$

Erläuterung

Der i-te Index der Indexliste $n_1 \ldots n_m$, die ein DDS-Blattelement in einem Pool bezeichnen soll, ist zu groß oder <=0.

BNK083 INDEXLIST FOR LEAF TOO LONG

Erläuterung

Die Indexliste, die ein DDS-Blatt bezeichnen soll, ist zu lang.

BNK084 LENGTH OF PL/1-VAR¬=LENGTH OF TOTAL-LEAF: var leaf

Erläuterung

Die Längen der im STORE- oder RETRIEVE-Statement verknüpften Elemente sind ungleich: var ist die Länge der PL/1-Variablen, leaf die Länge des zu bearbeitenden Blattes.

BNK085 LAST INDEX TOO GREAT OR <=0
INDEXLIST: $n_1 \ldots n_m$

Erläuterung

Es soll ein Blattelement bearbeitet werden, das im Pool nicht enthalten oder <=0 ist.

BNK086 LENGTH OF PL/1-VAR¬=LENGTH OF LEAF-ELEMENT: var leaf

Erläuterung

Die Längen der im STORE- oder RETRIEVE-Statement verknüpften Elemente sind ungleich. var ist die Länge der PL/1-Variablen, leaf die Länge des Blatt-Elements.

BNK087 INDEXLIST FOR LEAF-ELEMENT TOO LONG

Erläuterung

Die Indexliste, die ein DDS-Blattelement bezeichnen soll, ist zu lang.

BNK088 SUBARRAY NOT DEFINED
INDEX: i
INDEXLIST: $n_1 \ldots n_m$

Erläuterung

Der i-te Index der Indexliste zur Bezeichnung eines DDS-Elements im Pool weist auf ein undefiniertes DDS-Element.

BNK089 INDEX i TOO GREAT OR <=0
INDEXLIST: $n_1 \ldots n_m$

Erläuterung

Der i-te Index der Indexliste $n_1 \ldots n_m$ zur Bezeichnung eines DDS-Elements im Pool ist zu groß oder <=0.

BNK090 DYNAMIC ARRAY STORED FOR SEQUENTIAL ACCESS ONLY

Erläuterung

Das im RETRIEVE-Statement referierte Dynamic Array im angegebenen Datapool (POOL-Option) ist lediglich für sequentiellen Zugriff zugänglich. Für direkten Zugriff muß im zugehörigen STORE-Befehl die Option DIRECT angegeben werden.

BNK100 BLKSIZE SHOULD BE GREATER THAN 144

Erläuterung

Die kleinste zulässige Größe der Regions beträgt 144 Bytes. (Maximal eine Spurlänge, also bei 2314-Platten z.B. 7294 bytes.)

DDS001 DESCRIPTOR UNDEFINED
INTERRUPT AT INDEX i
INDEXLIST: $n_1 \ldots n_i \ldots n_m$

Erläuterung

Bei einem DDS-Zugriff wurde am i-ten Index der angegebenen Indexliste $n_1 \ldots n_m$ ein undefinierter Descriptor gefunden.

DDS002 NO STEP, BUT STEP REQUIRED
INTERRUPT AT INDEX i
INDEXLIST: $n_1 \ldots n_i \ldots n_m$

Erläuterung

Bei einem DDS-Zugriff überschreitet der i-te Index der Indexliste $n_1 \ldots n_m$ die definierte Länge eines DDS-Vektors, für den keine STEP-Option definiert ist.

DDS003 INDEXLIST TOO LONG. LEAF REACHED.
INTERRUPT AT INDEX i
INDEXLIST: $n_1 \ldots n_i \ldots n_m$

Erläuterung

Bei einem DDS-Zugriff wurde mit dem i-ten Index der Indexliste $n_1 \ldots n_m$ bereits ein Blatt erreicht.

DDS004 NO NODE REACHED
INTERRUPT AT INDEX i
INDEXLIST: $n_1 \ldots n_i \ldots n_m$

Erläuterung

Bei einem DDS-Zugriff auf einen Strukturknoten wurde mit der Indexliste $n_1 \ldots n_m$ kein Knoten erreicht.

DDS005 NO LEAF REACHED
INTERRUPT AT INDEX i
INDEXLIST: $n_1 \ldots n_i \ldots n_m$

Erläuterung

Bei einem DDS-Zugriff auf ein Strukturblatt wurde mit der Indexliste $n_1 \ldots n_m$ kein Blatt erreicht.

DDS006 DESCRIPTOR NOT INITIATED
INTERRUPT AT INDEX i
INDEXLIST: $n_1 \ldots n_i \ldots n_m$

Erläuterung

Es sollte ein DDS-Zugriff auf eine undefinierte Struktur erfolgen (unter Verwendung der Indexliste $n_1 \ldots n_m$).

DDS007 LENGTH OF TARGET¬= LENGTH OF SOURCE
INTERRUPT AT INDEX i
INDEXLIST: $n_1 \ldots n_m$

Erläuterung

Bei einem Zugriff auf ein Blatt-Element (mit der Indexliste $n_1 \ldots n_m$) entspricht die Länge des Blattelements nicht der Länge der Variablen, die für den Zugriff benutzt wird.

DDS008 INDEX <=0
INTERRUPT AT INDEX i
INDEXLIST: $n_1 \ldots n_i \ldots n_m$

Erläuterung

Bei einem DDS-Zugriff wurde der i-te Index <=o angegeben.

DDS009 NO FIXED SUBARRAY FOUND

Erläuterung

Die im LOOSE-Befehl angegebene DDS-Substruktur war nicht FIXED.

DDS010 DDS-STRUCTURE NOT FOUND

Erläuterung

Die im RESET-Befehl angegebene DDS-Struktur existiert nicht.

DDS011 DEFINE-OPTION: ,(,,,,,) CONTAINS ELEMENTS <=0
INDEXLIST: $i_1 \ldots i_m$

Erläuterung

Die im DEFINE-Statement angegebene Indexliste zur Festlegung der Strukturschachtelung enthält Werte <=0.

DDSO12 BLKSIZE TOO SMALL. BLKSIZE SHOULD BE >= blksize

Erläuterung

Um eine Reorganisation durchführen zu können, muß die BLKSIZE >= blksize sein.

DDSO13 CORE OVERFLOW BY DDS

Erläuterung

Trotz einer Reorganisation könnte kein Platz für DDS-Strukturen allokiert werden.

DDSO14 STEP-OPTION FOR FIXED SUBARRAY SUSPENDED

Erläuterung

Bei einem DDS-Zugriff sollte ein Vektor automatisch erweitert werden (STEP-Option), für den diese Operation vorübergehend verboten ist (FIXED-Vektor).

DDSO15 REGENT OPTION DA MISSING

Erläuterung

In der REGENT-Option des PROCEDURE-Statements der MAIN-Prozedur fehlt die Option DA.

DDSO16 DESCRIPTOR ¬=NULL( )

Erläuterung

Durch die Utility-Prozedur QQDIBD sollte ein Descriptor initialisiert werden, dessen Wert ¬=NULL ist.

DDS500 POOL: MAX= max OCCUPIED=occupied
NEEDED=needed

Erläuterung
Es wird eine DDS-Reorganisation erforderlich. Der Pool hat die maximale Größe von 'max' Bytes, belegt sind 'occupied' Bytes, benötigt werden 'needed' Bytes.

DDS501 REORG-PURGE: P=priority
N= nodes L=leaves R=references
S= sum

Erläuterung
Bei einer Reorganisation wurden 'nodes' Byte Knoten, 'leaves' Byte Blätter, 'references' Byte Referenzen und insgesamt 'sum' Bytes auf die Platte (QQDDSF) geschrieben. Die Priorität dieser Blöcke wäre 'priority'.

DDS502 DDS-POOL-STATISTIC
DDS502 ID=id
DDS502 DPOOL=dpool
DDS502 ACT-REAL=actual MAX-REAL=maxreal
DDS502 REAL: NODES=nodes REF=ref LEAVES=leaves
DDS502 ACT-SUM=actsum MAX-SUM=maxsum
DDS502 REORG-LEVEL=reorglevel
DDS502 NUMBER OF REORGS=reorgs
DDS502 NUMBER OF I/O= ios
DDS502 MAX ELEM.-SIZE=maxelem.
DDS502 text 1
DDS502 text 2
DDS502 text 3

Erläuterung

id	:	END STATISTIC wenn durch REGENT Abschluß
dpool	:	Größe des DDS-Pools (real) in Bytes
actreal	:	aktuell belegter Poolplatz in Bytes (real)
maxreal	:	maximal belegter Pool-Platz in Bytes (real)
nodes	:	davon 'nodes' Bytes durch Knoten 'ref' Bytes durch Referenzen und 'leaves' Bytes durch Blääter
actsum	:	aktuell belegter Platz für DDS (real und virtuell)
maxsum	:	maximal belegter Platz für DDS (real und virtuell)
reorg level	:	Stufe bis zu der reorganisiert wird
reorgs	:	Zahl der durchgeführten Reorganisationen
ios	:	Zahl der I/O-Operationen (READ, WRITE)
maxelem.	:	Maximale Elementgröße
text1	:	Informieren über die gewählte Form der
text2	:	Poolreorganisation (siehe Kap. 10,
text3	:	CALL (QQDREO)

MGN001 ×SUBSYSTEM STATEMENT MISSING. INPUT SKIPPED UNTIL THE NEXT ×SUBSYSTEM STATEMENT IF ANY

Erläuterung

Das erste Statement der Modulgenerator-Eingabe war kein ×SUBSYSTEM-Statement. Die Eingabe wird bis zum nächsten ×SUBSYSTEM-Statement ignoriert. Ist kein weiteres ×SUBSYSTEM-Statement vorhanden, erfolgt keine Aktion mehr.

MGN002 AFTER THE ×SUBSYSTEM STATEMENT WAS NOT A VALID STATEMENT

Erläuterung

Das Statement, welches sich unmittelbar nach dem ×SUBSYSTEM-Statement befand, war weder ein ×MODULE noch ein ×ROUTINE-Statement.

MGN003 MODULE HAS NO NAME. MODULE GENERATION INHIBITED

Erläuterung

Dies ist ein Folgefehler, welcher aus der Nachricht MGN002 resultiert. Es wird ein Modul ohne Namen erzeugt. Dieser wird vom LINKAGE EDITOR nicht geladen.

MGN004 FOLLOWING STATEMENT IS INCORRECT AND WILL BE IGNORED: text

Erläuterung

Das Statement text ist kein gültiges Statement für den Modulgenerator und wird aus diesem Grund ignoriert.

MGN005 THE DEFAULT COMPILER PARAMETER WAS CHANGED

Erläuterung

Die Standard Compiler-Parameter wurden mit dem ×CPARM-Statement überschrieben.

MGN006 THE DEFAULT LINKAGE EDITOR PARAMETER WAS CHANGED

Erläuterung

Die Standard LINKAGE EDITOR Parameter wurden mit dem *LPARM-Statement überschrieben.

MGN007 THE DEFAULT PRE-COMPILER PARAMETER WAS CHANGED

Erläuterung

Die Standard PRE-Compiler Parameter wurden mit dem *PPARM-Statement überschrieben.

MGN008 MORE THAN 100 ENTRIES REQUIRED

Erläuterung

Der RMM-Modul hat mehr als 100 RMM-Entries. Der Modulgenerator verarbeitet nur 100 Entries. Alle anderen werden ignoriert.

MGN009 INPUT MORE THAN 360 CHARACTER. INPUT TRUNCATED

Erläuterung

Ein Modulgeneratorbefehl besteht aus mehr als 360 Zeichen (Blanks mitgerechnet). Dies bedeutet, er ist größer als 5 Lochkarten. Die Eingabe wird nach dieser max.Grenze abgeschnitten.

MGN010 NO DD-STATEMENT FOR RESERVE ACTION. NO MODULE/ROUTINE GENERATION. WRONG DDNAME = ddname

Erläuterung

Während des Ladens eines Moduls wurde die angegebene Bibliothek (Lib-Parameter auf der *MODULE-Anweisung) nicht mit einer entsprechenden DD-Karte in der Job-Control-Language referiert. Es erfolgt keine Aktion des LINKAGE EDITORS.
ddname ist der Name der fehlenden DD-Karte

MGN011 THE ROUTINE m WAS LOADED ON LIBRARY WITH DD_NAME = d

Erläuterung

Die Routine m wurde in die Bibliothek mit dem DD-Namen d geladen.

MGN012 THE MODULE m WAS LOADED ON LIBRARY WITH DD_NAME = d

Erläuterung
Der Modul m wurde in die Bibliothek mit dem DD-Namen d geladen.

MGN013 RETURN CØDE AFTER MODULE/ROUTINE
GENERATION = r

Erläuterung
Die Ausführung der Modul bzw. Routinengenerierung schloß mit dem Returncode r ab.

MGN014 NO MODULE/ROUTINE GENERATION BECAUSE OF CONDITION CODE

Erläuterung
Es wurde kein Modul bzw. Routine erzeugt, da bei der Ausführung des LINKAGE EDITORS ein Fehler aufgetreten war.

MGN015 RETURN CODE AFTER MACROPROCESSOR STEP = r

Erläuterung
Die Ausführung des Makroprocessors schloß mit dem Returncode r ab.

MGN016 RETURN CODE AFTER REGENT TRANSLATOR STEP = r

Erläuterung
Die Ausführung des REGENT-Übersetzers schloß mit dem Returncode r ab.

MGN017 RETURN CODE AFTER PLR-PRECOMPILER STEP = r

Erläuterung
Die Ausführung des PLR-Precompilers schloß mit dem Returncode r ab.

MGN018 RETURN CODE AFTER COMPILE STEP = r

Erläuterung
Die Ausführung des PL/1-Compilers schloß mit dem Returncode r ab.

MGN019 RETURN CODE OF MODULGENERATOR = r

Erläuterung
Die Ausführung des Modulgenerators schloß mit dem Returncode r ab. Dies ist der maximale Returncode aller vorangegangenen Steps.

MGN020 ERROR IN MODULE GENERATION. MEMBER QQSUBS ON FILE SUBLIB CANNOT BE OPENED

Erläuterung
Beim Eröffnen der Subsystemtabelle von der Datei REGENT.PLSTRAN.DATA trat ein Fehler auf. DD-Karte fehlt oder ist falsch, oder I/O-Error.

MGN021 ERROR IN MODULE GENERATION WHILE READING FILE QQSUBLIB. QQPEMEM ERROR NO.13

Erläuterung
Beim Lesen der Subsystemtabelle von der Datei REGENT.PLSTRAN. DATA trat ein I/O-Error auf.

MGN022 ERROR IN MODULE GENERATION, SPECIFIED SUBSYSTEM m NOT FOUND, PREFIX 'ØØ' ASSUMED

Erläuterung
Das auf der *SUBSYSTEM-Anweisung angegebene Subsystem m wurde nicht gefunden. Die Module erhalten den Prefix 'ØØ'.

MGNO23 LIBRARY WITH DDNAME = d WAS COMPRESSED

Erläuterung
Die Bibliothek mit dem DD-Namen d wurde comprimiert.

MGNO24 LESS THAN P1 TRKS ØR P2 % AVAILABLE SPACE LEFT IN LIBRARY WITH DDNAME = d

Erläuterung
Die Bibliothek mit dem DD-Namen d wurde komprimiert. Danach wurde festgestellt, daß trotzdem weniger P2 % oder weniger als P1 TRKS Platz zur Verfügung steht. Die Bibliothek müßte aus diesem Grund vergrößert werden.

MGNO25 INVALID DESTROY STATEMENT.NAME LIST MISSING

Erläuterung
Es wurde vergessen den Namen des zu löschenden Moduls bzw. Routine anzugeben.

MGNO26 MODULE/MEMBER m HAS BEEN DESTROYED

Erläuterung
Es wurde der Modul bzw. die Routine m gelöscht.

MGNO27 MODULE/MEMBER m NOT DESTROYED, BECAUSE MEMBER NOT FOUND IN LIBRARY WITH DDNAME = d

Erläuterung
Es wurde versucht den Modul bzw. die Routine n zu löschen, sie befand sich aber nicht in der Bibliothek die mit dem DD-Namen d referiert wurde.

