 Karlsruhe, den 21.4.1977

Stand der PEARL - Entwicklung

L. Degelow
H.-J. Gottwald
H. Schoknecht

SIEMENS AG Karlsruhe, E STE 333

Inhaltsübersicht

1. Historie zur PEARL-Entwicklung

2. Charakterisierung der Echtzeitsprache PEARL

3. Aufbau eines PEARL-Moduls

3.1 Der Systemteil
Beispiel eines Systemteils

3.2 Der Problemteil
Sprachmittel zur Echtzeitverarbeitung

4. Der PEARL-Compiler PC30
Aufbau und Struktur des Compilers
Einige relevante Daten des Compilers

5. Erstellungsweg eines PEARL-Programms
Aufbau eines PEARL-Systems

6. Integration des PEARL-Compilers in das Spektrum der Dienstprogramme SIEMENS 300-16 Bit

7. Erwartungen durch Verwendung der Sprache PEARL

1. Historie der PEARL-Entwicklung

Seit etwa Mitte der sechziger Jahre gibt es Versuche, höhere Programmiersprachen vornehmlich für spezielle Anwendungen in der Echtzeitprogrammierung zu schaffen. Solche in der Praxis eingesetzten Sprache sind zumeist FORTRAN-Derivate. Vielfach behilft man sich auch mit vorgefertigten Programmen, Softwarepaketen, die zum Teil durch sprachähnliche Mittel individuell erweitert wurden (MADAM, ARSS etc.).

Ende der sechziger Jahre tauchten dann vor allem in Europa Sprachentwürfe mit universellem Charakter auf, z.B.:

RTL/2	in England
PROCOL	in Frankreich
PAS1 und PEARL	in der BRD

Im Herbst 1969 trat erstmals der PEARL-Arbeitskreis, eine Arbeitsgruppe aus Anwender, Herstellerfirmen, Softwarehäusern und Instituten, zusammen.

Ziel der Arbeiten war die Definition einer höheren Programmiersprache für leichtere und funktionsgerechtere Programmierung von Echtzeitproblemen. Eine solche Sprache soll auch den Austausch von Prozeß- und Experiment-Programmen erleichtern.

Ein erster Rohentwurf zur Sprache PEARL wurde 1973 vorgelegt. Um eine einheitliche Sprachdefinition bemühten sich von Anfang an die Implementatoren. Diese Einigung wurde im Januar dieses Jahres erzielt und durch die Definition von Basis-PEARL festgeschrieben.

2. Charakterisierung der Echtzeitsprache PEARL

PEARL wird meines Erachtens zu recht als universelle höhere Programmiersprache bezeichnet.

PEARL ist universell im doppelten Sinn:

Sie ist zum einen für ein breites Anwendungsspektrum, zum anderen auch praktisch für alle kommerziell eingesetzten Prozeßrechner geeignet.

Der Trend zu höheren, d.h. maschinenunabhängigen Programmiersprachen ist allgemein akzeptiert. Sie versprechen insbesondere verminderte Programmherstellungskosten.

Die Gründe dafür, daß man sich in der Prozeßdatenverarbeitung so lange mit maschinenabhängigen Sprachen behelfen mußte, sind in der stets vorhandenen Verschiedenartigkeit der Prozeßperipherie und in einigen Besonderheiten der Prozeßrechnersysteme zu suchen, die programmtechnisch schwierig zu beherrschen sind.

Diese Besonderheiten sind - kurz gesagt - folgende:

- Die Programme müssen über spezielle Geräte (Prozeßperipherie) mit dem technischen Prozeß verkehren, die für jedes System individuelle Adressen, Namen und Strukturen haben.

- Viele Programme müssen sich selbst schritthaltend mit der Real-Zeit und ggfs. den Ereignissen in anderen Programmen im Ablauf steuern können (Synchronisationsproblem).

PEARL enthält im Sprachkern Formulierungsmittel für diese Aufgaben.

Im folgenden möchte ich auf die wichtigsten Sprachmittel und die Struktur von PEARL-Modulen eingehen.

