Programmierung mit PEARL90

Richtlinien
Version 1.1/28.10.96

von L. Frevert

1 Einleitung: Sinn von Programmierrichtlinien

Es ist relativ leicht, eine Programmiersprache zu entwickeln, bei deren Benutzung keine Fehler gemacht werden können: sie darf nur den einzigen audführbaren Befehl STOP enthalten. Das heißt, je mehr Möglichkeiten eine Programmiersprache dem Benutzer bietet, umso mehr Fehler kann er machen. Jede Einschränkung, die getroffen wird, um eine Sprache sicherer zu machen, bedeutet gleichzeitig eine Einschränkung in ihrer Anwendungsbreite. Es ist daher sinnvoll, gewisse Einschränkungen nicht in der Sprache selbst vorzunehmen, sondern durch Richtlinien zu treffen, die auf Erfahrungen mit fehlerhaften oder fehleranfälligen Programmen beruhen. Gewiß kann es dann bei speziellen Problemen notwendig sein, sich über die eine oder andere Richtlinie hinwegzusetzen; dann sollte man das aber mit der gebotenen Vorsicht tun und gut zu begründen wissen, warum man gegen eine Richtlinie verstoßen mußte.

Die folgenden Richtlinien beruhen auf Erfahrungen bei der Erstellung relativ umfangreicher Programme, sowie aus der Beobachtung typischer Fehler in Praktikumsprogrammen und in Lösungen von Klausuraufgaben. Ihre Befolgung hilft, Programmierfehler zu vermeiden und die Wartbarkeit von Programmen zu erhöhen.

Bei jeder Richtlinie steht außerdem eine Begründung, warum sie befolgt werden sollte.

2 Regeln zur äußeren Form

2.1 Modulanfang

Jeder PEARL-Modul muß mit einem Kommentar beginnen, der den Zweck des Programmes bzw. des Moduls, die Versionsnummer des Moduls, das Datum der letzten Ænderung des Moduls und die Namen der Autoren enthält.

Ohne Informationen über den Zweck eines Moduls an seinem Anfang ist ein Modul erfahrungsgemäß nichts anderes als unnützer Datenschrott, weil auch der Autor nach absehbarer Zeit vergessen hat, wozu er ihn geschrieben hat.

2.2 Kommentare

Jedes ohne Benutzung eines Entwurfserkzeugs geschriebene PEARL-Programm sollte *schon bei der Erstniederschrift* mit Kommentaren versehen werden, die die Funktionsweise von Programmabschnitten erläutern. Insbesondere müssen Tasks und Prozeduren an ihrem Anfang eine Beschreibung ihres Zweckes enthalten. Schwer verständliche Einzelanweisungen sind ebenfalls zu erläutern. Richtwert: Das Verhältnis von Kommentarzeilen zu PEARL-Code sollte mindestens 1:5 sein.

Praktisch jedes Programm enthält noch Fehler, die sich erst bei längerer Benutzung bemerkbar machen. Ohne Kommentare erfordert ihre Beseitigung einen wesentlich höheren Aufand. Auf Kommentare im PEARL-Programm kann nur verzichtet werden, wenn es durch ein Programmierwerkzeug automatisch aus einem Entwurf erzeugt werden kann.

2.2.1 Task- und Prozedurbeschreibungen

Die Kommentare, die den Zweck einer Task oder Prozedur beschreiben, müssen sich direkt hinter dem Task- bzw. Prozeduranfang (nicht vor der Prozedur) befinden.

Andernfalls wird beim Kopieren einer vorhandenen Prozedur in ein neu zu entwickelndes Programm zu leicht vergessen, die Beschreibung mit zu kopieren.

2.3 Groß- und Kleinschreibung

Bezüglich der Groß- und Kleinschreibung in PEARL-Programmen sollte ein einheitliches Konzept verfolgt werden, z. B.:

1. Normale Bezeichner werden mit Kleinbuchstaben geschrieben,
2. alle Bezeichner von Modulnamen, Konstanten und selbstdefinierten Datentypen werden mit Großbuchstaben geschrieben,
3. Bezeichner von globalen Objekten werden mit großem Anfangsbuchstaben geschrieben,
4. bei Bezeichnern von Zeigern werden der 2. und 3. fBuchstabe groß geschrieben.