MGNO28 MODULE/MEMBER m NOT DESTROYED, BECAUSE DD-Card FOR LIBRARY d MISSING

Erläuterung
Es wurde versucht den Modul bzw. die Routine m zu löschen aber es fehlte die DD-Karte für die Bibliothek die mit dem DD-Namen d referiert wurde.

MGN029 nnn TRKS OF mmm ARE USED ON LIBRARY WITH DD-NAME dd-name

Erläuterung

Es sind von mmm Spuren einer Bibliothek mit dem DD-Namen dd-name nnn belegt. Eine zu kleine Differenz der beiden Werte würde bedeuten, daß vor dem nächsten Ladevorgang die Bibliothek komprimiert werden müßte.

Fehlermeldungen des PLR-Precompilers

Hinweis: In allen PLR-Fehlermeldungen wird nach der Fehlernummer die Nummer der PLR-Anweisung angegeben, in der der Fehler auftrat.

PLR001 END OF FILE BEFORE END OF STMT

Erläuterung

Während der Abarbeitung einer Anweisung wurde das Ende der Eingabe erreicht.

Beispiel:
```
CLOSE FILE(SYSPRINT)
   End of Dataset
```

PLR002 ERROR IN 'ON'-STMT.CONDITION WRONG

Erläuterung

Bei der Verwendung der 'ON'-Anweisung wurde eine falsche 'CONDITION' angegeben.

Beispiel:
```
ON EQUAL ...
```

PLR003 FIRST STMT. OF EXTERNAL PROCEDURE NOT 'PROCEDURE'

Erläuterung

Die erste Anweisung einer externen Prozedur muß eine 'PROCEDURE'-Anweisung sein. Eine externe Prozedur wird daran erkannt, daß ihre erste Anweisung auf Stufe 0(Null) steht. Dies kann versehentlich durch eine 'END'-Anweisung zuviel hervorgerufen werden.

Beispiel:
```
DCL B INIT(0);
DO I=1 TO 10;
   B = B+I;
END; /* der Schleife */
```

PLR004 LOGICAL END OF PROGRAM FOUND BEFORE END OF INPUT FILE

Erläuterung

Es wurde bei der Verarbeitung Stufe 0 erreicht, obwohl noch Eingabedaten vorhanden sind.

Beispiel:
```
A: PROC;
END A;
PUT LIST(B);
```

PLR005 Version 1:

STMT. TOO LONG. QUOTE (or '*/' respectively) INSERTED BEFORE 'string'

Erläuterung

Eine begonnene Zeichenkette oder ein begonnener Kommentar wurde als zu lang erkannt. Ein Hochkomma oder das Symbol '*/' wurde an der Stelle 'string' eingefügt. In der Eingabe fehlte das entsprechende Symbol um eine Zeichenkette oder einen Kommentar zu beenden.

Version 2:

STMT.TOO LONG. FIRST RECORD OF STATEMENT DROPPED

Erläuterung

Eine Anweisung wurde als zu lang erkannt. Die erste Karte wurde überlesen.

PLR006 IN EXPRESSION IDENTIFIER MISSING AFTER QUALIFYING POINT

Erläuterung

Die Angabe einer qualifizierten Variablen ist unvollständig und hört mit einem '.' (Punkt) auf.
Beispiel:

```
DCL 1 A,
      2 B,
        3 C;
D=A.B.;
```

PLR007 IN EXPRESSION NO VALID OPERAND AFTER OPERATOR

Erläuterung

Operator und Operand passen nicht zusammen.
Beispiel:

```
A = '123'+*'345';
```

PLR008 EXPRESSION STARTS WITH INVALID ITEM

Erläuterung

Ein Ausdruck beginnt mit einem ungültigen Zeichen.
Beispiel:

```
A = _123;    A = (**1+3)*4;
```

PLR009 INVALID STMT. KEYWORD

Erläuterung

Es wurde eine ungültige Anweisung gebraucht.
Beispiel:

```
PLOT POINT(1,2,3);
```

PLR010 END OF FILE BEFORE LOGICAL END OF PROGRAM

Erläuterung

Bevor das Programm logisch zu Ende war, wurde das Ende der Eingabedaten erreicht.
Beispiel:

```
CLOSE FILE(SYSPRINT);
  End of Dataset
```

Im Unterschied zu Fehler Nr.PLR001 ist die letzte Anweisung vollständig.

PLR011 QQPREF (or QQPLAB): STMT.PREFIX LIST OR LABEL LIST LONGER THAN 250 CHARS

Erläuterung

Der Bereich zur Aufnahme der 'prefixes' oder 'labels' einer Anweisung umfaßt nur 250 Zeichen.

PLR012 LABEL REFERENCED BY 'END'-STMT. NOT KNOWN. LABEL IGNORED

Erläuterung

Eine in der 'END'-Anweisung angegebene Marke ist (im betreffenden Block)nicht bekannt.
Beispiel:

```
A: PROC;
B: DO I=1 TO 10;
     J=I;
   END C;  /* C nicht gefunden */
```

PLR013 INVALID KEYWORD IN 'END'-STMT

Erläuterung

Es wurde ein ungültiges Wort in der 'END'-Anweisung angegeben, zum Beispiel in der Absicht, zwei 'DO'-Gruppen oder 'BEGIN'-Blöcke zu schließen.
Beispiel:

```
A: BEGIN;
   B: DO I=1 TO 10;
         J=I;
      END A B;
```

PLR014 ACTUAL BLOCK NO. blk_no NOT FOUND. PROBABLY BLOCK IS NOT ACTIVE

Erläuterung

Es wurde versucht, über dessen Nummer auf einen Block zuzugreifen, der nicht 'aktiv' ist. Das ist ein Precompiler-Fehler und darf nicht vorkommen.
Beispiel:

```
Block Nr.: 1 A: PROC;
           2 B: BEGIN;
             END B;

             hier wird aus irgendwelchen Gründen
             Block Nr.2 gesucht.
```

PLR015 NAME OF BLOCK TO BE SEARCHED IS BLANK

Erläuterung

Der 'Blocksuchroutine' wurde als Blockname eine 'blank' Zeichenkette übergeben. Precompiler-Fehler!

PLR016 IN EXPRESSION ')' MISSING OR MISPLACED

Erläuterung

')' an der falschen Stelle oder fehlend.
Beispiel:

```
A = (B+C+D,
```

PLR017 IN EXPRESSION AFTER INDEXLIST OR ARGUMENTLIST ')' MISSING OR MISPLACED

Erläuterung

')' an der falschen Stelle oder fehlend in Index- oder Argumentliste.
Beispiel:

```
A = I(B,C;   CALL D(A,B,C;
```

PLR023 HIGH BLOCK NO. blk_no NOT FOUND. PROBABLY BLOCK IS NOT ACTIVE

Erläuterung

Beim Eintragen eines neuen Blocks wurde als Nummer des Oberblocks eine angegeben, die entweder nicht vorhanden ist oder auf einen Block zugreift, der gar nicht Oberblock ist. Precompiler-Fehler!

PLR027 ERROR IN 'IF'-STMT

Erläuterung

Fehler in der 'IF'-Anweisung. Zum Beispiel im logischen Ausdruck oder 'THEN' vergessen.
Beispiel:

```
IF A *  THEN GOTO B;
IF A = B  GOTO C;
```

PLR028 MAJOR STRUCTURE LEVEL ASSUMED TO BE ONE

Erläuterung

Ein Element, das nicht aus einer 'substructure' ist, muß als Level 1 oder 0 haben.
Beispiel:

```
DCL 2 A, 3 B;
```

PLR036 DUPLICATE DECLARATION OF NAME 'name'

Erläuterung

Doppelte Deklaration von 'name'.
Beispiel:

```
DCL  A;
DCL  1 A, 2 B;
```

PLR040 Version 1

IDENTIFIER NO. inr NOT KNOWN IN BLOCK NO. bnr

Erläuterung

Die angegebene Nummer inr eines Namens ist im Block bnr nicht bekannt. Precompiler-Fehler!

Version 2

IDENTIFIER 'identifier' NOT KNOWN IN BLOCK NO. bnr

Erläuterung

Der angegebene Name ist im Block bnr nicht bekannt.

PLR048 IDENTIFIER 'identifier' SPECIFIES MORE LEVELS THAN PERMISSIBLE

Erläuterung

Der angegebene Name ist zu oft qualifiziert. Zur Zeit sind 14 Qualifizierungen erlaubt. Das heißt, es dürfen höchstens 14 Punkte enthalten sein.
Beispiel:

```
A.B.C.D.E.F.G.H.I.J.K.L.M.N.O (noch erlaubt!)
```

PLR058 IDENTIFIER 'identifier' AMBIGUOUS IN ACTUAL BLOCK

Erläuterung

Der angegebene Name ist Teil einer 'structure'. Die Qualifizierung ist jedoch nicht ausreichend, um festzustellen, aus welcher.
Beispiel:

```
DCL 1 A, 2 C, 3 D;
DCL 1 B, 2 C, 3 D, 4 E;
C.D = 10;   richtig: A.C.D oder A.D
```

Die Nachricht erscheint auch bei ausreichender, aber nicht vollständiger Qualifizierung.
Beispiel:

```
C.D.E = 10;  reicht zwar aus, aber C und auch C.D
             sind nicht eindeutig.
Abhilfe: B.C.D.E  oder  B.C.E  oder B.D.E oder B.E .
```

PLR066 ERROR IN DECLARE STMT., VARIABLE NAME OR STRUCTURE LEVEL MISSING OR WRONG. NEXT ITEM: 'item'

Erläuterung

Fehler in der 'DECLARE'-Anweisung. Der 'structure level' oder der Variablenname fehlen oder sind falsch.

PLR067 ERROR IN 'DECLARE STMT. ',' OR ';' MISSING. SCANPOINTER AT 'string'

Erläuterung

Fehler in der 'DECLARE'-Anweisung. Komma oder Strichpunkt fehlt.

PLR068 ERROR IN 'DECLARE STMT. ')' MISSING. SCANPOINTER AT 'string'

Erläuterung

Fehler in der 'DECLARE'-Anweisung. 'Klammer zu' fehlt.
Beispiel: DCL (A;

PLR072 ERROR IN ARGUMENT LIST. ARGUMENT NAME MISSING OR WRONG

Erläuterung

Fehler in der Argumentliste einer 'PROCEDURE'-Anweisung. Argumentname fehlt oder ist falsch.
Beispiel:

```
A: PROC( 1);   B: PROC( );
```

PLR073 ERROR IN ARGUMENT LIST. KOMMA OR ')' MISSING OR WRONG

Erläuterung

Fehler in der Argumentliste einer 'PROCEDURE'-Anweisung. Komma oder 'Klammer zu' falsch oder fehlend.
Beispiel:

```
A: PROC(B (,,));
```

PLR074 WRONG KEYWORD ON 'PROCEDURE'-STMT.

Erläuterung

Falsches Schlüsselwort in der 'PROCEDURE'-Anweisung.
Beispiel:

```
A: PROC MEIER;
```

PLR075 ERROR IN 'OPTIONS'-LIST ON 'PROCEDURE'-STMT. OPTION MISSING OR WRONG

Erläuterung

Fehler in der 'OPTIONS'-Liste bei der 'PROCEDURE'-Anweisung.
Keine oder falsche Option.
Beispiel:

```
A: PROC OPTIONS;
B: PROC OPTIONS(MEIER);
```

PLR076 ERROR IN 'PROCEDURE'-STMT. ')' MISSING

Erläuterung

Fehler in der 'PROCEDURE'-Anweisung. 'Klammer zu' fehlt.
Beispiel:

```
A: PROC OPTIONS(MAIN;
```

PLR077 ERROR IN 'OPTIONS'-LIST ON 'PROCEDURE'-STMT. KOMMA OR ')' MISSING

Erläuterung

Fehler in der 'OPTIONS'-Liste bei der 'PROCEDURE'-Anweisung.
Komma oder 'Klammer zu' fehlt.

PLR078 IN DECLARATION OF IDENTIFIER 'identifier' MISSING ATTRIBUTE(S) ASSUMED. ATTRIBUTE(S): 'attribute list'

Erläuterung

Fehlende Attribute hinzugefügt. Die Attribute sind:
'attributliste'.

PLR079 IN DECLARATION OF IDENTIFIER 'identifier' CONFLICTING ATTRIBUTE(S) IGNORED. ATTRIBUTE(S): 'attribute list'

Erläuterung

Im Widerspruch stehende Attribute ignoriert. Die Attribute sind: 'attributliste'.

PLR080 IN DECLARATION OF IDENTIFIER 'identifier' ADJUSTABLE DIMENSION SPECIFICATION 'dimension' INVALID FOR 'DYNAMIC' VARIABLE. REPLACED BY '10'

Erläuterung

Die Dimensionierung einer 'DYNAMIC' Variablen muß zur Precompilezeit bekannt sein. Wenn das nicht der Fall ist, wird angenommen, sie sei zehn.
Beispiel:

```
DCL A(N) DYNAMIC;
```

wird behandelt wie (und umgewandelt in)

```
DCL A(10) DYNAMIC;
```

PLR081 IN DECLARATION OF IDENTIFIER 'identifier' ADJUSTABLE PRECISION OR LENGTH SPECIFICATION 'length or precision' INVALID FOR 'DYNAMIC' VARIABLE. REPLACED BY '10'

Erläuterung

Die Längen- bzw. Genauigkeitsangabe einer 'DYNAMIC'-Variablen muß zur Precompilezeit bekannt sein. Wenn das nicht der Fall ist, wird angenommen, sie sei zehn.
Beispiel:

```
DCL A CHAR(N) DYNAMIC;
```

wird zu `DCL A CHAR(10) DYNAMIC;`

PLR082 IN DECLARATION OF IDENTIFIER 'identifier' ILLEGAL PRECISION SPECIFICATION 'length or precision' REPLACED BY DEFAULT PRECISION

Erläuterung

Die angegebene Genauigkeitsforderung ist nicht erlaubt. Sie wurde durch den Standardwert ersetzt.
Beispiel:

```
DCL N BIN FIXED(51);
```

wird zu `DCL N BIN FIXED; (zur Zeit 15)`

PLR083 IN DECLARATION OF IDENTIFIER 'identifier' CONVERSION ERROR IN CONVERTING LENGTH OR PRECISION SPECIFICATION 'length or precision'. THIS IS PRECOMPILER ERROR. PRECOMPILATION CONTINUES. LENGTH OR PRECISION TREATED AS NOT CONSTANT

Erläuterung

Umwandlungsfehler beim Interpretieren einer Längen- oder Genauigkeitsangabe, die als 'konstant' erkannt wurde, obwohl das nicht der Fall ist. Die Precompilation läuft trotz des Precompilerfehlers weiter, nachdem die Angabe jetzt als 'nicht konstant' gekennzeichnet wurde.
Beispiel:

```
DCL A CHAR(10+2) DYNAMIC;
```

Dies führt zuerst zu Fehler Nummer 83, dann zu 81 (wegen 'DYNAMIC').

PLR084 IN DECLARATION OF IDENTIFIER 'identifier' CONVERSION ERROR IN CONVERTING DIMENSION SPECIFICATION 'dimension'. THIS IS PRECOMPILER ERROR. PRECOMPILATION CONTINUES. DIMENSION TREATED AS NOT CONSTANT

Erläuterung

Umwandlungsfehler beim Interpretieren einer Dimensionsangabe. Begründung und Korrektur siehe PLR083.
Beispiel:

```
DCL A(10+2) DYNAMIC;
```

Dies führt wegen 'DYNAMIC' nach der Korrektur zu Fehler Nr.80.