3. Aufbau eines PEARL-Moduls

- siehe Bild 1 -

In einem PEARL-Programm sind die Beschreibung der System-Konfiguration (Anschluß der Peripherie = Systemteil) und die Problemformulierung (Problemteil) getrennt.

Diese Trennung hat den Vorteil, daß der Problem-Teil unabhängig von der Anlagen-Konfiguration formuliert werden kann. Durch Hinzufügen (Binden) eines entsprechenden System-Teiles wird das Programm auf der Zielmaschine ablauffähig.

Ein PEARL-Modul wird von den Schlüsselworten MODULE und MODEND eingefaßt. Der Systemteil wird durch das Schlüsselwort SYSTEM eingeleitet. Das Schlüsselwort PROBLEM ist das Ende des Systemteils bzw. leitet den Anfang des Problemteils ein.

Das Schlüsselwort MODEND kennzeichnet das Ende des Systemteils, wenn der Problemteil fehlt.
Es ist möglich den Systemteil oder einen Problemteil getrennt zu übersetzen.

Wird aus den einzelnen Tasks zu globalen Daten zugegriffen, so wird zusätzlich ein Common-Data-Modul mit dem Namen PEARLC generiert.

Ein Problemteil kann aus mehreren Tasks und/oder globalen Proceduren bestehen.

Aufbau eines PEARL-Moduls

```
/#PEARL
     MODULE;
     SYSTEM;                    /* Systemteil */
          .
          .
          .
        Anweisungen;
          .
          .
          .
     PROBLEM;                   /* Problemteil */
     DCL ..... ,
         ..... ;                /* globale Daten */

     T1: TASK;                  /* 1. Task       */
     DCL ..... ;
          .
          .
          .
        Anweisungen;
          .
          .
          .
     END;
      .
      .
      .
     Tn: TASK;                  /* n-te Task     */
      DCL ..... ;
          .
          .
          .
        Anweisungen
          .
          .
          .
     END;
     MODEND;
/=
/*
```

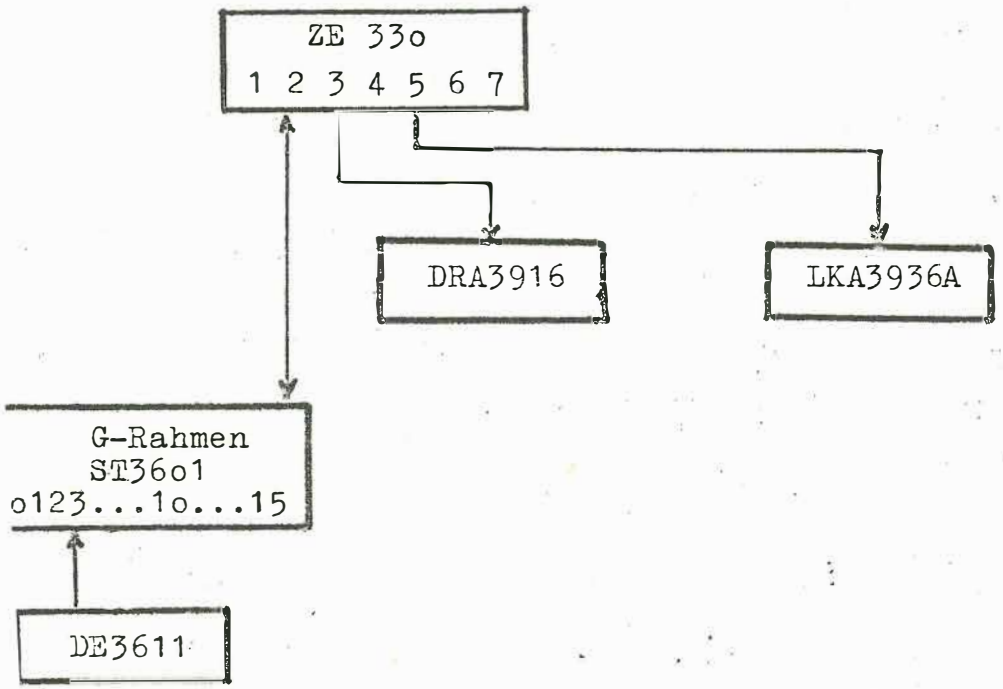
3.1 Der SYSTEM-Teil

Im SYSTEM-Teil wird die gesamte Hardware-Konfiguration der für die PEARL-Programme benötigten Peripherie-Geräte beschrieben.