Die Maßnahme erhöht die Selbstdokumentation

2.4 Verwendung von Leerzeilen und Einrückungen

Die Moduln sind durch Leerzeilen und Einrückungen in eine derartige Form zu bringen, daß ihre Struktur leicht erkannt werden kann (Siehe die Beispiele). (Dazu kann das PEARL-Tool layout.out verwendet werden.)

Mit entsprechenden Leerzeilen und Einrückungen versehene PEARL-Programme lassen die Programmstruktur im Sinne der Strukturierten Programmierung erkennen und erleichtern die spätere Wartung.

3 Programmstruktur

3.1 Aufteilung in Moduln

Größere Programme sollten in mehrere Moduln aufgeteilt werden. Ein Modul sollte nicht länger als 1500 Zeilen sein. Die Aufteilung sollte so erfolgen, daß möglichst wenige globale Objekte vereinbart werden müssen. Die einzelnen Moduln sollten nicht wechselseitig voneinander abhängen, das heißt, kein Modul, der die Exportschnittstelle eines anderen Moduls benutzt, sollte diesem anderen Modul ebenfalls eine Exportschnittstelle bieten,

Einer geschickte Aufteilung eines Programmes in einzelne Moduln erleichtert die Lokalisierung und Beseitigung von Programmierfehlern und spätere Verbesserungen des Programmes. Wechselseitige Abhängigkeiten von Moduln erschweren die Lokalisierung von Programmierfehlern bei der Integration zum Gesamtprogramm, selbst wenn die Moduln mit simulierten Schnittstellen einzeln getestet worden sind.

Die Benennung der Datenstationen sollte in einem speziellen Systemteil-Modul erfolgen. In ihm sind auch diejenigen Datenstationen zu vereinbaren, die von mehreren Moduln benutzt werden.

Auf diese Weise kann die Protierung eines Programmes auf ein anderes Betriebssystem erleichtert werden.

3.2 Import- und Exportschnittstellen

3.2.1 Globale Daten und Tasks

Mit dem Attribut GLOBAL sollten keine Variablen, Semaphore, Bolts und Tasks versehen werden. Der Zugriff auf derartige Objekte aus anderen Moduln heraus sollte mit Hilfe von Prozeduren erfolgen (Beispiel 1).

Benutzung von Variablen, Semaphore, Bolts und Tasks aus mehreren Tasks heraus erfordert sorgfältige Koordination. Derartige Koordinationsaufgaben sollten deshalb nicht über Modulgrenzen hinausreichen. Bei rein sequentiellen Programmen (mit nur einer Task) erleichtert die Kapselung von Variablen im Sinne der objektorientierten Programmierung die spätere Wartung (Beispiel 1).

```
                                                  statt
DCL druck FLOAT;!aendern m. z.B. Druck.setzen(5.);   | DCL druck FLOAT GLOBAL;
DCL zugriff BOLT;                                    |
setzen:PROC (neuerdruck FLOAT);                      |
   RESERVE zugriff;                                  |
   druck:=neuerdruck;                                |
   FREE zugriff;                                     |
END;                                                 |
DCL Druck STRUCT[setzen REF PROC(FLOAT)] GLOBAL      |
                    INIT(setzen);                    |
```

Beispiel 1: Globales Datenobjekt (Datenkapselung)

3.2.2 Exportschnittstellen (global verfügbare Prozeduren)

Prozeduren, die auch von anderen Moduln aus aufgerufen werden müssen, sollten ebenfalls nicht mit dem Attribut GLOBAL vereinbart werden. Stattdessen sollte ihr Aufruf mit Hilfe einer objektorientierten Schnittstelle (einer als GLOBAL vereinbarte Struktur, die Pointer auf die betreffenden Prozeduren enthält) erfolgen (Beispiel 1).

Die Benutzung von objektorientierten Schnittstellen ermöglicht es, die Exportschnittstelle eines Moduls auf eine entsprechende Deklaration zu konzentrieren und verbessert dadurch die Selbstdokumentation des Moduls.

3.2.3 Ort der Schnittstellen-Vereinbarungen

Die Import-Schnittstellen eines Moduls (Spezifikationen) sollten am Modulanfang stehen. Die objektorientierte Exportschnittstelle sollte am Modulende deklariert werden.

Die wichtigen Schnittstellen sind dadurch leichter auffindbar

4 Deklarationen

4.1 Bezeichnungen

Alle Namen von Variablen, Tasks, Prozeduren usw. sind so zu wählen, daß ersichtlich ist, welchem Zweck sie dienen, und daß Verwechslungen durch Schreibfehlern vorgebeugt wird.