PLR085 Version 1

IN DECLARATION OF IDENTIFIER 'identifier' 'PICTURE' SPECIFICATION MISSING OR INVALID. TREATED AS NOT CONSTANT. MAPPING IMPOSSIBLE

Erläuterung

Die 'PICTURE'-Spezifikation fehlt. Sie wird als 'nicht konstant' betrachtet. Dadurch ist die Länge der Variablen nicht feststellbar.
Beispiel:

```
DCL A PIC;
```

PLR085 Version 2

IN DECLARATION OF IDENTIFIER 'identifier' 'PICTURE' SPECIFICATION MISSING OR INVALID. PICTURE SPECIFICATION IS: 'string' TREATED AS NOT CONSTANT. MAPPING IMPOSSIBLE

Erläuterung

Die angegebene 'PICTURE'-Spezifikation 'string' ist ungültig. Sie wird als 'nicht konstant' betrachtet. Dadurch ist die Länge der Variablen nicht feststellbar.
Beispiel:

```
DCL A PIC 'ABCD';
```

PLR086 IN DECLARATION OF IDENTIFIER 'identifier' 'CONTROLLED' ATTRIBUTE NOT SUPPORTED FOR VARIABLE THAT IS NOT PARAMETER IN PROCEDURE WITHOUT 'MAIN' OPTION. REPLACED BY 'BASED'

Erläuterung

In Programmen ohne 'MAIN'-Option ist das Attribut 'CONTROLLED' nur bei Parametern erlaubt. Steht es dennoch, so wird es durch 'BASED' ersetzt.
Beispiel:

```
A: PROC;
   DCL B CONTROLLED;
```

erlaubt:

```
A: PROC OPTIONS(MAIN);
```

bzw.

```
   A : PROC(B);
```

PLR087 IN DECLARATION OF IDENTIFIER 'identifier' MISSING ATTRIBUTE 'VARIABLE' INSERTED IN PROCEDURE WITHOUT 'MAIN' OPTION

Erläuterung

In Programmen ohne 'MAIN'-Option ist bei manchen Datentypen das Attribut 'VARIABLE' erforderlich. Wenn es nicht steht, wird es hinzugefügt.
Beispiel:

```
A: PROC;
   DCL B FILE;
```

PLR088 DURING PASS 2 READING BEYOND END OF STMT. THIS IS PRECOMPILER ERROR

Erläuterung

Im zweiten Pass wird über das Ende einer Anweisung hinaus gelesen. Das ist ein Precompilerfehler und darf nicht vorkommen.

PLR089 ERROR IN EXPRESSION.IDENTIFIER MISSING OR MISPLACED

Erläuterung

Fehler im Ausdruck. Element steht falsch oder fehlt.
Beispiel:

```
A = B C;    A = B+;
```

PLR090 ERROR IN EXPRESSION OR ';' MISSING IN CALL OR LINK STMT

Erläuterung

Fehler im Ausdruck oder Strichpunkt fehlt bei der 'LINK'- oder 'CALL'-Anweisung.
Beispiel:

```
LINK A(B C);
CALL A(B+C) X;
```

PLR091 ERROR IN EXPRESSION, ')' MISSING OR MISPLACED

Erläuterung

Fehler im Ausdruck. 'Klammer zu' falsch oder fehlend.
Beispiel:

```
A = (B+C))*D;    A = (B+C*D;
```

PLR092 AFTER 'CALL' OR 'LINK' NO VALID ENTRY REFERENCE FOUND

Erläuterung

Nach 'CALL' oder 'LINK' kein gültiger 'ENTRY'-Datentyp.
Beispiel:

```
A: PROC;
   CALL E;  (E ist standard, d.h. DEC FLOAT(6))
richtig:  A: PROC;
   DCL E ENTRY EXT;
```

PLR093 AFTER 'LOAD' NO VALID 'DYNAMIC ENTRY' REFERENCE FOUND

Erläuterung

Nach 'LOAD' steht kein 'DYNAMIC ENTRY'.
Beispiel:

```
A: PROC;
   DCL B ENTRY;
   LOAD B;
```

PLR094 IN RELEASE STMT NO VALID ENTRY REFERENCE GIVEN

Erläuterung

In der 'RELEASE'-Anweisung steht keine gültige 'ENTRY'-Referenz.

PLR095 IN REORG MODULE STMT ';' MISSING OR MISPLACED

Erläuterung

In der 'REORG MODULE'-Anweisung Strichpunkt falsch oder fehlend.
Beispiel:

```
REORG MODULE
REORG MODULE A B;
```

PLR096 IN REORG MODULE STMT NO VALID PRIORITY GIVEN

Erläuterung

In der 'REORG MODULE'-Anweisung ist die angegebene Priorität ungültig.

PLR102 ERROR IN DYNAMIC SUBARRAY-INDEXLIST, ',' OR ')' MISSING OR MISPLACED

Erläuterung

In der Indexliste eines 'DYNAMIC SUBARRAYS' stehen Komma oder 'Klammer zu' falsch bzw. fehlen.
Beispiel:

```
B = A(1,2)(3 4);   B = A(1,2)(3,,4);
B = A(1,2)(3,4;    B = A(1,2)(3,4));
```

PLR103 ERROR IN DYNAMIC ELEMENT REFERENCE, AFTER '->' IDENTIFIER MISSING

Erläuterung

Bei der Referenz eines 'DYNAMIC'-Elements fehlt der Name nach '->'.

PLR104 ERROR IN DYNAMIC-INDEXLIST, ',' OR ')' MISSING OR MISPLACED

Erläuterung

In der 'DYNAMIC'-Indexliste fehlen Komma oder 'Klammer zu' bzw. stehen falsch.

PLR105 DYNAMIC-ARRAY REFERENCE WITHOUT DESCRIPTOR QUALIFICATION REFERS TO DYNAMIC-ARRAY WITHOUT ASSOCIATED DESCRIPTOR

Erläuterung

Bei der Referenz auf einen Dynamic Array wurde kein Descriptor angegeben, obwohl auch im 'DECLARE' keiner assoziiert wurde. Beispiel:

```
DCL A DYNAMIC;          DCL B BASED;
A = 5;                  B = 5;
```

Bei 'BASED' ist es das gleiche!

PLR106 IN 'statement' STMT: EXPRESSION AFTER 'option' OPTION OF ILLEGAL TYPE. REPLACED BY DEFAULT VALUE IF ANY. ELSE STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

Der Ausdruck hinter einer Option ist von unerlaubtem Typ. Sofern für diese Option ein Standardwert vorgegeben ist, wird dieser angenommen. Ansonsten wird die Anweisung ignoriert. Beispiel:

```
MESSAGE COUNT(A+B);
```

Hier muß eine Element-Referenz stehen (die wiederum vom geforderten Typ ist!).

PLR107 IN 'statement' STMT: EXPRESSION AFTER 'option' OPTION INVALID. SCANPOINTER AT 'string'. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

Nach einer Option ist der angegebene Ausdruck in sich ungültig.
Beispiel: MESSAGE COUNT(A+);

PLR108 IN 'statement' STMT: INVALID OPTION SPECIFIED BEGINNING 'string'. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

Es wurde eine Option verwendet, die bei der entsprechenden Anweisung nicht zugelassen ist.

PLR109 IN 'statement' STMT: 'option' OPTION MISSING. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

Die Option 'option' muß in der entsprechenden Anweisung stehen. Dies ist jedoch nicht der Fall.

PLR110 IN 'statement' STMT: COMMA MISSING IN EXPRESSIONLIST AFTER 'option' OPTION. SCANPOINTER AT 'string'. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

In der Liste von Ausdrücken, die hinter einer Option stehen, fehlt ein Komma.
Beispiel: STORE ..... TO(1,2 3);

PLR111 IN 'statement' STMT: ')' MISSING. SCANPOINTER AT 'string'. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

Es fehlt eine 'Klammer zu'.
Beispiel:

MESSAGE COUNT(M;

PLR112 IN 'statement' STMT: '(' MISSING. SCANPOINTER AT 'string'. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

Es fehlt eine 'Klammer auf'.
Beispiel:

```
MESSAGE COUNT A);
```

PLR113 IN 'statement' STMT: INVALID PASSWORD TYPE BEGINNING 'password_type'. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

In der Angabe, welche Operation durch welche Passwörter geschützt ist, wurde eine falsche Operation angegeben.
Beispiel:

```
INITIATE BANK(B)  PASSWORD(ALL=X,
    WP=Y, LOCK=Z);   (LOCK ist falsch)
```

PLR114 IN 'statement' STMT: EXPRESSION MISSING AFTER 'option' OPTION. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

Nach der angegebenen Option fehlt der erwartete Ausdruck.
Es ist kein Standardwert vorhanden.
Beispiel:

```
MESSAGE COUNT;
```

PLR115 IN 'statement' STMT: EXPRESSION LIST TOO LONG AFTER 'option' OPTION. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

Nach der 'option' Option folgt eine zu lange Liste von Ausdrücken.
Beispiel: `stmt  OPTION(A,B,C,D);`

Es sind nur 3 statt 4 Ausdrücke erlaubt.

PLR116 'IF-THEN-ELSE' TOO DEEPLY NESTED

Erläuterung

Die 'IF'-Anweisung ist zu stark geschachtelt.

PLR117 ERROR IN DYNAMIC-ELEMENT REFERENCE, AFTER '->' NO DYNAMIC-ELEMENT SPECIFIED

Erläuterung

Bei der erforderlichen Dynamic Element Referenz ist das Element nach '->' nicht 'DYNAMIC'.

PLR118 IN 'LOAD STMT' SEMICOLON MISSING OR MISPLACED

Erläuterung

In der 'LOAD'-Anweisung fehlt ein Semikolon oder steht falsch.
Beispiel:

```
LOAD A B;  LOAD;
```

PLR119 STMT MAY CONTAIN INVALID PLR-VARIABLE REFERENCE

Erläuterung

In der Anweisung wird eine ungültige Referenz auf eine PLR-Variable verwendet.
Beispiel:

```
DCL L DYNAMIC;
ALLOC L;
```

PLR120 IN 'statement' STMT: 'option' OPTION SPECIFIED AGAIN. PREVIOUS SPECIFICATION IGNORED

Erläuterung

Dieselbe Option wurde erneut angegeben. Die alte wird gänzlich ignoriert zusammen mit dem möglicherweise dahinter angegebenen Ausdruck bzw. der Ausdruckliste.
Beispiel:

```
MESSAGE COUNT(M) COUNT(N);
```

Wirksam ist 'N'.

PLR121 ')' MISSING AFTER 'option' OPTION IN 'GET' OR 'PUT' STMT. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

In der 'GET' oder 'PUT'-Anweisung fehlt eine 'Klammer zu' nach der 'option' Option.
Beispiel:

```
PUT FILE(SYSPRINT LIST(A);
```

PLR122 FORMAT LIST MISSING AFTER 'EDIT' DATA LIST IN 'GET' OR 'PUT' STMT. STATMENT OR REST OF STATEMENT IGNORED

Erläuterung

In der 'GET' oder 'PUT'-Anweisung fehlt die Format-Angabe nach der 'EDIT'-Datenliste.
Beispiel:

```
PUT FILE(SYSPRINT) EDIT(A);
```

PLR123 ')' MISSING AFTER FORMAT LIST IN 'GET' OR 'PUT' STMT. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

In der 'GET' oder 'PUT'-Anweisung fehlt eine 'Klammer zu' nach der Formatliste.
Beispiel:

```
GET FILE(SYSIN) EDIT(A) (F(5);
```

PLR124 IN FORMAT LIST IN 'GET' OR 'PUT' STMT ')' MISSING AFTER ITERATION FACTOR EXPRESSION. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

In der 'GET' oder 'PUT'-Anweisung fehlt die 'Klammer zu' hinter einem Wiederholungsfaktor in der Formatliste.
Beispiel:

```
PUT FILE(SYSPRINT) EDIT(A) ((80 A(1));
```

PLR125 IN FORMAT LIST IN 'GET' OR 'PUT' STMT INVALID FORMAT ITEM 'item' FOUND. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

In der 'GET' oder 'PUT'-Anweisung wurde in der Formatliste ein ungültiges Formatelement namens 'item' verwendet.
Beispiel:

```
PUT FILE(SYSPRINT) EDIT(A) (Ø(10),F(5));
```

Ø ist ungültig.

PLR126 IN FORMAT LIST IN 'GET' OR 'PUT' STMT ')' MISSING AFTER NESTED FORMAT LIST. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

In der 'GET' oder 'PUT'-Anweisung fehlt eine 'Klammer zu' in einer geschachtelten Formatliste.
Beispiel:

```
PUT FILE(SYSPRINT) EDIT(A,B,C.D) (2(F(5), E(6));
```

PLR127 IN FORMAT LIST IN 'GET' OR 'PUT' STMT ')' MISSING AFTER FORMAT SPECIFICATION. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

In der 'GET' oder 'PUT'-Anweisung fehlt eine 'Klammer zu' nach einer Formatangabe.
Beispiel:

```
GET FILE(SYSIN) EDIT(A) (X(1, F(5));
```

PLR128 IN 'GET' OR 'PUT' STMT ERROR IN EXPRESSION AFTER 'option' OPTION. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

In der 'GET' oder 'PUT'-Anweisung wurde ein Fehler in dem der 'option' Option zugehörigen Ausdruck gefunden.
Beispiel:

```
PUT FILE(SYSPRINT) LIST(A+B+);
```

PLR129 IN FORMAT LIST IN 'GET' OR 'PUT' STMT ERROR IN EXPRESSION IN 'format' FORMAT SPECIFICATION. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

In der 'GET' oder 'PUT'-Anweisung wurde in der Formatliste bei einer Formatangabe ein ungültiger Ausdruck verwendet.
Beispiel:

```
GET FILE(SYSIN) EDIT(A) (F(I+));
```

PLR130 WARNING, OPTIMIZING COMPILER MAY NOT ACCEPT GENERATED VALID PL/1-CODE

Erläuterung

Der erzeugte, gültige PL/1-Code wird möglicherweise vom Optimizing Compiler nicht verstanden.

PLR131 IN FORMAT LIST INVALID 'P' FORMAT ITEM. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

In einer Formatliste wurde eine ungültige Picture-Spezifikation verwendet.
Beispiel:

```
A: FORMAT(P(ABC));
```

PLR132 INVALID OPTION 'option' IN 'GET' OR 'PUT' STMT. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

In der 'GET' oder 'PUT'-Anweisung wurde ein ungültige Option verwendet.
Beispiel:

```
GET FIL(SYSIN) LIST(A);
```

(versehentlich 'FIL' statt 'FILE')

PLR133 ERROR IN 'statement' STMT IN EXPRESSION AFTER 'option' OPTION

Erläuterung

In dem zur Option 'option' gehörenden Ausdruck wurde ein Fehler gefunden.

PLR134 IN 'statement' STMT INVALID OPTION 'option'

Erläuterung

Es wurde eine ungültige Option verwendet.

PLR135 IN 'statement' STMT AFTER 'option' OPTION '(' MISSING

Erläuterung

Nach der 'option' Option fehlt eine 'Klammer auf'.

PLR136 IN 'statement' STMT AFTER 'option' OPTION ')' MISSING

Erläuterung

Nach der 'option' Option fehlt eine 'Klammer zu'.

PLR137 IN 'RETURN' STMT '(' MISSING AFTER 'RETURN'

Erläuterung

In der 'RETURN'-Anweisung fehlt eine 'Klammer auf'.
Beispiel:

```
          RETURN A);
```

PLR138 IN 'RETURN' STMT ERROR IN 'RETURN' EXPRESSION

Erläuterung

In der 'RETURN'-Anweisung wurde ein Fehler in dem zurückzugebenden Ausdruck entdeckt.
Beispiel:

```
          RETURN (A-);
```

PLR139 IN 'RETURN' STMT ')' MISSING AFTER 'RETURN' EXPRESSION

Erläuterung

In der 'RETURN'-Anweisung wurde eine 'Klammer zu' vergessen.
Beispiel:

```
RETURN (A;
```

PLR140 INVALID ITEM IN ASSIGNMENT STMT STARTING: 'item'

Erläuterung

In einer Zuweisungsanweisung steht etwas nicht Erlaubtes.
Beispiel:

```
A=B,C;
```

PLR141 IN 'RELEASE STMT' SEMICOLON MISSING OR MISPLACED

Erläuterung

In der 'RELEASE'-Anweisung steht ein Semikolon falsch oder es fehlt.
Beispiel: RELEASE; RELEASE A B;

PLR143 IN 'DECLARE STMT' DIMENSION, PRECISION, LENGTH, OR PARANTHESIZED OPTION IS TOO LONG AND HAS BEEN TRUNCATED

Erläuterung

In der 'DECLARE'-Anweisung ist der für Dimension, Länge, Genauigkeit oder sonst bei einem Attribut in Klammern stehende Ausdruck zu lang und wurde entsprechend gekürzt.