Den einzelnen Anschlußstellen der E/A-Geräte, Prozeßendstellen, Interrupts und Signals bzw. den Knotenpunkten kann ein Anwendername zugeordnet werden. Dieser Anwendername muß im PROBLEM-Teil spezifiziert werden, damit er in diesem verwendet werden kann. Der SYSTEM-Teil beschreibt die Richtung des gewünschten Datenflusses zwischen den einzelnen Anschlußstellen.

Beispiel für einen SYSTEM-Teil

```
SYSTEM;
ST3601 <--> ZE330 * 2;
DE1:       DE3611 --> ST3601 * 3;
DRUA:      DRA3916 <-- ZE330 * 3;
LKAU:      LKA3936A <-- ZE330 * 5;
```



n die Zentraleinheit ZE330 soll ein Schnelldrucker, eine
ochkartenausgabe direkt und eine Digitaleingabe über einen
-Rahmen (Grundsteuerung) angeschlossen werden.

3.2 Der PROBLEM-Teil

Alle innerhalb des PROBLEM-Teils verwendeten Größen müssen in diesem deklariert bzw. spezifiziert werden. Der gesamte PROBLEM-Teil besteht demgemäß aus dem Modul-Spezifikationssatz, dem Modul-Deklarationssatz, den globalen Proceduren und den **Tasks.**

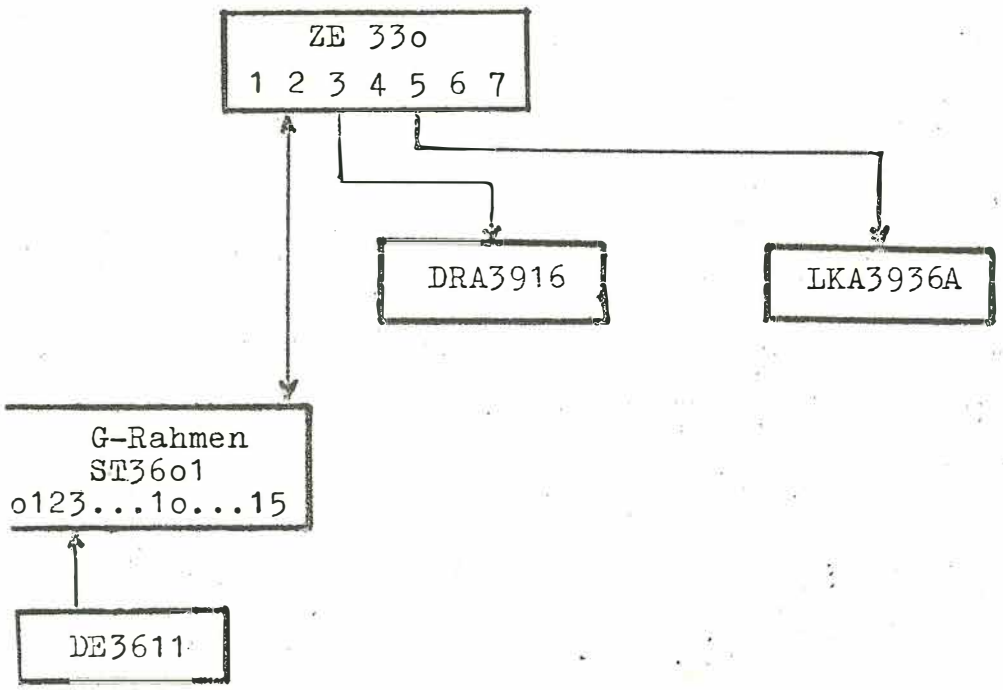
Dabei sind die Tasks Programme im Sinne des Betriebssystems.