Dadurch erhöht sich die Selbstdokumentation und Wartbarkeit der Programme.

4.2 Konstanten

4.2.1 Benutzung von Konstantenwerten

Außer den Zahlenwerten 0, 1 und 2 und den Bitwerten '1'B und '0'B sollten keine Zahlen-, Zeichenketten- oder Bitketten-Werte an anderer Stelle verwendet werden als in der Deklaration benannter Konstanten (Beispiel 2).

Programme werden dadurch wartungfreundlicher.

```
                                        statt
DCL FELDLAENGE INV FIXED INIT(100);   |    DCL feld(100) FIXED;
 ......                               |
                                      |
bearbeite:PROC(feld() FIXED IDENT);   |    bearbeite:PROC;
 .....                                |      ......
END;                                  |    END;
                                      |
start:TASK;                           |    start:TASK;
  DCL feld(FELDLAENGE) FIXED;         |      CALL bearbeite;
  CALL bearbeite feld;                |    END;
END;                                  |
```

Beispiel 2: Verwendung von Konstanten, Ort von Deklarationen

4.2.2 Ort der Deklaration von Konstanten

Benannte Konstanten sollten gemeinsam am Anfang der Problemteile deklariert werden.

Dadurch wird die Wartung erleichtert.

4.3 Variablen-Deklarationen

4.3.1 Modulglobale Variable

Variable sollten nicht mit dem Attribut GLOBAL vereinbart werden (siehe oben). Außerdem sollte ihre Deklaration möglichst nicht modulglobal erfolgen, sondern lokal innerhalb der Prozeduren und Tasks, die sie benutzen (Beispile 2).

Auf jede Variable, die außerhalb von Prozeduren und Tasks vereinbart worden ist, könnten im Prinzip mehrere Tasks gleichzeitig zugreifen. Dann müßten diese Zugriffe mit Semaphoren oder Bolts so koordiniert werden, daß z.B. die Variable nicht gleichzeitig von zwei Tasks verändert wird. Deshalb sollte man die Anzahl der modulglobalen Variablen auf diejenigen beschränken, die zum Informationsaustausch zwischen Tasks unbedingt notwendig ist. Der Informationsaustausch zwischen Tasks und Prozeduren, sowie von Prozeduren untereinander kann bekanntlich mit Hilfe von Paramteren erfolgen.

4.3.2 Ort der Vereinbarung

Modulglobale Variable sollten gemeinsam am Anfang des Problemteils deklariert werden.

Dadurch wird die Selbstdokumentation und Wartung erleichtert

4.3.3 Initialisierung von Variablen

Anfangswerte von normalen (nicht Referenz-)Variablen sollten durch explizite Zuweisung gegeben werden, nicht bei der Deklaration durch INIT(...) (Beispiel 3). Referenzvariablen hingegen sollte bei der Deklaration stets ein Anfangswert gegeben werden.

Da bei modulglobalen Variablen das Setzen von Anfangswerten durch INIT nur beim Laden des Programmes erfolgt, sind solche Programme nicht mehrfach startbar. Bei prozedur- und tasklokalen Variablen verbessert es die Selbstdokumentation, wenn die Anfangswerte dort zugewiesen werden, wo sie gebraucht werden.

Bei Referenz-Variablen, die keine Anfangswerte haben, besteht hingegen die Gefahr, daß man mit ihnen auf verbotene Speicherbereiche zuzugreifen versucht.

```
                          statt
start:TASK;                 |      start:TASK;
  DCL fertig BIT(1);        |        DCL fertig BIT(1) INIT('1'B);
  ......                    |        ......
  fertig:='0'B;             |        ......
  WHILE NOT fertig REPEAT   |        WHILE NOT fertig REPEAT
    ....                    |          ....
  END;                      |        END;
END;                        |      END;
```

Beispiel 3: Initialisierung von Variablen, WHILE-Schleifen

4.4 Prozeduren

4.4.1 INV-Attribut bei Parametern

Formale Parameter, die innerhalb der Prozeduren nicht verändert werden, müssen mit dem Attribut INV versehen werden.

Man erkennt dadurch insbesondere bei REF-Variablen, ob sie oder Variable, auf die sie zeigen, durch die Prozedur verändert werden. Außerdem werden dadurch manche Programmierfehler schon bei der Kompilation entdeckt.