PLR144 IN 'GET' OR 'PUT' STMT. ')' MISSING OR MISPLACED IN DATA LIST AFTER IMPLICIT DO-LOOP

Erläuterung

In einer 'GET' oder 'PUT'-Anweisung wurde nach einem impliziten 'DO'-Loop eine 'Klammer zu' vergessen oder falsch gesetzt.
Beispiel:

```
GET FILE(SYSIN) LIST((A(I) DO I=1 TO 10);
```

PLR145 RECEIVING VALUE IN ASSIGNMENT STMT IS DYNAMIC ENTRY CONSTANT

Erläuterung

Links vom Gleichheitszeichen steht eine Dynamic Entry Konstante (eine Konstante darf nie links vom Gleichheitszeichen stehen).

PLR146 RECEIVING VALUE IN ASSIGNMENT STMT IS DESCRIPTOR

Erläuterung

Links vom Gleichheitszeichen steht ein Deskriptor.

PLR147 IN ASSIGNMENT STMT INVALID USE OF DYNAMIC ARRAY DESCRIPTOR

Erläuterung

In einer Zuweisungsanweisung wird ein Deskriptor in ungültiger Weise verwendet.
Beispiel:

```
DCL(E,D) DESC;   DCL A DYNAMIC;
   D= ADDR(E->A);
```

PLR148 AFTER 'DISPLAY' CHARACTER-STRING EXPRESSION MISSING

Erläuterung

In der 'DISPLAY'-Anweisung fehlt der Zeichenkettenausdruck.

PLR149 AFTER CHARACTER-STRING EXPRESSION IN 'DISPLAY' STMT ')' MISSING

Erläuterung

In der 'DISPLAY'-Anweisung fehlt nach dem Zeichenkettenausdruck eine 'Klammer zu'.

PLR150 IN 'statement' STMT: SEMICOLON MISSING. SCANPOINTER AT 'string' STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

Es fehlt ein Semikolon (obwohl die Anweisung bereits vollständig ist).

PLR151 IN 'RETURN' STMT ';' MISSING OR MISPLACED. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

In der 'RETURN'-Anweisung fehlt ein Semikolon oder steht falsch.

PLR152 INVALID USE OF PLR-VARIABLE IN 'DO' STMT. REPLACED BY NONITERATIVE 'DO'

Erläuterung

In einer 'DO'-Anweisung wurde eine PLR-Variable falsch verwendet. Aus der iterativen 'DO'-Gruppe wurde eine einfache gemacht.

PLR153 DATASET 'SUBLIB' COULD NOT BE OPENED FOR MEMBER 'member'

Erläuterung

Der Common des Subsystems konnte nicht gefunden werden. Zum Beispiel wegen eines I/O-Errors, weil die DD-Karte für 'SUBLIB' fehlt oder fehlerhaft ist, weil die referierte Datei zerstört ist.

PLR154 BLOCKSIZE OF 'SUBLIB' DATASET TOO LARGE FOR INCLUDING COMMON INTO MODULE

Erläuterung

Die Blocksize der referierten Datei ist zu groß. Sie muß zur Zeit kleiner als 3201 sein.

PLR155 I/O ERROR WHEN READING COMMON FROM FILE 'SUBLIB'

Erläuterung

Beim Lesen des Commons trat ein I/O-Error auf.

PLR160 IN 'statement' STMT: MISSING COMMA ASSUMED. SCANPOINTER AT 'string'

Erläuterung

Ein fehlendes Komma wurde als vorhanden angenommen.
Beispiel:

```
DEFINE A(1,2)(3,2)(4,5,6);
```

PLR161 IN 'statement' STMT: MISSING 'STRUCTURE' OR 'POOLTABLE' OPTION. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

Wenigstens eine der beiden Optionen muß stehen. Es wurde aber keine der beiden angegeben.

PLR162 IN 'statement' STMT: NOT ANY OPTION SPECIFIED. STMT IGNORED

Erläuterung

Wenigstens eine der möglichen Optionen muß stehen. Es wurde aber keine angegeben.

PLR163 'MOD' CONFLICTS WITH 'CATALOG'. STATEMENT OR REST OF STATEMENT IGNORED

Erläuterung

Die Optionen 'MOD' und 'CATALOG' stehen zueinander im Widerspruch.

PLR165 IN 'statement' STMT: INVALID SYNTAX '()' IN 'option' OPTION SPECIFICATION. DEFAULT VALUE(S) ASSUMED IF ANY

Erläuterung

Nach einer Option, die von einem geklammerten Ausdruck gefolgt ist (oder gefolgt sein kann), stehen nur die Klammern. Es wird so getan, als stünde zwischen den Klammern der Standardwert (sofern einer vorgegeben ist).
Beispiel:

```
MESSAGE TEXT();
```

Es wird "(Nullstring) übertragen.

PLR166 IN DECLARATION OF IDENTIFIER 'identifier' NAME 'name' SPECIFIED AFTER 'LIKE' NOT KNOWN IN THIS BLOCK

Erläuterung

Der nach dem 'LIKE'-Attribut genannte Name ist in dem betreffenden Block nicht bekannt.
Beispiel:

```
A: PROC;
   DCL ; C LIKE B;
END A;
```

PLR167 IN DECLARATION OF IDENTIFIER 'identifier' NAME 'name' SPECIFIED AFTER 'LIKE' IS NOT A MAJOR OR MINOR STRUCTURE NAME

Erläuterung

Der nach 'LIKE' angegebene Name identifiziert keine Struktur.
Beispiel:

```
A: PROC;
   DCL C;
   DCL 1 B LIKE C;
END A;
```

PLR168 IN DECLARATION OF IDENTIFIER 'identifier' WITH 'LIKE' ATTRIBUTE ADDITIONAL SUBSTRUCTURES OR ELEMENTARY NAMES ARE NOT ALLOWED

Erläuterung

Nach der Verwendung des 'LIKE'-Attributs dürfen nur noch gleich hohe oder höhere Strukturstufen angegeben werden.
Beispiel:

```
DCL 1 A,
      2 B,
        3 C LIKE D,
        3 E, erlaubt
        4 F LIKE D,  verboten
        4 G,  verboten
        2 H;  erlaubt
```

PLR169 IN DECLARATION OF IDENTIFIER 'identifier' MISSING NAME SPECIFICATION AFTER 'LIKE' ATTRIBUTE

Erläuterung

Nach dem 'LIKE'-Attribut wurde kein Name angegeben.
Beispiel:

```
DCL 1 A LIKE;
```

PLR170 IN DECLARATION OF IDENTIFIER 'identifier' ADJUSTABLE DIMENSION SPECIFICATION 'dimension' FOR 'DESCRIPTOR' WITH 'STATIC' OR 'BASED' ATTRIBUTE INHIBITS INITIALISATION WITH 'NULL()' VALUE

Erläuterung

Eine variable Dimensionsangabe für Deskriptoren, die das Attribut 'BASED' oder 'STATIC' führen, verhindert die Initialisierung mit 'NULL()' (weil die Dimension nicht berechnet werden kann).
Beispiel:

```
DCL 1 A BASED,
      2 L, 2 D(N REFER(A.L)) DESC;
```

PLR171 IN 'statement' STMT: LABEL REFERENCE IS NOT A LABEL CONSTANT AND MAY POINT OUT OF BLOCK

Erläuterung

In einer 'GOTO'-Anweisung ist das Sprungziel keine Sprungmarke, sondern eine Variable oder eine Funktionsprozedur, so daß nicht erkannt werden kann, ob das Sprungziel außerhalb des aktuellen Blocks liegt.
Beispiel:

```
A: PROC;
   DCL B EXT ENTRY(LABEL,LABEL,BIN FIXED(15))
       RETURNS(LABEL);
   DCL C LABEL VARIABLE;

D:E:;
BEGIN;
GOTO B(E,D,10);
C=D;
GOTO C;
```

PLR172 IN 'statement' STMT: CONTROL TRANSFER OUT OF BLOCK MAY CAUSE UNPREDICTABLE RESULTS

Erläuterung

Die entsprechende Anweisung führt zum Verlassen des aktuellen Blocks. Die möglicherweise erforderlichen Verwaltungsarbeiten vor dem Verlassen eines Blocks können nicht gemacht werden.

Beispiel:

```
BEGIN;
  DCL A DYNAMIC;
B:;
BEGIN;
  GOTO B;
END;
```

POL001 EOF ON INPUT FILE

Erläuterung

Ende der Eingabe-Datei P.SYSIN angetroffen, bevor das POL-Programm logisch zu Ende war. Abbruch der Übersetzung. (EOF = End of File, Ende der Eingabedatei)

POL005 STATEMENT TOO LONG, QUOTE (*/) INSERTED BEFORE next
POL005 STATEMENT TOO LONG, FIRST RECORD, record, DROPPED

Erläuterung

Eine POL- oder PL/1-Anweisung ist zu lang (mehr als 3500 Zeichen bzw. etwa 50 Karten). Möglicherweise wurde ein Apostroph oder Kommentarende vergessen. Tritt der Fehler in einer Zeichenkette oder einem Kommentar auf, wird ein Apostroph oder */ eingefügt (vor dem Semikolon oder der nächsten unpaarigen Klammer zu), sonst wird der erste Teil der Anweisung aus dem Eingabepuffer entfernt.
Dieser Fehler tritt auch auf, wenn in STATEMENT-Definitionen mehrere POL-Anweisungen nacheinander verarbeitet werden und ein Anruf von QQNEW_STATEMENT dazwischen fehlt.
'next' ist der Teil der Anweisung, vor den ein Apostroph oder */ eingefügt wurde.
'record' ist der Teil der Anweisung, der aus dem Eingabepuffer gelöscht wurde.

POL006 IN PROCEDURE NEXT_EXPRESSION IDENTIFIER MISSING AFTER "." OR "->" IN A QUALIFIED REFERENCE

Erläuterung

In einem Ausdruck fehlt nach "." oder "->" in einem qualifizierten Namen eine Benennung (Identifier).
Beispiel: A.+ B.5 oder P->+X.

POL007 IN PROCEDURE NEXT_EXPRESSION INVALID ITEM AFTER OPERATOR

Erläuterung

In einem Ausdruck steht nach einem Operator weder ein Name noch eine Konstante, noch ein geklammerter Ausdruck.
Beispiel: 3 + *4, A ?, C** A

POL008 IN PROCEDURE NEXT_EXPRESSION EXPRESSION STARTS WITH INVALID ITEM

Erläuterung

In einer Anweisung wird ein arithmetischer, logischer oder Zeichenkettenausdruck erwartet. Der Ausdruck fängt mit einem ungültigen Element an.
Beispiel: ?,)), ///

POL016 IN PROCEDURE NEXT_EXPRESSION EXPRESSION CONTAINS UNMATCHED PARENTHESIS

Erläuterung

Ein Audruck enthält eine unpaarige Klammer.
Beispiel: (A+(B-C)

POL017 IN PROCEDURE NEXT_EXPRESSION EXPRESSION CONTAINS UNMATCHED PARANTHESIS IN ARGUMENT OR SUBSCRIPT LIST

Erläuterung

Ein Ausdruck enthält eine unpaarige Klammer in einer Argumenten- oder Indexliste.
Beispiele: A(3,(4,5) oder B(X+Y,((B))

POL018 IN PROCEDURE NEXT_ITEM EOF ON INPUT FILE

Erläuterung

In der PLS-Funktion NEXT_ITEM wurde EOF auf SYSIN festgestellt. (EOF = End of File, Ende der Datei).

POL019 IN PROCEDURE NEXT_REAL EOF ON INPUT FILE

Erläuterung

Statt einer reellen Konstanten wurde das Ende der Eingabedatei angetroffen.

POL020 IN PROCEDURE NEXT_REAL (bzw. NEXT_INTEGER) NO REAL (bzw.INTEGER) NUMBER FOUND. NEXT ITEM IS: next

Erläuterung

In der PLS-Funktion NEXT_REAL oder NEXT_INTEGER wurde keine Realzahl bzw. Integerzahl gefunden. Es wird 0 zurückgegeben. 'next' ist das statt der Real- oder Integerkonstanten in der Anweisung stehende Element.

POL021 IN PROCEDURE NEXT_IDENTIFIER EOF ON INPUT FILE

Erläuterung

Statt einer Benennung (Identifier) wurde das Ende der Eingabedatei angetroffen.

POL022 IN PROCEDURE NEXT_BITSTRING (bzw.NEXT_STRING oder NEXT_OPERATOR) EOF ON INPUT FILE

Erläuterung

Statt einer Bitkette (bzw. einer Zeichenkette oder einem Operatoren) wurde das Ende der Eingabedatei angetroffen.

POL023 IN PROCEDURE NEXT_IDENTIFIER
(bzw.NEXT_BITSTRING
oder NEXT_STRING
oder NEXT_OPERATOR)
NO IDENTIFIER (bzw. BITSTRING oder CHARACTERSTRING
oder OPERATOR) FOUND

Erläuterung

In der PLS-Funktion NEXT_BIT, NEXT_STRING, NEXT_IDENTIFIER oder NEXT_OPERATOR wurde das verlangte Syntaxelement in der Eingabe nicht gefunden. Es wird '' für die Zeichen- oder Bitketten, der Nullstring Identifier oder ↓ als Operator zurückgegeben.

POL024 IN PROCEDURE NEXT_N MORE CHARACTERS REQUESTED THAN AVAILABLE BEFORE EOF

Erläuterung

In der PLS-Funktion NEXT_N waren weniger Zeichen bis zum EOF auf SYSIN vorhanden als angefordert wurden. Die vorhandenen werden zurückgegeben. (EOF = End of File, Ende der Eingabedatei).

POL025 IN PROCEDURE NEXT_WORD EOF ON IN_PUT FILE

Erläuterung

In der PLS-Funktion NEXT_WORD wurde EOF auf SYSIN festgestellt. (EOF = End of File, Ende der Eingabedatei).

POL026 IN PROCEDURE THIS_STATEMENT EOF ON INPUT FILE

Erläuterung

In der PLS-Funktion THIS_STATEMENT wurde EOF auf SYSIN festgestellt. (EOF = End of File, Ende der Eingabedatei).

POL027 ERROR IN PL/1-IF-STATEMENT

Erläuterung

Eine PL/1-IF-Anweisung ist fehlerhaft.
Beispiel: IF A GOTO B;
IF (3+2>5 THEN DO;

POL029 POL STATEMENT TO BE TREATED AS PL/1-STATEMENT, BUT NO PL/1- OR SYSTEM STATEMENT WITH THAT NAME FOUND. STATEMENT NAME:name

Erläuterung

Mittels der Anweisung PLI; sollte eine Anweisung als System- oder PL/1-Anweisung aufgefaßt werden, eine Anweisung dieses Typs gibt es jedoch nicht als System- oder PL/1-Anweisung. 'name' ist der Name der fehlerhaften Anweisung.
Die Anweisung wird ignoriert.
Beispiel: STA 'KEINPLI'; PLI; END STA;

POL030 IN PROCEDURE NEXT_WORD ";" FOUND, STATEMENT-END MAY BE SKIPPED

Erläuterung

In der PLS-Funktion NEXT_WORD wurde als erstes Zeichen ein Semikolon gefunden und zurückgegeben. Das Anweisungsende wird möglicherweise dadurch übersehen. Wahrscheinlich ist die Anweisung unvollständig.

POL031 STATEMENT NOT COMPLETELY PROCESSED, ";" ASSUMED. NEXT ITEM: next

Erläuterung

Eine POL-Anweisung wurde nicht vollständig abgearbeitet. Es wird angenommen, daß ein Semikolon fehlt und daß an dieser Stelle eine neue Anweisung beginnt. Ist diese Annahme falsch, erfolgt anschließend Fehlermeldung Nr.32. 'next' ist das folgende Element in der Eingabe.