Die gesamten arithmetischen Anweisungen, sowie die Anweisungen zur Organisation der Real-Zeit-Verarbeitung können nur innerhalb der Tasks oder Proceduren formuliert werden. Der Problemteil als Ganzes ist nicht ausführbar. Datenaustausch zwischen TASK's erfolgt über globale, mindestens auf Modulebene deklarierte, Daten und Daten-Bereiche. Hierfür stehen dem Anwender alle üblichen arithmetischen Funktionen einer höheren Programmiersprache (FORTRAN, ALGOL, PL/1) zur Verfügung.

Zusätzlich kann er den Datentyp STRUCT verwenden. Dieser Datentyp faßt einzelne Standard-Datentypen zu einem Verbund zusammen. Der Zugriff zu einem einzelnen Element der Struktur ist über die Selektion möglich.

Beispiel für einen SYSTEM-Teil

```
SYSTEM;
ST3601 <---> ZE330 * 2;
DE1:         DE3611 ---> ST3601 * 3;
DRUA:        DRA3916 <--- ZE330 * 3;
LKAU:        LKA3936A <-- ZE330 * 5;
```



n die Zentraleinheit ZE330 soll ein Schnelldrucker, eine
ochkartenausgabe direkt und eine Digitaleingabe über einen
-Rahmen (Grundsteuerung) angeschlossen werden.

3.2 Der PROBLEM-Teil

Alle innerhalb des PROBLEM-Teils verwendeten Größen müssen in diesem deklariert bzw. spezifiziert werden. Der gesamte PROBLEM-Teil besteht demgemäß aus dem Modul-Spezifikationssatz, dem Modul-Deklarationssatz, den globalen Proceduren und den **Tasks.**

Dabei sind die Tasks Programme im Sinne des Betriebssystems.

Die gesamten arithmetischen Anweisungen, sowie die Anweisungen zur Organisation der Real-Zeit-Verarbeitung können nur innerhalb der Tasks oder Proceduren formuliert werden. Der Problemteil als Ganzes ist nicht ausführbar. Datenaustausch zwischen TASK's erfolgt über globale, mindestens auf Modulebene deklarierte, Daten und Daten-Bereiche. Hierfür stehen dem Anwender alle üblichen arithmetischen Funktionen einer höheren Programmiersprache (FORTRAN, ALGOL, PL/1) zur Verfügung.

Zusätzlich kann er den Datentyp STRUCT verwenden. Dieser Datentyp faßt einzelne Standard-Datentypen zu einem Verbund zusammen. Der Zugriff zu einem einzelnen Element der Struktur ist über die Selektion möglich.

Beispiel

Deklaration

```
DECLARE  CM  STRUCT (/ L FIXED, M FLOAT /);
```

Anwendung (Selektion)

```
CM.L : =  15 ;
```

Aufbau eines PROBLEM-Teils:

```
PROBLEM;
SPC K   SIGNAL      /x MODUL-SPEZIFIKATIONSSATZ*/
SPC L   ENTRY (INV FIXED);
     .
     .
     .
DCL X   (3,5) FIXED, /*MODUL-DEKLARATIONSSATZ*/
DCL Y   STRUCT (/S1 BIT, S2 FLOAT/);
     .
     .
     .
   T1 : TASK;
     .
     .
     .
        X (1,4) := 12;
     .
     .
     .
        END;
   P1 : PROC (Z FIXED) RETURNS (FLOAT);
     .
     .
     .
        RETURN (Y,S2);
     .
     .
     .
        END;
     .
     .
MODEND;
```

PEARL-Sprachmittel zur Real-Zeit-Verarbeitung

Da Prozeßsteuerungs-Systeme in der Regel aus einer Anzahl von in sich autonomen Programmen bestehen, die Einzelprozesse steuern, ist eine Koordinierung auf höherer Ebene erforderlich. Um diese asynchron verlaufenden Prozesse zu koordinieren, besitzt PEARL eine Reihe von Sprachelementen.