4.4.2 RETURN-Anweisung

Die RETURN-Anweisung von Funktionsprozeduren muß direkt an deren Ende stehen. In normalen Prozeduren sollten keine RETURN-Anweisungen verwendet werden.

Man vermeidet so Verklemmungen dadurch, daß Prozeduren in kritischen Abschnitten (zwischen REQUEST- und RELEASE-Anweisungen) verlassen werden

4.5 Tasks

4.5.1 Prozeduren versus Tasks

Bei jeder ACTIVATE-Anweisung für eine Task ist zu prüfen, ob sie nicht durch den Aufruf einer entsprechenden parameterlosen Prozedur ersetzt werden kann.

Sequentielle Abläufe von Programmteilen sind leichter zu durchschauen als ihr paralleler Ablauf.

4.5.2 Anzahl von Tasks

Die Anzahl der in einem Programm verwendeten Tasks ist möglichst klein zu halten. Jedoch gilt: Die Zahl der Tasks sollte mindestens so groß sein wie die Zahl der Interrupts.

Da nur eine Einplanung pro Task gültig sein kann, benötigt man je Interrupt eine Task, um auf alle Interrupt reagieren zu können.

4.5.3 Tasks mit sehr ähnlichen Aufgaben

Falls mehrere Tasks sehr ähnliche Aufgaben zu erledigen haben, sollte jede von ihnen nur den Aufruf einer entsprechenden Prozedur enthalten

Es ist leichter, eine einzige Prozedur zu testen und zu verbessern, als viele Tasks.

5 Referenzen

5.1 Initialisierung

Referenz-Variable (Zeiger) sollen sofort bei ihrer Vereinbarung mit INIT(..) mit einem Anfangswert (evtl. NIL) versehen werden.

Man vermeidet damit, daß eine Referenz-Variable unkontrolliert irgendwohin zeigt.

5.2 Dereferenzierung

Wenn irgend möglich, sollte explizit mit CONT dereferenziert werden.

Die Selbstdokumentation von Programmen wird dadurch verbessert. Falls a und b normale Variable gleichen Typs sind, ist "WHILE a ISNT b REPEAT" genau so unsinnig wie "WHILE 5 /= 6 REPEAT". "WHILE CONT azeiger ISNT b REPEAT hingegen kann sinnvoll sein, und man sieht durch CONT sofort, daß hier der Wert eines Zeigers (ein Name) mit einem anderen Namen verglichen wird.

5.3 Verwendung von INV bei Referenz-Variablen

Bei der Deklaration von Referenz-Variablen und bei der Verwendung von Referenz-Variablen als formale Parameter von Prozeduren muß mit Hilfe des Attributes INV genau bezeichnet werden, ob die Referenz-Variable, die Variable, auf die sie zeigt, oder sogar beide gegen überschreiben geschützt sein sollen.

Dadurch werden unangenehme Programmierfehler schon bei der Kompilation entdeckt.

6 Sequentielle Abläufe

6.1 Sprunganweisung

Die Sprunganweisung GOTO darf nicht verwendet werden, außer in Reaktionen auf SIGNALs in ON-Anweisungen.

Verwendung von GOTO ergibt schlecht durchschaubare Programme. Zur Steuerung des sequentiellen Programmablauf enthält PEARL90 sehr gute andere Sprachmittel, z. B. die EXIT-Anweisung zum Verlassen von Schleifen.

6.2 IF-Verzweigungen

IF-Verzweigungen sollten im PEARL-Code möglichst nicht ineinander geschachtelt werden, sondern in aufeinander folgende IF-Verzweigungen aufgelöst werden (Beispiel 3).

Programme werden dadurch bezüglich ihrer Logik leichter durchschaubar.

```
                    statt
    IF a<0 THEN       |          IF a<0 THEN
       ...            |            ...
    FIN;              |           ELSE
    IF a==0 THEN      |            IF a==0 THEN
       ...            |              ...
    FIN;              |             ELSE
    IF a>0 THEN       |               ...
      ...             |            FIN;
    FIN;              |          FIN;
```

Beispiel 3: Initialisierung von Variablen, WHILE-Schleifen

6.3 CASE-Verzweigungen

6.3.1 Zugelassene Arten

CASE-Verzweigungen mit implizit numerierten Alternativen sollten nicht verwendet werden; stattdessen sollten die CASE-Verzweigungen mit explizit angegebenen Fällen benutzt werden (Beispiel 4).