POL032 STATEMENT NOT FOUND, WRONG KEYWORD OR DATATYPE. STATEMENT NAME: name

Erläuterung

POL-Anweisung dieses Namens oder dieses Datentyps ist nicht definiert. Die Anweisung wird in Kommentarzeichen eingeschlossen. 'name' ist der Name der fehlerhaften Anweisung.

POL033 LEVEL OF SUBSYSTEM NESTING GREATER THAN 50, PROCESSING TERMINATED

Erläuterung

Mehr als die zulässige Anzahl Subsysteme wurden geschachtelt. Die Grenze der Schachteltiefe beträgt 50 Subsysteme. Die Übersetzung wird abgebrochen.

POL039 IN REGENT-OPTIONLIST WRONG KEYWORD ENCOUNTERED. NEXT ITEM: option

Erläuterung

In der REGENT-Optionen-Liste auf der PROC-Anweisung wurde ein falsches Schlüsselwort gefunden. 'option' ist die fehlerhafte Option. Sie wird ignoriert.

POL044 CLAUSE NOT FOUND. CLAUSE NAME: name

Erläuterung

In einer Anweisungstreiberroutine wurde eine CLAUSE aufgerufen, deren Name im Subsystem nicht bekannt ist. 'name' ist der Name der CLAUSE.

POL047 SUBSYSTEM SPECIFIED IN REGENT-OPTION "SUBSYSTEM" NOT FOUND. NEXT ITEM: name

Erläuterung

In der REGENT-Option einer PROC-Anweisung wurde nach SUBSYSTEM= ein ungültiger Name angegeben. 'name' ist der Name des Subsystems.

POL048 IN ENTER STATEMENT SUBSYSTEM name NOT FOUND

Erläuterung

Das Subsystem mit dem Namen 'name', das mittels der ENTER-Anweisung eröffnet werden sollte, ist nicht vorhanden.

POL060 STATEMENT name NOT FOUND, name_1-STATEMENT ASSUMED

Erläuterung

Die Anweisung mit dem Namen 'name' wurde nicht gefunden,es wurde die Anweisung 'name_1' statt dessen angenommen. Diese Meldung wird dann vorkommen, wenn der Anfang von 'name' eine gültige Anweisung darstellt.
Beispiel: PUTLIST SKIP;
Meldung: STATEMENT PUTLIST NOT FOUND, PUT-STATEMENT ASSUMED

POL069 CHARACTER VALUE RETURNED BY PLS-FUNCTION function-name IS TOO LONG AND HAS BEEN TRUNCATED

Erläuterung

Eine PLS-Funktion soll einen CHARACTER-Wert zurück liefern. Der Characterstring ist zu lang und wird abgeschnitten.
Die PLS-Funktionen NEXT_ITEM, NEXT_REAL, NEXT_INTEGER, NEXT_BIT, NEXT_STRING, NEXT_OPERATOR, NEXT_IDENTIFIER, NEXT_N, NEXT_WORD und NEXT_EXPRESSION können eine maximal 250 Zeichen lange Zeichenkette verarbeiten, die Funktion THIS_ STATEMENT kann bis zu 3500 Zeichen zurückliefern. 'function name' ist der Name der PLS-Funktion.

RMM000 REGENT MODUL MANAGEMENT NOT ACTIVE

Erläuterung

Die Modulverwaltung im REGENT ist nicht initialisiert. In der REGENT-Option auf der PROC-Anweisung der Main-Procedure wurde NOMOD angegeben. Abbruch folgt.

RMM002 MODULE m NOT FOUND

Erläuterung

Der Modul m ist nicht in der Bibliothek, welche unter dem DD-Namen 'STEPLIB' angesprochen wurde. Der aufgerufene RMM-Entry kann nicht ausgeführt werden. Etwa erwartete Ergebnisse sind undefiniert.

RMM003 MODULE m NOT EXECUTABLE

Erläuterung

Der Modul m ist vom LINKAGE EDITOR als nicht ausführbar markiert. Dies ist in der SYSPRINT-Liste bei der Erzeugung des Moduls ersichtlich. Der aufgerufene RMM-Entry kann nicht ausgeführt werden. Etwa erwartete Ergebnisse sind undefiniert.

RMM004 INSUFFICIENT CORE FOR MODULE m, MODULE SIZE IS l BYTES

Erläuterung

Vor der Ausführung des Moduls m wurde festgestellt, daß er nicht mehr in den Arbeitsspeicher passen wird. Der aufgerufene RMM-Entry kann nicht ausgeführt werden. Etwa erwartete Ergebnisse sind undefiniert. Der Fehler kann nur durch Vergrößerung des Arbeitsspeichers behoben werden. l ist die Länge des Moduls, der geladen werden sollte in Bytes.

RMM005 RMM-ENTRY e NOT FOUND IN MODULE m

Erläuterung

Es soll der RMM-Entry e in dem Modul m aufgerufen werden, obwohl e nicht Bestandteil von m ist. Der aufgerufene RMM-Entry kann nicht ausgeführt werden. Etwa erwartete Ergebnisse sind undefiniert. Dies ist ein Anwendungsfehler und kann durch Änderung des Aufrufes beseitigt werden. Die RMM-Entry-Namen sind in der SYSPRINT-Liste bei der Erzeugung des Moduls zu finden.

RMM006 MODULE NAME m IS AN ALIAS NAME AND THE MODULE IS NOT RENT OR REUS

Erläuterung

Es soll der Modul m ausgeführt werden. Bei der Ausführung wurde festgestellt, daß m ein Alias-Name im OS-Sinne ist und daß dieser Modul nicht reentrant bzw. reusable deklariert ist. Dieser Fehler kann auch auftreten, wenn in dem erzeugten Modul Programme eingebunden werden, welche nicht RENT oder REUS sind. Siehe dazu die SYSPRINT-Liste bei der Erzeugung des Moduls. Der aufgerufene RMM-Entry kann nicht ausgeführt werden. Etwa erwartete Ergebnisse sind undefiniert.

RMM007 POOL OVERFLOW BY d BYTES

Erläuterung

Die angegebene Modulpoolgröße wurde um d Bytes überschritten.

RMM008 MODULE/MEMBER m DELETED

Erläuterung

Der Modul m wurde aus der Liste der inaktiven Module gestrichen und im Kernspeicher gelöscht.

RMM009 A CORE REORGANIZATION HAS BEEN INVOLVED

Erläuterung
Der angegebene Modulpool ist voll belegt. Es werden alle inaktiven Module daraus entfernt.

RMM010 LINK RMM-ENTRY e IN MODULE m

Erläuterung
Es soll der REGENT-Modul m dynamisch, mit Hilfe des OS-LINK-Makros, in den Arbeitsspeicher geladen und ausgeführt werden. Die Kontrolle wird an den RMM-Entry e übergeben.

RMM011 RETURN FROM m

Erläuterung
Die Ausführung des Moduls m wurde abgeschlossen.

RMM020 LOAD RMM-ENTRY e IN MODULE m

Erläuterung
Es soll der RMM-Entry e in Modul m dynamisch, mit Hilfe des OS-LOAD-Makros, in den Arbeitsspeicher geladen werden.

RMM021 MODULE m HAS BEEN LOADED

Erläuterung
Der REGENT-Modul m wurde mit Hilfe des OS-LOAD-Makros dynamisch in den Arbeitsspeicher geladen.

RMM022 MODULE m WAS ALREADY LOADED

Erläuterung
Der REGENT-Modul m befindet sich schon im Arbeitsspeicher.

RMM030 RELEASE RMM-ENTRY e IN MODULE m

Erläuterung
Es soll der RMM-Entry e im Modul m aus dem Arbeitsspeicher gelöscht werden.

RMM031 MODULE m RELEASED

Erläuterung
Der REGENT-Modul m wurde aus dem Arbeitsspeicher gelöscht.

RMM032 MODULE m NOT FOUND

Erläuterung
Es sollte der RMM-Entry e im Modul m aus dem Arbeitsspeicher gelöscht werden, obwohl er nicht dynamisch geladen worden war.

RMM043 MODULE m NOT PREVIOUSLY DELETED

Erläuterung
Bei Abschluß einer REGENT-Ausführung mit der FINISH-Anweisung wurde festgestellt, daß der Modul m aufgrund eines LOAD auf einen Modul m enthaltenen RMM-Entry ohne die zugehörige RELEASE-Anweisung noch im Arbeitsspeicher war. Der Modul wird gelöscht.

RMM050 PRESENT RMM POOL SIZE p BYTES

Erläuterung
Die angegebenen bzw. vom REGENT Laufzeitsystem angenommene Poolsize betrug p Bytes.

RMM051 MAXIMUM POOL USED FOR MODULES: p BYTES

Erläuterung
Maximaler Platzbedarf der während des Laufes im Speicher befindlichen RMM-Module in p Bytes.

RMM052 MINIMUM POOL NEEDED FOR MODULES: p BYTES

Erläuterung
Platzbedarf aller gleichzeitig aktiven RMM-Module in p Bytes. Dies ist die minimale Modul-Pool-Größe, die erforderlich ist.

RMM053 TOTAL LENGTH OF ALL MODULES USED p BYTES

Erläuterung
Gesamtlänge aller in einer REGENT-Ausführung verwendeten RMM-Module in p Bytes. Wenn die Poolgröße auf diesen Wert gesetzt wird, wird jeder Modul nur einmal von der Platte geholt.

RMM054 THE MOST OFTEN LINKED/LOADED MODULE/MEMBERS

Erläuterung
Nach dieser Überschrift folgt eine Liste aller RMM-Module, die während einer REGENT-Ausführung bemüht wurden. Die Liste ist nach der Häufigkeit der Benutzungen sortiert.

RMM055 THE MODULE/MEMBERS MOST OFTEN RETRIEVED FROM DISK:

Erläuterung
Nach dieser Überschrift folgt eine Liste aller RMM-Module, die während der Ausführung von der REGENT-Bibliothek geholt wurden. Die Liste ist nach der Häufigkeit der Plattenzugriffe sortiert.

RMM060 THE POOL SIZE HAS BEEN CHANGED FROM m_1 BYTES TO m_2 BYTES

Erläuterung
Die aktuelle Modulpoolgröße m_1 wurde während einer REGENT-Ausführung in die Poolgröße von m_2 geändert.

RMM061 REGION TOO SMALL FOR MODUL POOL.MODULE SIZE 1 BYTES

Erläuterung

Ein Modul mit der Länge 1 konnte nicht geladen werden, da zu wenig Speicherplatz vorhanden und obwohl die Modul-Pool-Größe ausreichen würde. Die Region ist also zu klein, um den Modul-Pool aufzunehmen. Es wird reorganisiert und ein erneutes Laden versucht. Sollte dies wieder mißlingen, so wird Meldung RMM004 erfolgen.

13.2 Nachrichten des Subsystems DABAL

Fehlermeldungen SUBSYSTEM DABAL

001 INVALID SYNTAX IN OPTION: op-name

Erläuterung

In der option op-name wurde die Syntax fehlerhaft spezifiziert, z.B. ein falsches oder vergessenes Schlüsselwort, fehlendes = ,) etc.

002 KEYWORD MUST BE FOLLOWED BY '=' OR ','

Erläuterung

In der Paßwortliste müssen die Paßworte durch Kommata getrennt werden bzw. mit '=' zugewiesen werden.

003 INVALID OR MISSING KEYWORD

Erläuterung

Es wurde ein Schlüsselwort nicht angegeben bzw. falsch angegeben (verschrieben). Häufig ist dies POOL, BANK, TO, FROM.

004 name = INVALID IDENTIFIER

Erläuterung

Der links vom Gleichzeitszeichen ausgedruckte Name ist innerhalb der Syntax ein unzulässiger Identifier oder Schlüsselwort.

005 INVALID IDENTIFIER IN STATEMENT: st-name

Erläuterung

Das Statement st-name enthält einen falschen Identifier.

006 MISSING OR WRONG OPTION: op1/op2/...

Erläuterung

Mindestens eine der Options op1,...opn muß im Statement spezifiziert werden, fehlt aber.

007 INVALID IDENTIFIER: name SKIPPED

Erläuterung

name ist ein falscher Identifier im Statement und wurde geskippt.

008 ARGUMENT LIST IN STATEMENT 'SPACE' CONTAINS MORE THAN 5 VARIABLES OR RIGHT PARENTHESIS MISSING. ARGUMENTS REPLACED BY DUMMY ARGUMENTS

Erläuterung

Wenn mehr als 5 Argumente bei Space angegeben werden oder die schließende Klammer fehlt, werden sie im Aufruf durch Standard-Dummy-Argumente ersetzt, was zu Fehlern führen kann.

009 INVALID STRING SKIPPED: string

Erläuterung

Die Zeichenkette string wurde als ungültig erkannt und übersprungen.

010 KEYWORD: name1 or name2... MISSING IN STATEMENT: st-name

Erläuterung

Im Statement st-name fehlt wenigstens eines der Schlüsselwörter name1,name2,...

011 CARD NO: nn TRUNCATED. STATEMENT: CHANGE CARD

Erläuterung

Bei der Änderung einer Zeichenkette wurde die Spalte 72 der Karte überschritten.

012 STRING: str NOT FOUND. STATEMENT st

Erläuterung

Im Statement st (z.B. CHANGE CARD, FIND) wurde der spezifizierte String str nicht im Deck gefunden.

013 OPENED BANK name SAVED. MAY LEAD TO DISASTREOUS ERRORS, IF BANK WAS OPENED FOR UPDATE OR OUTPUT

Erläuterung

Die geöffnete Bank mit dem Namen 'name' wurde gerettet (z.B. auf Band). Dies kann zu gravierenden Fehlern führen, falls mit der Bank im update/output mode gearbeitet wurde. Daten können noch in der Pufferverwaltung stehen, die noch nicht auf die Bank weggeschrieben sind.

014 CLOSED BANK name SAVED

Erläuterung

Selbsterklärend.

015 CHANGED: no ... CARD

Erläuterung

Die Karte mit der Nummer 'no' wurde geändert.

016 FOUND: no ... CARD

Erläuterung

Die Karte mit der Nummer 'no' wurde im Deck gefunden.

017 NO CARDS IN THIS POOL

Erläuterung

Der Pool ist leer.

018 BANK NOT OPENED

Erläuterung

Die Bank ist nicht geöffnet.

019 NO ACCESS RIGHT TO READ BANK/POOL

Erläuterung

Keine Berechtigung, die Bank/Pool zu lesen.

020 NO ACCESS RIGHT TO READ BANK PRIVILEGED

Erläuterung

Keine privilegierte Leseberechtigung für die Bank.

021 BANK OR POOL NOT OPENED, PASSWORDS UNCHANGED

Erläuterung

Der Pool oder die Bank sind nicht geöffnet. Paßwörter können nicht geändert werden. Evtl. falscher Pool angegeben.

022 NO ACCESS RIGHT TO CHANGE PASSWORDS

Erläuterung

Keine Berechtigung zum Ändern der Paßwörter.

023 BANK OR POOL NOT OPEN. NAME OR LENGTH NOT CHANGED

Erläuterung

Die Bank/Pool war nicht geöffnet. Der Name/Länge wurde nicht verändert.

024 NO ACCESS RIGHT TO WRITE.LENGTH NOT CHANGED

Erläuterung

Keine Schreibberechtigung. Länge unverändert.

025 NO ACCESS RIGHT TO CHANGE NAME

Erläuterung

Keine Berechtigung zur Namensänderung.

026 DATAPOOL NOT OPENED

Erläuterung

Selbsterklärend.

027 POOL: name NO DATAPOOL

Erläuterung

Der Pool mit Name name ist kein Datapool.

028 POOL EMPTY

Erläuterung

Selbsterklärend.

029 INVALID POOL REFERENCE OR POOL NOT OPENED

Erläuterung

Der angegebene Pool ist nicht offen bzw. falsch spezifiziert.