Arbeiten mit TASK's

In PEARL werden alle Tasks mit ihren bei der Deklaration angegebenen Namen angesprochen.

Dem Anwender stehen Aufrufe zum Starten (ACTIVATE), Beenden (TERMINATE), Anhalten (SUSPEND), Fortsetzen (CONTINUE) einer Task zur Verfügung. Weiterhin kann eine Task angehalten werden; sie wird auf ein bestimmtes Ereignis hin fortgesetzt. Die Aufrufe können mit einer entsprechenden Zeit-Modifikation (Schedule) versehen werden.

Beispiele:

```
AT 5:Ø:Ø EVERY 2HRS ACTIVATE T;
```

Die Task T soll um 5 Uhr ab, alle 2 Stunden zyklisch gestartet werden.

```
WHEN IT1 RESUME;
```

Die laufende Task wird angehalten und bei Eintreffen des Interrupts IT1 fortgesetzt.

Alle mit einem Schedule versehenen Aufrufe einer Task (z.B. mit Namen T1) können vom Anwender zurückgenommen werden.

Beispiel:

```
PREVENT T1;
```

Alarm-, Interrupt- und Signal-Anwendung

Mit Hilfe des oben erwähnten WHEN-Statements kann der Anwender auf das Eintreffen eines Alarms warten.

Weiterhin stehen ihm eine Anzahl Standard-Signals zur Verfügung, die compiler- und systemabhängig sind. Der Anwender hat die Möglichkeit, auf ein Signal durch das ON-Statement zu reagieren. Wünscht er dies nicht, wird eine Standard-Fehler-Routine durchlaufen.

Beispiel:

```
ON SIG1 :   :   /*SIGNAL-REAKTION*/
```

Die Reaktion auf ein Signal kann der Anwender auch durch die folgende Anweisung einleiten:

```
INDUCE SIG1;
```

Bei Eintreffen eines Signals wird die ON-Reaktion durchlaufen und das Programm gegebenenfalls an der Aufrufstelle fortgesetzt.

Semaphore

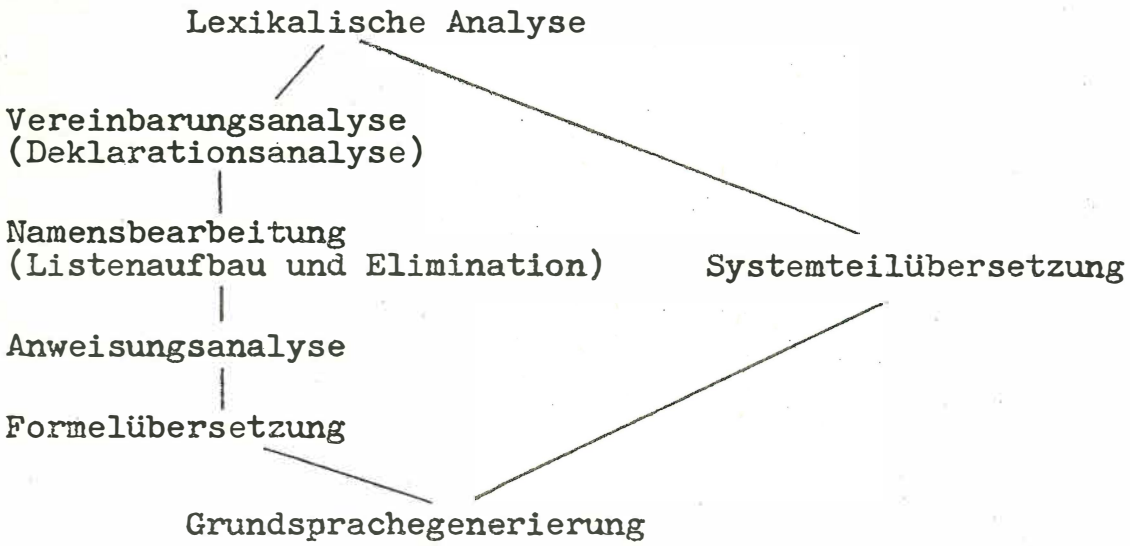
Semaphore stehen dem Anwender zur Koordinierung von Tasks zur Verfügung. Besonders zur Koordinierung der Ein-/Ausgabe-Vorgänge eignet sich die Verwendung von Semaphoren. Zur Koordinierung dienen die REQUEST-Anweisung (Semaphor-Variable erniedrigen) und die RELEASE-Anweisung (Semaphor-Variable erhöhen).