CASE-Verzweigungen mit implizit numerierten Alternativen laufen zwar etwas schneller, führen aber auch leichter zu Programmierfehlern.

```
                              statt
    CASE menuewahl              |          CASE menuewahl
      ALT (1)                   |            ALT
         fixedbearbeitung;      |               fixedbearbeitung;
      ALT (2)                   |            ALT
         floatbearbeitung;      |               floatbearbeitung;
      OUT                       |          FIN;
         fehlermeldung;         |
    FIN;                        |
```

Beispiel 4: CASE-Anweisung

6.3.2 OUT-Zweige

Eine CASE-Verzwigung sollte immer einen OUT-Zweig aufweisen (Beispiel 4).

Man vermeidet dadurch, daß gar nichts gemacht wird, wenn keine Alternative durchlaufen wurde.

6.4 REPEAT-Blöcke

6.4.1 WHILE-Schleifen

Bei Schleifen vom Typ WHILE Bedingung REPEAT ist dafür zu sorgen, daß die Bedingung direkt vor dem Schleifenanfang gesetzt wird. (Beispiel 3).

Andernfalls wird das Setzen der Bedingung leicht vergessen, sodaß der Eintritt in die Schleife bei Ausführung des Programmes vom Zufall abhängt.

6.4.2 Endlos-Schleifen

Endlosschleifen sind nur zulässig, wenn in der Schleife nicht ausschließlich Anweisungen stehen, die zu ununterbrochenen Rechnungen führen (Beispiel 5).

Derartige Schleifen beschäftigen den Prozessor einer Einprozessormaschine ununterbrochen und führen dazu, daß parallelen Tasks keine oder zu wenig Rechenzeit zur Verfügung steht.

```
                                    statt
REPEAT                                |    REPEAT
  IF autozahl == 0 THEN               |      IF autozahl == 0 THEN
    EXIT;                             |        GOTO ende;
  FIN;                                |      FIN;
  IF tageszeit > wachenschluss THEN   |      IF tageszeit > wachenschluss THEN
    EXIT;                             |        GOTO ende;
  FIN;                                |      FIN;
  AFTER wartezeit RESUME;             |    END;ende:;
END;                                  |
IF autozahl == 0 THEN                 |
  ......                              |
FIN;                                  |
IF tageszeit > wachenschluss THEN     |
  .......                             |
FIN;                                  |
```

Beispiel 5: Endlosschleife mit mehreren Abbruchbedingungen

6.4.3 Schleifen mit mehreren Abbruchbedingungen

Bei Schleifen, deren Abbruch durch mehrere Bedingungen bewirkt werden kann, muß sofort anschließend geprüft werden, welche der Bedingungen zum Abbruch geführt hat, und für jede Bedingung eine entsprechende Reaktion vorgesehen werden (Beispiel 5).

Andernfalls entstehen Programme, die unter bestimmten Umständen fehlerhaft sind.

7 Parallelarbeit

7.1 Tasks

7.1.1 Aktivierung und Einplanung

Für jede Task sollte nur eine einzige ACTIVATE-Anweisung im Programm existieren.

Bei normalem ACTIVATE ist man dadurch relativ sicher, daß eine Task nicht gleichzeitig durch zwei Tasks aktiviert wird. Bei Einplanungen gilt sowieso nur die letzte; deshalb ist es z.B. grob falsch zu schreiben "WHEN erster_Interrupt ACTIVATE diesetask;" und dann "WHEN zweiter_Interrupt ACTIVATE diesetask;", um auf beide Interrupts zu reagieren.

7.1.2 Terminierung

Eine Task sollte nicht durch eine andere mit TERMINATE beendet werden. Falls das doch unbedingt notwendig ist, muß man durch geeignete Maßnahmen dafür sorgen, daß die zu terminierende Task nicht in einem kritischen Abschnitt oder innerhalb einer Prozedur terminiert wird, die sich in einem anderen Modul befindet (Beispiel 6).

Dadurch vermeidet man Verklemmungen.

7.1.3 Ausplanung

Falls eine eingeplante Task mit PREVENT ausgeplant werden soll, welche eine RESUME-Anweisung enthält, muß dafür gesorgt werden, daß die Ausplanung nicht während des RESUME-Wartezustandes erfolgen kann.