030 REFERENCED POOL NO DATAPOOL

Erläuterung

Der Pool ist kein Datapool.

031 OUTPUT POOL IN STATEMENT 'EXPAND PLI' TOO SMALL

Erläuterung

Der Outputpool im Statement 'Expand pli' ist zu klein. Mehr Space angeben.

032 DATAPOOL NOT COPIED

Erläuterung

Der Datapool wurde nicht kopiert.

033 FROM-POOL OR FROM-BANK NO VALID REFERENCE

Erläuterung

Die zu kopierende Datei/Bank ist ungültig, z.B. nicht geöffnet.

034 TO-POOL OR TO-BANK NO VALID REFERENCE

Erläuterung

Die Datei/Bank, in die kopiert werden soll, ist eine ungültige Referenz. Meistens nicht geöffnet!

035 NO VALID FATHER-SON RELATION: nam1,nam2

Erläuterung

Es wurde beim Kopieren eine nicht erlaubte Beziehung zwischen Input und Output-Datei angegeben (From, To). Die Dateien, um die es sich handelt, haben die Namen nam1 und nam2.

036 BANK OR POOL NOT COPIED

Erläuterung

Datei wurde nicht kopiert.

037 NO ACCESS RIGHT TO WRITE INTO BANK

Erläuterung

Keine Schreibberechtigung für die Bank.

038 INVALID POOL REFERENCE

Erläuterung

Kein gültiger Pool.

039 INVALID BLKSIZE SPECIFIED. BLKSIZE NOT CHANGED

Erläuterung

Beim Compress wurde eine falsche blksize angegben im Parameter NEWBLK. Minimum 144.

040 COMPRESS ON OPENED BANK. BANK WILL BE CLOSED

Erläuterung

Die zu komprimierende Datenbank war offen und wird geschlossen.

041 BANK: name COMPRESSED AND CLOSED

Erläuterung

Die Bank mit dem Namen name wurde komprimiert und ist geschlossen.

042 DATAPOOL FULL. IN STATEMENT 'COPY DATA' ONLY zz BYTES COPIED

Erläuterung

Im Statement 'COPY DATA' wurden nur zz bytes kopiert, da der Datapool danach voll beschrieben war.

043 OUTPUTFILE IMPLICIT OPENED

Erläuterung

Im Statement 'Copy Data' war der Outputfile geschlossen und wurde implizit geöffnet.

044 DATAPOOL: name FULLY COPIED ON OUTPUTFILE

Erläuterung

Der Eingabepool im Statement 'Copy Data' wurde vollständig auf den Ausgabefile kopiert.

045 INPUTFILE FULLY COPIED

Erläuterung

Der Eingabefile im Statement 'Copy Data' wurde vollständig kopiert.

046 INPUTFILE IMPLICIT OPENED

Erläuterung

Der Inputfile im Statement 'Copy Data' wurde implizit geöffnet.

047 INPUTFILE NOT FULLY COPIED. DATAPOOL FULL

Erläuterung

Der Ausgabepool im Statement 'Copy Data' ist voll. Der Inputfile wurde nicht vollständig kopiert.

048 POOL NO DATAPOOL

Erläuterung

Der referierte Pool ist kein Datapool.

049 NEW LENGTH LITTLER OR EQUAL
OLD LENGTH NOTHING DONE

Erläuterung
Im Befehl CHANGE LENGTH (length) wurde eine length angegeben, die kleiner oder gleich der alten Länge ist: keine Aktion.

050 NOT ENOUGH SPACE AVAILABLE TO EXTEND DATA_POOL

Erläuterung
Der Befehl CHANGE LENGTH (length) kann nicht ausgeführt werden, da nicht genügend Platz für die Bank reserviert ist.

051 NAME BLANK, UNCHANGED

Erläuterung
Im Befehl CHANGE NAME (name) wurden Leerzeichen als 'name' angegeben: keine Namensänderung.

Fehlermeldungen Subsystem EDIT

001 POOL-OPTION MISSING, STATEMENT SKIPPED

Erläuterung

Die POOL-Klausel fehlt.

002 INVALID STRING SKIPPED: string

Erläuterung

Die Zeichenkette 'string' wurde als ungültig erkannt und übersprungen.

003 STRING OR COL OPTION MISSING

Erläuterung

Die STRING- oder die COL-Klausel fehlt.

004 REPLACE OR CONTINUE OPTION MISSING

Erläuterung

REPLACE- oder CONTINUE-Klausel fehlt.

005 STRING OR SET OPTION MISSING

Erläuterung

STRING- oder SET-Klausel fehlt.

006 DECK NOT IN POOL

Erläuterung

Das in der DECK-Klausel des FIND-Statements angegebene Kartenpaket wurde nicht im EDIT-Pool gefunden.

007 STRING NOT FOUND: string

Erläuterung

Die im FIND-Statement angegebene Zeichenkette 'string' wurde in dem angegebenen Bereich (DECK,POS) nicht gefunden.

008 FOUND CARD IS INSERTED, PART2 OF SEQU_NR MISSING

Erläuterung

Die zu suchende Karte befindet sich in einem eingefügten Kartenpaket, so daß die Karten-Folgenummer zweiteilig ist: SET(CARD1:CARD2).

009 FOUND ON CARD1 card1:card2

Erläuterung

Die gesuchte Zeichenkette wurde auf der Karte card1:card2 gefunden.

010 NO CARDS IN WORKFILE

Erläuterung

Die Arbeitsdatei ist leer.

011 CARD NR. card NOT FOUND

Erläuterung

Die Karte mit der Folgenummer card befindet sich nicht in der Arbeitsdatei.

012 NO CARDS INSERTED

Erläuterung

Auf Grund der angegebenen zweiteiligen Folgenummer wurde eine eingefügte Karte gesucht, hinter der angegebenen Karte (1.Teil der Folgenummer) wurden aber keine Karten eingefügt.

013 EDIT-POOL NO DATAPOOL

Erläuterung

Der in der POOL-Klausel des ENTER EDIT-Statement referierte Pool ist kein Datapool.

014 NO CARDS IN EDIT-POOL

Erläuterung

Der in der POOL-Klausel des ENTER EDIT Statements referierte Pool ist leer.

015 FILE WITH DDNAME=QQWORKDA SHOULD BE bytes BYTES LONGER

Erläuterung

Die Hilfsdatei QQWORKDA, die die EDIT-Datei aufnehmen soll, sollte um mindestens bytes Bytes länger sein. Für einzufügende Karten ist natürlich zusätzlich Platz erforderlich.

016 CARD NR. card TRUNCATED

Erläuterung

Bei der Änderung einer Zeichenkette wurde die Spalte 72 der Karte überschritten.

017 STRING TRUNCATED: string

Erläuterung

Die einzufügende Zeichenkette 'string' ist länger als 72 Zeichen.

018 WORK-FILE GREATER THAN EDIT-FILE, NOTHING SAVED

Erläuterung

Die Arbeitsdatei enthält mehr Karten als die EDIT-Datei aufnehmen kann (z.B. durch INSERT und MERGE). Um die EDIT-Datai nicht zu zerstören, wird das SAVE nicht ausgeführt.

019 POOL FOR MERGE NOT OPEN

Erläuterung

Der in der POOL-Klausel des MERGE-Statements referierte Pool ist nicht geöffnet.

020 MERGE POOL NO DATA_POOL

Erläuterung

Der in der POOL-Klausel des MERGE-Statements referierte Pool ist kein Datapool.

13.3 Nachrichten des Subsystems PLS

Die Nachrichten des Subsystems PLS sind unterteilt in die Nachrichten, die zur Übersetzungszeit des Subsystems anfallen, und die Nachrichten, die zur Ausführungszeit des Subsystems anfallen. Die Nachrichten zur Übersetzungszeit besitzen als erstes Wort des Nachrichtentextes eine Identifikation, die aus den Buchstaben PLS und einer dreistelligen Nummer besteht. Die Nachrichten zur Ausführungszeit besitzen als erstes Wort nur eine dreistellige Nummer. Die Nachrichten sind im folgenden nach ihren Identifikationen geordnet.

Übersetzungszeit-Nachrichten

PLS034 IN DESTROY-STATEMENT NO VALID KEYWORD OR SUBSYSTEMNAME FOUND AFTER THE KEYWORD "DESTROY"

Erläuterung

In einer DESTROY-Anweisung steht nach DESTROY kein gültiges Schlüsselwort und kein gültiger Subsystemname.

PLS035 IN STATEMENT-DEFINITION-STATEMENT "DATATYPE" SPECIFIED, BUT NO VALID DATATYPE FOUND

Erläuterung

In einer STATEMENT-Anweisung wurde nach DATATYPE kein gültiger Datentyp angegeben (REAL, INT, OP, ID, BIT, STR).

PLS037 IN STATEMENT DEFINITION MORE THAN 20 ALIASES, REST OF ALIAS NAMES IGNORED

Erläuterung

In einer STATEMENT-Anweisung wurden mehr als die zulässige Anzahl Aliasnamen angegeben. Es sind maximal 20 ALIAS-Namen zulässig.

PLS038 IN STATEMENT-, CLAUSE- OR DATASTRUCTURE DEFINITION STATEMENT ERROR DURING READING OR WRITING COMFILE

Erläuterung

Während der Bearbeitung einer STATEMENT-, CLAUSE-, DATA-STRUCTURE- oder MACRO DAT-Anweisung wurde beim Beschreiben der Zwischendatei COMFILE ein Fehler festgestellt. Die gerade laufende Bearbeitung wird eingestellt.

PLS041 IN DESTROY DATASTRUCTURE STATEMENT TOO MANY NAMES IN LIST

Erläuterung

In einer DESTROY DATA STRUCTURE-Anweisung enthält die Namensliste zu viele Namen. Die Anzahl muß <=50 sein.

PLS042 IN DESTROY CLAUSE STATEMENT TOO MANY NAMES IN LIST

Erläuterung

In einer DESTROY CLAUSE-Anweisung enthält die Namensliste zu viele Namen. Die maximale Anzahl ist 50.

PLS043 IN DESTROY STATEMENT STATEMENT TOO MANY NAMES IN LIST

Erläuterung

In einer DESTROY STATEMENT-Anweisung enthält die Namensliste zu viele Namen. Die maximale Anzahl ist 50.

PLS045 IN STATEMENT-, CLAUSE- OR DATASTRUCTURE DEFINITION STATEMENT LOGICAL END OF STATEMENT FOUND BUT NO SEMICOLON ENCOUNTERED. NEXT ITEM, text, SKIPPED

Erläuterung

In einer STATEMENT-, CLAUSE- oder DATASTRUCTURE-Anweisung wurde am logischen Anweisungsende kein Semikolon gefunden. Der Rest der Anweisung wird übergangen. 'text' ist der ignorierte Teil der Anweisung.

PLS046 DATASTRUCTURE COMMON DOES NOT BEGIN WITH KEYWORD "DCL" OR "DECLARE". NEXT ITEM: text

Erläuterung

Eine Subsystemcommon-Definition beginnt nicht mit DCL oder DECLARE. 'text' ist der statt DCL angetroffene Text.

PLS049 IN STATEMENT DEFINITION "END"-STATEMENT MISPLACED

Erläuterung

In einer Statement- oder Clause-Definition wurde eine END-Anweisung an falscher Stelle angetroffen, d.h. kein zugehöriges PROC, BEGIN oder DO ist vorhanden.

PLS050 IN INITIATE STATEMENT NO VALID KEYWORD OR SUBSYSTEMNAME FOUND. NEXT ITEM: text

Erläuterung

In der INITIATE-Anweisung steht nach INITIATE kein gültiges Schlüsselwort und kein gültiger Subsystemname. 'text' ist der folgende falsche Text.

PLS051 IN STATEMENT- OR CLAUSE-DEFINITION DECLARATIONLIST IN DECLARE-STATEMENT DOES NOT START WITH A NUMBER, NAME OR "(". NEXT ITEM: text

Erläuterung

In einer Statement- oder Clause-Definition steht eine fehlerhafte DCL-Anweisung. Nach DCL bzw. dem trennenden Komma steht weder eine Levelnummer, noch ein Name, noch ein "(". 'text' ist der Anfang des fehlerhaften Anweisungsteils.

PLS052 IN STATEMENT DEFINITION EOF ON INPUT BEFORE "END STATEMENT"

Erläuterung

In einer Statement- oder Clause-Definition wurde EOF auf SYSIN angetroffen, bevor eine "END STATEMENT"-Anweisung auftrat. (EOF = End of File, Ende der Datei)

PLS053 IN STATEMENT DEFINITION TOO MANY ACTIVE VARIABLES, REST WILL BE INACTIVE

Erläuterung

Eine Statement- oder Clause-Definition enthält zu viele aktive Variable. Die Anzahl aktiver Makrozeit-Variablen in Makrozeit-Datenstruktur plus Anzahl aktiver Variabler in der Definition darf höchstens 80 sein.

PLS054 IN STATEMENT DEFINITION TOO MANY NESTED BLOCKS OR GROUPS

Erläuterung

In einer Clause- oder Statement-Definition sind Blöcke oder DO-Gruppen zu tief geschachtelt. Die Schachteltiefe darf höchstens 10 sein.

PLS055 IN CLAUSE OR STATEMENT-DEFINITION WRONG KEYWORD OR DATATYPE AFTER KEYWORD "SKIP" OR "SKIPB". NEXT ITEM: text

Erläuterung

In einer Statement- oder Clause-Definition steht nach SKIP oder SKIPB ein falsches Schlüsselwort oder ein ungültiger Datentyp. 'text' ist der ungültige Teil der Anweisung.

PLS056 MACROTIME DATASTRUCTURE DOES NOT START WITH "DCL" OR "DECLARE". NEXT ITEM: text

Erläuterung

Eine Makrozeit-Datenstruktur-Definition beginnt nicht mit DCL oder DECLARE. 'text' ist der Anfang der falschen Anweisung.

PLS057 IN FILE-STATEMENT IN GROUP OF NAMES NEITHER "D", "M", "N" NOR A NAME FOUND. NEXT ITEM: text

Erläuterung

In einer FILE-Anweisung steht in einer Namensgruppe weder "D", "M", "N" noch ein Name. 'text' ist der ungültige Anweisungsteil.

PLS061 IN STATEMENT DEFINITION "THEN" MISSING IN IF-STATEMENT. ONE HAS BEEN INSERTED

Erläuterung

In einer IF-Anweisung fehlt "THEN" oder steht an der falschen Stelle. "THEN" wird eingefügt.

PLS062 IN DESTROY STATEMENT DATATYPE-STATEMENT NO VALID DATATYPE SPECIFIED. NEXT ITEM: text

Erläuterung

In einer DESTROY STATEMENT DATATYPE wurde kein gültiger Datentyp (INT, REA, OPE, IDE, BIT, STR, ASS, %AS) gefunden. 'text' ist der ungültige Teil der Anweisung.

PLS063 TOO MANY ACTIVE VARIABLES IN MACROTIME DATASTRUCTURE

Erläuterung

Eine Makrozeit-Datenstruktur enthält mehr als 60 aktive Variable.

PLS064 ERROR IN DECLARE-STATEMENT IN MACROTIME DATASTRUCTURE, "," , ";" OR ")" MISSING OR MISPLACED

Erläuterung

In einer Makrozeit-Datenstruktur fehlt in der DECLARE-Anweisung ein "," , ";" oder ")" oder steht an der falschen Stelle.

PLS065 IN DATASTRUCTURE DEFINITION DECLARE STATEMENT CONTAINS TOO MANY NAMES

Erläuterung

Eine einzelne DECLARE-Anweisung in einer DATASTRUCTURE-Definition enthält mehr als 200 Variable. 200 ist die maximal zulässige Anzahl.

PLS066 DECLARATIONLIST IN DECLARE-STATEMENT DOES NOT START WITH NAME, NUMBER OR ")". NEXT ITEM: text

Erläuterung

In einer Datenstruktur-Definition beginnt eine Namensliste nach einem DCL oder einem trennenden Komma nicht mit einem Namen, einer Levelnummer oder einer ")". 'text' ist der ungültige Teil der Anweisung.