Hat die Semaphor-Variable den Wert Ø und wird eine REQUEST-Anweisung ausgeführt, so wird die aufrufende Task angehalten und fortgesetzt, wenn von einer anderen Task eine RELEASE-Anweisung auf diese Semaphor-Variable durchlaufen wird.

Ist der Wert der Semaphor-Variablen bei Ausführung der REQUEST-Anweisung größer Ø, so wird die Semaphor-Variable um 1 erniedrigt und die aufrufende Task fortgesetzt.

4. Der PEARL-Compiler PC30

Ein in der Sprache PEARL geschriebenes Programm wird vom Compiler in mehreren Durchläufen (= Pässen) auf eine Folge von Befehlen auf Grundsprache-Ebene zurückgeführt. Die einzelnen Pässe haben folgende Aufgaben:



Alle Teile des Compilers werden mit dem Binder BD30 zu einem ablauffähigen System gebunden. Die Arbeitsfassung des PC30 wird mit dem Segmentiersystem SEGSYS erstellt.

Einige Daten des Compilers:

minimaler Laufbereich : 16K Worte

siehe Bild 3

5. Erstellungsweg eines PEARL-Programms

Der PEARL-Compiler liest die Quellsprache wahlweise von Lochkarte bzw. PSD (Bedienparameter) und übersetzt i.a. einen PEARL-Modul in mehrere Grundsprachemodule der Systeme 300-16 Bit.

Die Quellsprache sowie die Grundsprachemodule entsprechen der Norm des Bibliothekssystems BISY.

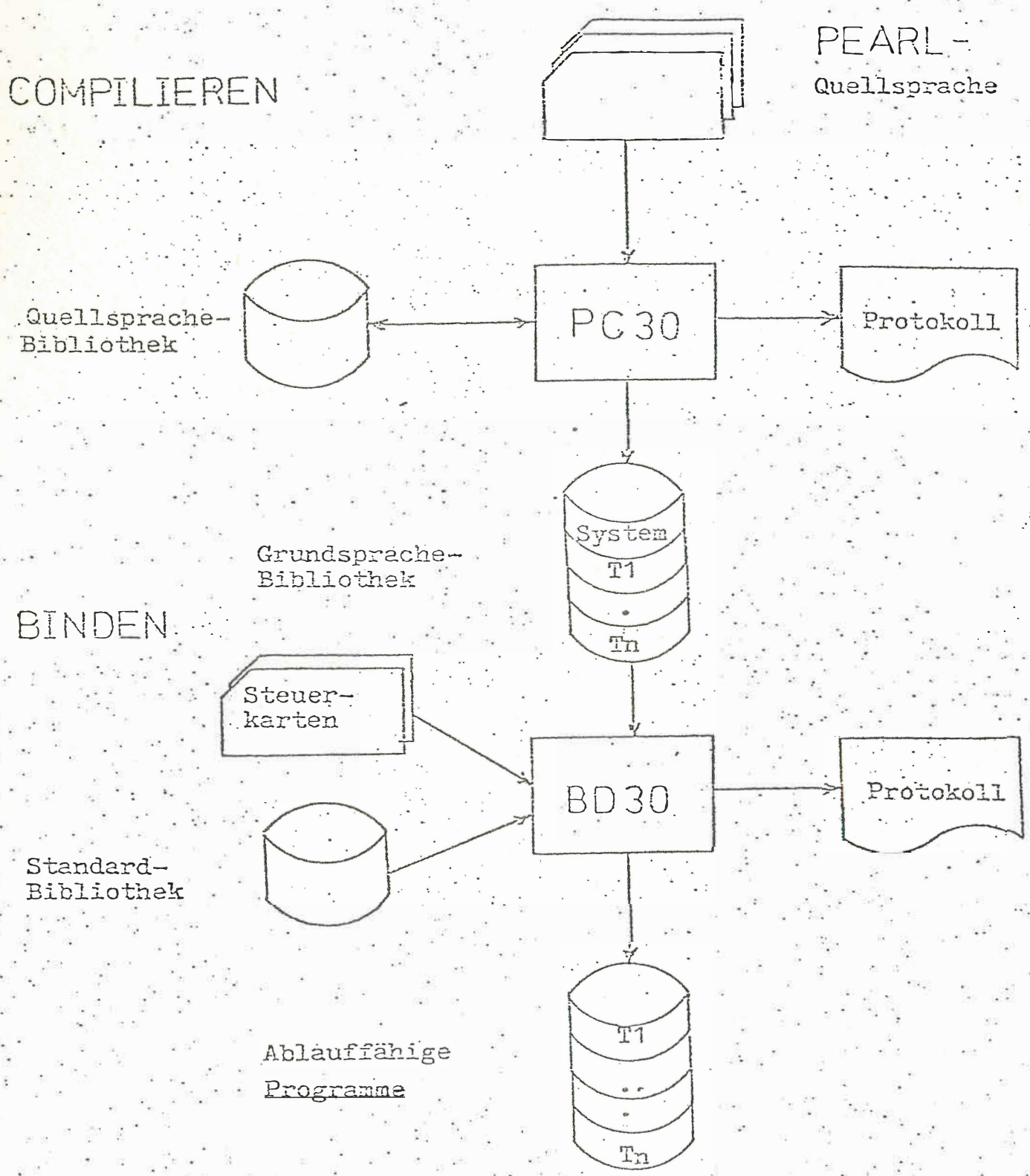
Durch Binden mittels des Binders BD30 der einzelnen Tasks und den entsprechenden Funktionen aus der Standardlibrary entstehen ladbare Grundsprachemodule.

Durch Laden dieser Module mittels des Ladebinders werden die Beziehungen zu den globalen Größen (Variablen) hergestellt und es entsteht ein ablauffähiges PEARL-System.

Aufgrund der Tatsache, daß einzelne Teile eines PEARL-Systems getrennt übersetzt werden können, sind die Steuerkarten für den BD30 vom Anwender zu schreiben. Der PEARL-Compiler teilt beim Übersetzungslauf jedoch die Parameter am Bediengerät mit.