PLS067 IN DATASTRUCTURE DEFINITION DECLARATIONLIST IN DECLARE STATEMENT DOES NOT END WITH "," OR ";". NEXT ITEM: text

Erläuterung

Am Ende einer Deklaration in einer Datenstruktur-Definition steht nicht "," oder ";". 'text' ist der folgende Anweisungsteil.

PLS068 IN DATASTRUCTURE DEFINITION A NESTED DECLARATIONLIST IN A DECLARE-STATEMENT DOES NOT END WITH "," OR ")". NEXT ITEM: text

Erläuterung

Eine geschachtelte Deklaration in einer Datenstruktur-Definition wird nicht durch ")" oder "," abgeschlossen.
Beispiel:

```
DCL (A BIN FIXED, B PTR;
```

'text' ist der folgende Anweisungsteil, im Beispiel das Semikolon.

PLS070 IN COMMON DECLARATION AFTER "DCL 1" NO COMMA FOUND. NEXT ITEM: text

Erläuterung

In einer COMMON-Deklaration fehlt nach "DCL 1" das Komma. Alle Worte bis zum nächsten Komma oder Semikolon werden übergangen.
Beispiel:

```
DATA COMMON;
DCL 1 XYZ, ... (XYZ wird übergangen).
```

'text' ist der fehlerhafte Anweisungsteil.

Ausführungszeit-Nachrichten

101 ERROR IN SUBSYSTEM-STATEMENT, MEMBER QQSUBS ON FILE SUBLIB CANNOT BE OPENED

Erläuterung

Beim Eröffnen der Datei SUBLIB erfolgte ein Open-Fehler. SUBLIB-DD-Karte falsch oder Dataset REGENT.PLSTRAN.DATA nicht in Ordnung oder Member QQSUBS fehlt.

102 ERROR IN SUBSYSTEM-STATEMENT WHILE READING MEMBER QQSUBS ON FILE SUBLIB, QQMEM-ERROR 13

Erläuterung

Beim Lesen der Subsystemtabelle QQSUBS von der Datei mit DD-Namen SUBLIB trat ein ständiger I/O-Fehler auf. Der Lesevorgang wurde abgebrochen.
Im Statement: SUBSYSTEM

103 ERROR IN SUBSYSTEM-STATEMENT, SPECIFIED SUBSYSTEM name NOT FOUND

Erläuterung

Das Subsystem "name" ist im System nicht bekannt.
Im Statement: SUBSYSTEM

104 ERROR IN SUBSYSTEM-STATEMENT, SPECIFIED KEY key NOT VALID FOR SUBSYSTEM name

Erläuterung

Der für das Subsystem "name" angegebene Schlüssel "key" ist nicht gültig.
Im Statement: SUBSYSTEM

105 ERROR IN DESTROY MACROTIME-DATASTRUCTURE, MEMBER QQSUBS ON FILE SUBLIB CANNOT BE OPENED

Erläuterung

Siehe 101.
Im Statement: DESTROY MACROTIME DATASTRUCTURE

106 ERROR IN DESTROY MACROTIME-DATASTRUCTURE WHILE READING MEMBER QQSUBS ON FILE SUBLIB QQPMEM-ERROR 13

Erläuterung

Siehe 102.
Im Statement: DESTROY MACROTIME DATASTRUCTURE

107 ERROR IN DESTROY MACROTIME-DATASTRUCTURE, SPECIFIED SUBSYSTEM name NOT FOUND

Erläuterung

Siehe 103.
Im Statement: DESTROY MACROTIME DATASTRUCTURE

108 ERROR IN DESTROY MACROTIME-DATASTRUCTURE, MEMBER QQSUBS ON FILE SUBLIB NOT WRITTEN, QQPMEM-ERROR = fault

Erläuterung

Die Subsystemtabelle QQSUBS auf der Datei mit dem DD-Namen SUBLIB konnte nicht geschrieben werden wegen eines I/O-Fehlers.
"fault" = 8 oder 13 oder 16 Permanenter I/O-Fehler beim Öffnen, Schreiben, Lesen, Schließen der Datei
= 12 Das Directory der Datei ist voll
= 20 Die Datei ist nicht korrekt eröffnet, oder der DCB wurde überschrieben.
Im Statement: DESTROY MACROTIME DATASTRUCTURE

109 ERROR IN DESTROY MACROTIME DATASTRUCTURE WHILE WRITING FILE SUBLIB, QQPMEM-ERROR = fault

Erläuterung

Die Statement-Tabelle des Subsystems konnte nicht auf die Datei mit dem DD-Namen SUBLIB geschrieben werden.
"fault": siehe 108.
Im Statement: DESTROY MACROTIME DATASTRUCTURE

110 MACROTIME-DATASTRUCTURE OF SUBSYSTEM name HAS BEEN DESTROYED

Erläuterung

Vollzugsmeldung nach dem erfolgreichen Zerstören einer Makrozeit-Datenstruktur.

111 ERROR IN DESTROY_SUBSYSTEM STATEMENT, MEMBER QQSUBS ON FILE SUBLIB CANNOT BE OPENED

Erläuterung

Siehe 101.
Im Statement: DESTROY SUBSYSTEM

112 ERROR IN DESTROY_SUBSYSTEM STATEMENT WHILE READING MEMBER QQSUBS ON FILE SUBLIB

Erläuterung

Siehe 102.
Im Statement: DESTROY SUBSYSTEM

113 ERROR IN DESTROY_SUBSYSTEM STATEMENT, SPECIFIED SUBSYSTEM name NOT FOUND

Erläuterung

Siehe 103.
Im Statement: DESTROY SUBSYSTEM

114 ERROR IN DESTROY-SUBSYSTEM-STATEMENT, SPECIFIED KEY key NOT VALID FOR SUBSYSTEM name

Erläuterung

Siehe 104.
Im Statement: DESTROY SUBSYSTEM

115 ERROR IN DESTROY_SUBSYSTEM STATEMENT WHILE READING MEMBER member ON FILE SUBLIB, QQPMEM-ERROR 13

Erläuterung

Beim Lesen der Statement-Tabelle des Subsystems ("member") trat ein permanenter I/O-Fehler auf. Die Datei wird unter dem DD-Namen SUBLIB angesprochen. Im Statement: DESTROY SUBSYSTEM

116 ERROR IN DESTROY_SUBSYSTEM STATEMENT, MEMBER QQSUBS ON FILE SUBLIB NOT WRITTEN, QQPMEM-ERROR = fault

Erläuterung

Siehe 108.
Im Statement: DESTROY SUBSYSTEM

117 SUBSYSTEM name HAS BEEN DESTROYED

Erläuterung

Vollzugsmeldung des erfolgreichen Abschlusses der Zerstörung des Subsystems "name".

118 ERROR IN DATASTRUCTURE-DEFINITION, MEMBER QQSUBS ON FILE SUBLIB CANNOT BE OPENED

Erläuterung

Siehe 101.
Im Statement: DATASTRUCTURE

119 ERROR IN DATASTRUCTURE-DEFINITION WHILE READING MEMBER QQSUBS ON FILE SUBLIB, QQPMEM-ERROR 13

Erläuterung

Siehe 102.
Im Statement: DATASTRUCTURE

120 ERROR IN DATASTRUCTURE-DEFINITION, SPECIFIED SUBSYSTEM name NOT FOUND

Erläuterung

Siehe 103.
Im Statement: DATASTRUCTURE

121 ERROR IN DATASTRUCTURE-DEFINITION WHILE READING MEMBER member ON FILE SUBLIB, QQPMEM-ERROR 13

Erläuterung

Siehe 115
Im Statement: DATASTRUCTURE

122 ERROR IN DATASTRUCTURE-DEFINITION WHILE WRITING MEMBER member ON FILE SUBLIB, QQPMEM-ERROR NO. fault

Erläuterung

Beim Schreiben der Statement-Tabelle des Subsystems ("member") von der Datei mit dem DD-Namen SUBLIB trat ein permanenter I/O-Fehler auf. "fault" siehe 108.
Im Statement: DATASTRUCTURE

123 ERROR IN DATASTRUCTURE-DEFINITION, MEMBER QQSUBS ON FILE SUBLIB NOT WRITTEN, QQPMEM-ERROR = fault

Erläuterung

Siehe 108.
Im Statement: DATASTRUCTURE

124 ERROR IN DATASTRUCTURE-DEFINITION, FILE COMFILE CANNOT BE OPENED, QQPMEM-ERROR = fault

Erläuterung

Die Datei mit dem Namen COMFILE kann nicht eröffnet werden. DDNAME fehlt oder falsch, oder I/O-Fehler. "fault" siehe 108.
Im Statement: DATASTRUCTURE

125 ERROR IN DATASTRUCTURE-DEFINITION, FILE SUBLIB CANNOT BE OPENED, QQPMEM-ERROR = fault

Erläuterung

Beim Öffnen der Datei mit dem DD-Namen SUBLIB trat ein Fehler auf. "fault" siehe 108.
Im Statement: DATASTRUCTURE

126 ERROR IN DATASTRUCTURE DEFINITION WHILE READING FILE COMFILE, QQPMEM-ERROR = fault

Erläuterung

Beim Lesen der Datei mit dem Namen COMFILE trat ein Fehler auf. "fault" siehe 108.
Im Statement: DATASTRUCTURE

127 ERROR IN DATASTRUCTURE DEFINITION, WHILE WRITING COMMON ON FILE SUBLIB, QQPMEM-ERROR NO. fault

Erläuterung

Beim Schreiben der COMMON-Struktur auf die Datei mit dem DD-Namen SUBLIB trat ein Fehler auf. "fault" siehe 108.
Im Statement: DATASTRUCTURE

128 ERROR IN DATASTRUCTURE DEFINITION WHILE WRITING COMMON ON FILE SUBLIB, QQPMEM-ERROR = fault

Erläuterung

Siehe 127.

129 ERROR IN DATASTRUCTURE DEFINITION WHILE CLOSING FILE COMFILE, QQPMEM-ERROR = fault

Erläuterung

Beim Schließen der Datei mit dem DD-Namen COMFILE trat ein Fehler auf. "fault" siehe 108.
Im Statement: DATASTRUCTURE

130 ERROR IN DATASTRUCTURE DEFINITION WHILE WRITING FILE SUBLIB, QQPMEM-ERROR = fault

Erläuterung

Beim Beschreiben der Datei mit dem DD-Namen SUBLIB trat ein Fehler auf. "fault" siehe 108.
Im Statement: DATASTRUCTURE

131 DATASTRUCTURE dname HAS BEEN ADDED TO FILE SUBLIB WITH MEMBERNAME name

Erläuterung

Vollzugsmeldung nach der erfolgreichen Definition einer Datenstruktur. "name" = Membername auf der Datei SUBLIB. "dname" = Name der Datenstruktur.

132 ERROR IN STATEMENT-DEFINITION-PROCESSING, FILE COMFILE CANNOT BE OPENED FOR OBJECT-CODE

Erläuterung

Beim Öffnen der Datei mit dem DD-Namen COMFILE trat ein Fehler auf. DD-Karte vergessen oder falsch, oder I/O-Error.

133 ERROR IN STATEMENT DEFINITION PROCESSING, OBJECT CODE CANNOT BE COPIED TO COMFILE, FAULT = fault

Erläuterung

Siehe 132, "fault" siehe 108.
Im Statement: STATEMENT

134 ERROR IN DESTROY-STATEMENT, MEMBER QQSUBS ON FILE SUBLIB CANNOT BE OPENED

Erläuterung

Siehe 101.
Im Statement: DESTROY

135 ERROR IN DESTROY-STATEMENT WHILE READING MEMBER QQSUBS ON FILE SUBLIB, QQPMEM-ERROR 13

Erläuterung

Siehe 102.
Im Statement: DESTROY

136 ERROR IN DESTROY-STATEMENT, SPECIFIED SUBSYSTEM name NOT FOUND

Erläuterung

Siehe 103.
Im Statement: DESTROY

137 ERROR IN DESTROY-STATEMENT WHILE READING MEMBER member ON FILE SUBLIB, QQPMEM-ERROR 13

Erläuterung

Siehe 115.
Im Statement: DESTROY

138 ERROR IN DESTROY-STATEMENT, MEMBER QQSUBS ON FILE SUBLIB NOT WRITTEN, QQPMEM-ERROR = fault

Erläuterung

Siehe 108.
Im Statement: DESTROY

139 ERROR IN DESTROY-STATEMENT WHILE WRITING MEMBER member ON FILE SUBLIB, QQPMEM-ERROR NO. fault

Erläuterung

Siehe 122.
Im Statement: DESTROY

140 ERROR IN MACROTIME-DATA-DEFINITION, MEMBER QQSUBS ON FILE SUBLIB CANNOT BE OPENED

Erläuterung

Siehe 101.
Im Statement: MACROTIME DATASTRUCTURE

141 ERROR IN MACROTIME-DATA-DEFINITION WHILE READING MEMBER QQSUBS ON FILE SUBLIB, QQPMEM-ERROR 13

Erläuterung

Siehe 102.
Im Statement: MACROTIME DATASTRUCTURE

142 ERROR IN MACROTIME-DATA-DEFINITION, SPECIFIED SUBSYSTEM name NOT FOUND

Erläuterung

Siehe 103.
Im Statement: MACROTIME DATASTRUCTURE

143 ERROR IN MACROTIME-DATA-DEFINITION, MEMBER QQSUBS ON FILE SUBLIB NOT WRITTEN, QQPMEM-ERROR = fault

Erläuterung

Siehe 108.
Im Statement: MACROTIME DATASTRUCTURE

144 ERROR IN MACROTIME-DATA-DEFINITION, FILE COMFILE CANNOT BE OPENED, QQPMEM-ERROR = fault

Erläuterung

Siehe 124.
Im Statement: MACROTIME DATASTRUCTURE

145 ERROR IN MACROTIME-DATA-DEFINITION, FILE SUBLIB CANNOT BE OPENED, QQPMEM-ERROR = fault

Erläuterung

Siehe 125.
Im Statement: MACROTIME DATASTRUCTURE

146 ERROR IN MACROTIME-DATA-DEFINITION WHILE READING FILE COMFILE QQPMEM-ERROR = fault

Erläuterung

Siehe 126.
Im Statement: MACROTIME DATASTRUCTURE

147 ERROR IN MACROTIME-DATA-DEFINITION WHILE WRITING FILE SUBLIB, QQPMEM-ERROR = fault

Erläuterung

Siehe 130.
Im Statement: MACROTIME DATASTRUCTURE

148 MACROTIME-DATASTRUCTURE HAS BEEN ADDED TO FILE SUBLIB WITH MEMBERNAME member

Erläuterung

Vollzugsmeldung des erfolgreichen Abschlusses der Definition einer Übersetzungszeit-Datenstruktur."member" = Membername der Datenstruktur in der Datei mit dem DD-Namen SUBLIB.

149 ERROR IN SUBSYSTEM INITIALISATION, MEMBER QQSUBS ON FILE SUBLIB CANNOT BE OPENED

Erläuterung

Siehe 101.
Im Statement: INITIATE

150 ERROR IN SUBSYSTEM INITIALISATION WHILE READING MEMBER QQSUBS ON FILE SUBLIB, ERROR 13 IN QQPMEM

Erläuterung

Siehe 102.
Im Statement: INITIATE

151 SUBSYSTEM name HAS BEEN DESTROYED AND WILL BE NEW INITIALISED

Erläuterung

Das Subsystem "name", das initialisiert werden soll, ist schon vorhanden. Vor der Neuinitialisierung wird es zerstört.

152 ERROR IN SUBSYSTEM INITIALISATION, SUBSYSTEM name ALREADY EXISTS, BUT WRONG KEY SPECIFIED

Erläuterung

Das Subsystem "name" existiert bereits und ist durch einen anderen als den jetzt angegebenen Schlüssel geschützt. Die Initialisierung wird nicht vorgenommen.

153 ERROR IN SUBSYSTEM INITIALISATION,SUBSYSTEM NAME name INVALID, HAS IDENTICAL BEGINNING AS EXISTING ABBREVIABLE SUBSYSTEM NAME

Erläuterung

Die Initialisierung des Subsystems "name" wurde nicht vorgenommen, da der Anfang des Subsystemnamens mit einem schon vorhandenen Subsystem übereinstimmt.
Beispiel: SUBSYSTEM 'DAT.EN' existiert bereits, wenn jetzt INITIATE SUBSYSTEM 'DATAMANAGEMENT' initialisiert werden soll, erscheint Fehlermeldung 153.

154 WARNING IN SUBSYSTEM CREATION, NEW ABBREVIABLE SUBSYSTEM NAME name CAUSES EXISTING SUBSYSTEM TO BE INACCESSIBLE

Erläuterung

Durch die Initialisierung des neuen Subsystems "name" können bestehende Subsysteme nicht mehr verwendet werden.
Beispiel: Es gibt Subsystem STRUKTUR, wenn jetzt mit INITIATE SUBSYSTEM 'STRU.DL' ein neues Subsystem erzeugt wird, werden alle Aufrufe ENTER STRUKTUR als Aufrufe von STRUDL mißverstanden.

155 ERROR IN SUBSYSTEM INITIALISATION, MEMBER QQSUBS ON FILE SUBLIB CANNOT BE OPENED

Erläuterung

Siehe 101.
Im Statement: INITIATE

156 ERROR IN SUBSYSTEM INITIALISATION, MEMBER QQSUBS ON FILE SUBLIB NOT WRITTEN, QQPMEM-ERROR = fault

Erläuterung

Siehe 108.
Im Statement: INITIATE

157 ERROR IN SUBSYSTEM INITIALISATION, MEMBER PLREG ON FILE SUBLIB CANNOT BE OPENED, QQPMEM-ERROR = fault

Erläuterung

Siehe 125.
Im Statement: INITIATE

158 ERROR IN SUBSYSTEM INITIALISATION, MEMBER PLREG ON FILE SUBLIB CANNOT BE READ, QQPMEM-ERROR = fault

Erläuterung

Beim Lesen der Datei mit dem DD-Namen SUBLIB trat ein Fehler auf. "fault" siehe 108.
Im Statement: INITIATE

159 ERROR IN SUBSYSTEM INITIALISATION, MEMBER member CANNOT BE ADDED TO FILE SUBLIB, QQPMEM-ERROR = fault

Erläuterung

Siehe 122.
Im Statement: INITIATE

160 ERROR IN SUBSYSTEM INITIALISATION, COMMON DATASTRUCTURE CANNOT BE ADDED TO FILE SUBLIB, QQPMEM-ERROR = fault

Erläuterung

Die Standard-COMMON-Datenstruktur konnte nicht auf die Datei mit dem DD-Namen SUBLIB geschrieben werden. "fault" siehe 108.
Im Statement: INITIATE

161 ERROR IN SUBSYSTEM INITIALISATION, MACROTIME-DATASTRUCTURE CANNOT BE ADDED TO FILE SUBLIB, QQPMEM-ERROR = fault

Erläuterung

Die Standard-Übersetzungszeit-Datenstruktur konnte nicht auf die Datei mit dem DD-Namen SUBLIB geschrieben werden. "fault" siehe 108.
Im Statement: INITIATE

162 SUBSYSTEM name HAS BEEN INITIATED

Erläuterung

Vollzugsmeldung des erfolgreichen Abschlusses der Initialisierung des Subsystems "name".

163 ERROR IN READING SUBSYSTEM-TABLE FROM FILE SUBLIB, QQPMEM-FAULT = fault, QQPMEM-ICODE = code

Erläuterung

Beim Lesen der Subsystemtabelle von der Datei mit dem DD-Namen SUBLIB trat ein Fehler auf. "fault" siehe 108.
"code" = 2: Der Fehler trat beim OPEN auf
= 4: Der Fehler trat beim Lesen auf
= 6: Der Fehler trat beim CLOSE auf
Im Statement: FILE

164 ERROR IN FILE-PROCESSING, STATEMENT name NOT FOUND

Erläuterung

In einem FILE-Statement wurde die Statement-Definition "name" angesprochen. Dieses Statement wurde aber nicht definiert.

165 ERROR IN FILE-PROCESSING, MEMBER QQSUBS ON FILE SUBLIB CANNOT BE OPENED

Erläuterung

Siehe 101.
Im Statement: FILE

166 ERROR IN FILE PROCESSING WHILE READING MEMBER QQSUBS ON FILE SUBLIB, QQPMEM-ERROR = fault

Erläuterung

Siehe 102.
Im Statement: FILE

167 ERROR IN FILE-PROCESSING, SPECIFIED SUBSYSTEM name NOT FOUND

Erläuterung

Siehe 103.
Im Statement: FILE

168 ERROR IN FILE PROCESSING WHILE READING MEMBER member ON FILE SUBLIB, QQPMEM-ERROR = fault

Erläuterung

Beim Lesen des Members "member" von der Datei mit dem DD-Namen SUBLIB trat ein Fehler auf. "fault" siehe 108.

169 ERROR IN FILE PROCESSING STATEMENT NAME name INVALID, HAS IDENTICAL BEGINNINGS AS EXISTING ABBREVIABLE STATEMENT n1

Erläuterung

Der Statement-Name "name" ist ungültig, da er den gleichen Anfang hat wie ein bereits vorhandener abkürzbarer Statement-Name "n1".
Beispiel: Vorhanden: 'PLO.T', ungültig: PLOTFORMAT.

170 WARNING IN FILE PROCESSING, NEW ABBREVIABLE STATEMENT name CAUSES EXISTING STATEMENT(S) TO BE INACCESSIBLE

Erläuterung

Durch das neue Statement "name" können vorhandene Statements nicht mehr benutzt werden.
Beispiel: Das neue Statement 'PU.NCH' macht das vorhandene Statement 'PUT' unbenutzbar.

171 ERROR IN FILE-PROCESSING, MEMBER QQSUBS ON FILE SUBLIB NOT WRITTEN, QQPMEM-ERROR = fault

Erläuterung

Siehe 108.
Im Statement: FILE

172 ERROR IN FILE PROCESSING WHILE WRITING MEMBER member ON FILE SUBLIB, QQPMEM-ERROR NO. fault

Erläuterung

Beim Schreiben des Members "member" auf die Datei mit dem DD-Namen SUBLIB trat ein Fehler auf. "fault" siehe 108.
Im Statement: FILE

173 STATEMENT MODULE name READY FOR LINK EDIT

Erläuterung

Vollzugsmeldung. Der Anweisungstreiber-Modul mit dem Namen "name" ist fertig zur Verarbeitung durch den Linkage Editor.

174 ERROR IN STATEMENT-DEFINITION WHILE READING MEMBER QQSUBS ON FILE SUBLIB, QQPMEM-ERROR 13

Erläuterung

Siehe 102.
Im Statement: STATEMENT

175 ERROR IN STATEMENT-DEFINITION, SPECIFIED SUBSYSTEM name NOT FOUND

Erläuterung

Siehe 103.
Im Statement: STATEMENT

176 ERROR IN STATEMENT DEFINITION PROCESSING, MACROTIME DATASTRUCTURE CANNOT BE INCLUDED FROM FILE SUBLIB

Erläuterung

Die Übersetzungszeit-Datenstruktur kann bei einer Statement-Definition nicht von der Datei mit dem DD-Namen SUBLIB gelesen werden.

196 ERROR IN STATEMENT DEFINITION PROCESSING, ACTIVE VARIABLES CANNOT BE READ FROM FILE COMFILE

Erläuterung

Die Liste der aktiven Variablen kann bei einer Statement-Definition nicht von der Datei mit dem Namen COMFILE gelesen werden. DD-Karte für COMFILE fehlt oder falsch, oder I/O-Fehler.

197 ERROR IN STATEMENT DEFINITION PROCESSING, MORE THAN 90 ACTIVE VARIABLES

Erläuterung

Eine Statement-Definition enthält zu viele aktive Variable. Entweder die Definition in Clauses unterteilen oder die Anzahl der Variablen reduzieren oder einen Teil der Variablen UNACTIVE deklarieren.

180 ERROR DURING PLPASS2 WHILE READING COMFILE, QQPMEM-ERROR = fault IN STATEMENT NO. stnr

Erläuterung

In der Anweisung Nr. "stnr" innerhalb einer Statement-Definition trat ein I/O-Fehler auf. "fault" siehe 108. Die Datei hat den DD-Namen COMFILE.

181 ERROR IN STATEMENT DEFINITION PROCESSING, IN PLS-FUNCTION name ")" MISSING IN STATEMENT NO. stnr

Erläuterung

In der Anweisung Nr. "stnr" innerhalb einer Statement-Definition wurde die PLS-Funktion "name" falsch angewandt. Eine schließende Klammer fehlt.

182 ERROR IN STATEMENT DEFINITION PROCESSING, IN PLS-FUNCTION name ARGUMENTS MISSPELLED IN STATEMENT NO. stnr

Erläuterung

In der Anweisung Nr. "stnr" innerhalb einer Statement-Definition wurde die PLS-Funktion "name" falsch angewandt. Die Argumentliste der Funktion ist fehlerhaft.

190 ERROR IN DESTROY-STATEMENT, NAME name NOT FOUND

Erläuterung

In einer DESTROY STATEMENT-Anweisung wurde ein unbekannter Statementname "name" angegeben.

191 STATEMENT name HAS BEEN DESTROYED
192 DATATYPE-STATEMENT type HAS BEEN DESTROYED

Erläuterung

Vollzugsmeldung des erfolgreichen Abschlusses einer DESTROY-STATEMENT-Anweisung mit dem Namen "name" oder des Datentyps "type".

193 SUBSYSTEM name NOT FOUND

Erläuterung

Siehe 103.
Im Statement: LIST STATEMENTS

194 STATEMENT name HAS BEEN PROCESSED
194 CLAUSE name HAS BEEN PROCESSED

Erläuterung

Vollzugsmeldung des erfolgreichen Abschlusses einer Statement- oder Clause-Definition mit dem Namen "name".

195 ERROR IN STATEMENT-DEFINITION MEMBER QQSUBS ON FILE SUBLIB CANNOT BE OPENED

Erläuterung

Siehe 101.
Im Statement: STATEMENT

L i t e r a t u r

[1] Schlechtendahl, E.G., Enderle, G.: The Design of the Integrated CAD-System REGENT, Proc. Workshop on General Purpose CAD-Systems, Toulouse 1976

[2] Enderle, G., Schlechtendahl, E.G.: The CAD-System REGENT, 12th Automation Conference, Boston, Mass.1975

[3] Schlechtendahl, E.G., Leinemann,K.: The REGENT-System for CAD, Proc. IFIP WG 5.2 Working Conf., Austin, Texas 1976

[4] Schlechtendahl, E.G.: Comparison of Integrated Systems for CAD. In: Institution of Electrical Engineers (Hrsg.): Int. Conf. on Computer Aided Design, Southampton, April 8-11, 1974. IEE conf. publ. No. 111, (1974) pp. 111-116

[5] Schlechtendahl, E.G.: Das integrierte CAD-System REGENT, KFK-CAD 2, 1975

[6] Schlechtendahl, E.G.: Grundzüge des integrierten CAD-Systems REGENT, Angewandte Informatik 18 (1976) 11, S.490-496

[7] Enderle, G.: Problemorientierte Sprachen im REGENT-System, Angewandte Informatik 18 (1976) 12, S.543-549

[8] Leinemann, K.: Dynamische Datenstrukturen des integrierten CAD-Systems REGENT, Angewandte Informatik 19 (1977) 1, S.26-32

[9] Schuster, R.: GIPSY-Graphische Fähigkeiten als Bestandteil eines Systems für den rechnergestützten Entwurf, Angewandte Informatik 19 (1977) 4, S.155-167

/10/ Leinemann, K.: Der Einfluß von Betriebssystemen mit virtuellem Adreßraum (Paging) auf das Konzept dynamischer Datenfelder, KFK-Ext. 8/75-1

/11/ Schlechtendahl, E.G., Enderle, G., Leinemann, K., Schuster, R.: Das Programmsystem REGENT im praktischen schen Einsatz.
GI-Fachtagung: Methoden der Informatik für Rechnergestütztes Entwerfen und Konstruieren, München, 19.-21.Oktober 1977

/12/ Enderle, G.: Definition, Übersetzung und Anwendung benutzerorientierter Sprachen im CAD-System REGENT, KFK 2204, 1975

/13/ Enderle, G. und Steil, A.: Der REGENT-PLR-Precompiler, Programmlogik, KFK-Ext.8/76-3

/14/ Schuster, R.: System und Sprache zur Behandlung graphischer Information im rechnergestützten Entwurf (GIPSY), KFK 2305, 1976

/15/ Enderle, G.: FLUST-2D- Ein Programm zur Berechnung der zweidimensionalen Strömung eines kompressiblen Mediums in aneinandergekoppelten Rechteckbereichen. KfK 2679, 1978 (in Vorbereitung)

ANHANG

Ergänzende Dokumentation

Folgende Primärberichte enthalten unveröffentlichte Informationen von vorläufigem und betriebsinternem Charakter. Eine Zurverfügungstellung der Berichte ist nach entsprechender einzelvertraglicher Vereinbarung über die Nutzung des darin enthaltenen know how (know-how-Vertrag) möglich. Entsprechende Anfragen sind an die Abteilung Patente und Lizenzen des KfK zu richten.

Interaktive Version von REGENT

Interaktive Benutzung von REGENT, IRE/6/1605.1/270/76

Programmlogik, Übersicht

Leinemann, K.: Die REGENT-Datenbank, IRE/6/1605.1/OH1A/244/1975

Bechler, K.H.: Dokumentation der Datenbankroutinen und Datenbankutilities, IRE/6/1605.1/OH1A/243/1975

Leinemann, K.: Dynamische Datenfelder des REGENT-Systemkerns, IRE/6/1605.1/OH1A/246/1975

Pee, A.: Vergleich IMS 2 mit Datenbankverwaltung, REGENT/10/1973

Leinemann, K.: Erfahrungen mit der REGENT-DDS-Verwaltung (Verwaltung des virtuellen Datenpools) IRE/6/1605.1/364/77

Olbrich, W.: Beschreibung einiger Basisfunktionen des REGENT-Systemkerns, IRE/6/POH1A/320/1976

Einzel-Source-Dokumentation

REGENT-Quelldokumentation
Für alle REGENT-Teilbereiche einheitliche handschriftliche Dokumentation auf Formblätter für

- alle Prozeduren
- alle Datenstrukturen
- alle Makros

Bericht Nr. IRE/6/OH1A/345/1977

Subsysteme

Wossmann,J.: Das REGENT-Subsystem FRAME, IRE/6/1605.1/266/1976

Doll,F.: Erzeugung der Problemsprache FLOWCHART(FLOW) mit Hilfe von PLS, IRE/6/204/1975

Doll, R.: Erstellung eines Zeichenausgabeteils für das REGENT-Subsystem FLOW zur Darstellung von Fluß-diagrammen, IRE/6/1605.1/238/1975

Stölting, K.: Benutzungsanleitung für das REGENT-Subsystem YAQUIR, IRE/6/4223/272/1976

Ladisch, R., Marek,U.: LITERATUR-Handbuch, GfK,IRE, 1976

Katz, F.: NERZ-REGENT-Subsystem zur Netz-Erzeugung, Benutzerhandbuch,IRE/6/1605.1/276/1976

Bechler, K.H., Schumann, U.: REMAC - ein REGENT-Subsystem zur Berechnung inkompressibler Strömungen mit freien Oberflächen und MHD-Kräften, IRE/6/310/76

Schlechtendahl, E.G. ;Enderle, G.:
Spezifikation von Änderungen und Erweiterungen
an GIPSY. Primärbericht 09.02.01P01E, 1978

Bechler, K.H.: FLUSTPLOT, ein Programm zur Auswertung der Ergebnisse des Fluiddynamikcodes FLUST. Primärbericht 06.01.02P01D, 1978

Beschreibung des REGENT-Subsystems HDR-EVALUATION zur Auswertung der Blowdown-Berechnungen. Notiz IRE/6/355/1977