ABLAUFBLOCKBILD Zur

Erstellung eines ablauffähigen PEARL-Programms

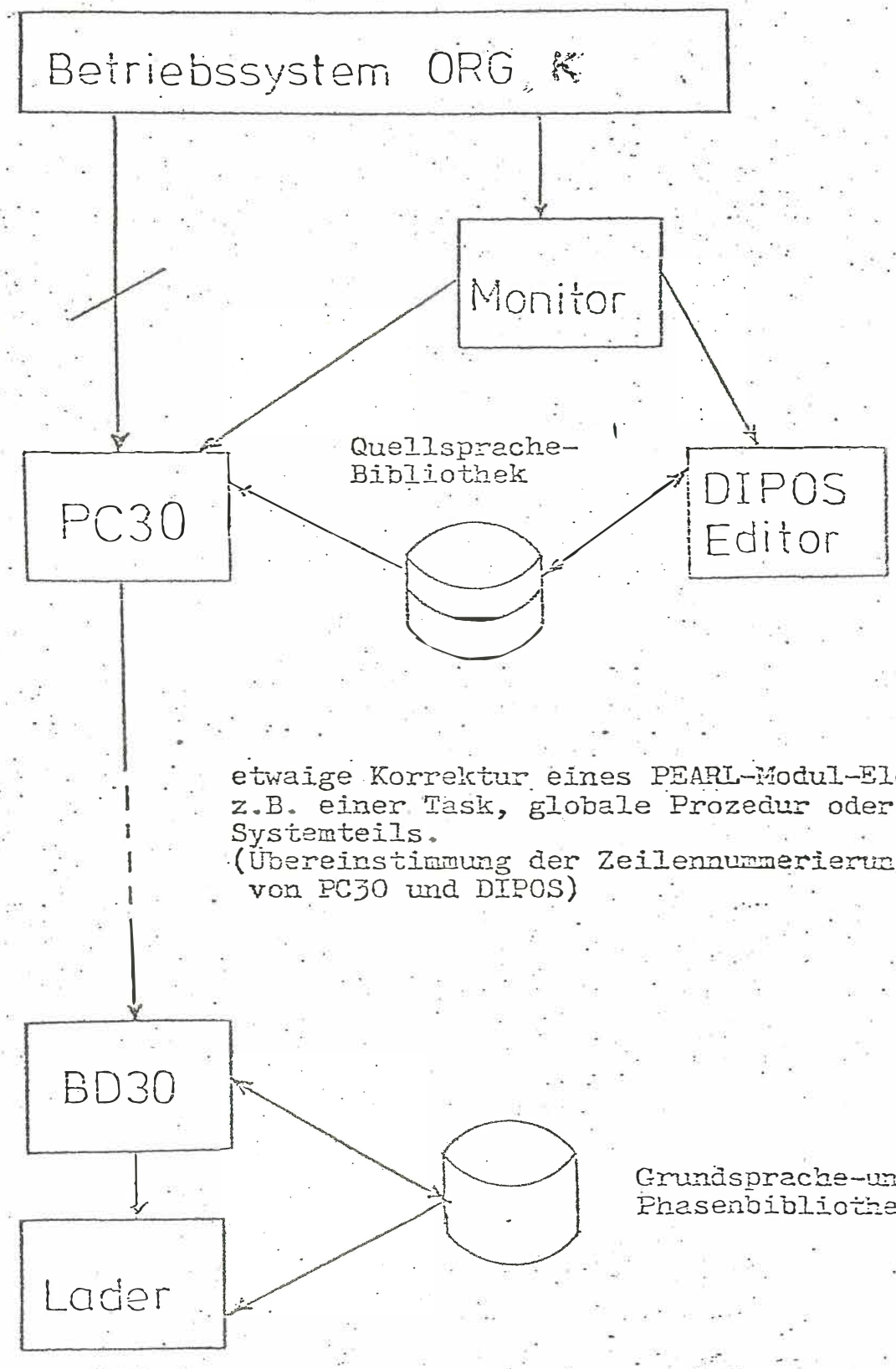


6. Integration des PEARL-Compilers PC30 in das Spektrum der Dienstprogramme SIEMENS 300-16 Bit

Der PEARL-Compiler ist unter dem Betriebssystem ORG 330 K und/oder dem Monitor MONI 330 ablauffähig. Der In- sowie Output lassen sich mit der Standardsoftware bearbeiten. Insbesondere lassen sich die PEARL-Quelldaten mittels Editor (Funktion des DIPOS) korrigieren (Übereinstimmung der Zeilennummerierung).

Integration des PEARL-Compilers PC30

in das Spektrum der Dienstprogramme SIEMENS 300-16 Bit



7. Erwartungen durch Anwendung von PEARL

Aufgrund der immer höher steigenden Erstellungskosten der Software bei gleichzeitigem Absinken der Hardwarekosten, werden mit Verwendung von PEARL die Programmerstellungskosten sinken.

PEARL kann bei

- der Aufgabenanalyse? wenig!
- Systementwurf? einiges!
- Programmtest? einiges!
- Dokumentation? viel!
- Inbetriebnahme? wenig!
- Programm-Wartung? einiges!

zur Verminderung der Kosten und Vereinheitlichung beitragen.

Was PEARL nicht leistet ist die Festlegung komplizierter Datenstrukturen auf Grund der Problemstellung selbst, zusätzlich die Gesamtproblematik der Generierungs- und Bedienungsebene.