Die PREVENT-Anweisung bewirkt außer der Streichung einer Einplanung auch die unbedingte Suspendierung einer Task, die mit RESUME wartet.

```
                                    statt
DCL terminate_moeglich SEMA PRESET(1); |
arbeitstask:TASK;                      |     arbeitstask:TASK;
   REQUEST terminate_moeglich;         |        REPEAT
     REPEAT                            |          CALL im_anderen_modul;
       CALL im_anderen_modul;          |          CALL in_diesem_modul;
       RELEASE terminate_moeglich;     |          CALL unterbrechung_verboten;
         CALL in_diesem_modul;         |        END;
       REQUEST terminate_moeglich;     |     END;
       CALL abbruch_verboten;          |
     END;                              |
   RELEASE terminate_moeglich;         |
END;                                   |
terminierung:PROC;                     |     terminierung:PROC;
  REQUEST terminate_moeglich;          |       TERMINATE arbeitstask;
    TERMINATE arbeitstask;             |     END;
  RELEASE terminate_moeglich;          |
END;                                   |
```

Beispiel 6: Terminierung einer Task nur an ungefährlicher Stelle

7.1.4 Suspendierung

Eine Task sollte sich nie selbst suspendieren. Die Suspendierung sollte stets durch eine andere Task erfolgen, die dann auch die zugehörige CONTINUE-Anweisung enthalten sollte. Insbesondere sollte man nie Tasks mit SUSPEND-CONTINUE synchronisieren (Beispiel 7).

Sie muß auf jeden Fall durch die andere Task mit CONTINUE fortgesetzt werden. Falls das zeitlich zu früh erfolgt, bleibt die selbst-suspendierte Task hängen.

```
                              statt
                                |
 DCL zeitgleich SEMA PRESET(0); |
 synchrone_task:TASK;           |   synchrone_task:TASK;
    .......                     |      .......
    REQUEST zeitgleich;         |      SUSPEND;
    ......                      |      ......
 END;                           |   END;
 synchronisierung:TASK;         |   synchronisierung:TASK;
    ......                      |      ......
    RELEASE zeitgleich;         |      CONTINUE synchrone_task;
    ......                      |      ......
 END;                           |   END;
```

Beispiel 7: Synchronisierung zweier Tasks

7.1.5 Ausplanung und Terminierung

Falls eine Task ausgeplant wird, müssen PREVENT- und TERMINATE-Anweisung genau diese Reihenfolge haben.

Bei umgekehrter Reihenfolge kann die Task terminiert werden, obwohl sie noch nicht aktiv ist, und dann aktiviert werden, bevor sie ausgeplant ist.

7.2 Semaphore und Bolts

7.2.1 Zugriff auf modulglobale Objekte

Zugriffe auf modulglobale Objekte, außer Aufrufen von Prozeduren und Aktivierungen von Tasks, sollten stets mit Semaphor- oder Bolt-Anweisungen koordiniert werden.

Falls die Zugriffe zufällig durch zwei Tasks gleichzeitig erfolgen, kann das zu schweren Fehlern führen.

7.2.2 Anfang und Ende kritischer Abschnitte

Anfang und Ende kritischer Abschnitte, d. h. REQUEST und RELEASE, ENTER und LEAVE bzw. RESERVE und FREE müssen stets paarweise außerhalb oder paarweise innerhalb einer REPEAT..END-Schleife bzw. einem Zweig einer IF- oder CASE-Verzweigung liegen (Beispiel 6).

Bei anderer Programmierung wird entweder RELEASE, FREE oder LEAVE häufiger als REQUEST, RESERVE bzw. ENTER ausgeführt, wodurch die Schutzwirkung entfällt, oder seltener als REQUEST usw., wodurch es zu Verklemmungen kommt.

7.2.3 Synchronisation von Tasks

Für die Synchronisation von Tasks sollten Semaphor-Operationen verwendet werden, und nicht etwa SUSPEND und CONTINUE (Beispiel 7).

Falls die Task, die synchronisiert werden soll (warten und gleichzeitig mit einer anderen weiterlaufen), den Wartepunkt (SUSPEND) noch nicht erreicht haben sollte, erfolgt das CONTINUE zu früh und bleibt daher wirkungslos.

8 Ein/Ausgabe

8.1 Fehlerprüfung

Bei jeder Ein- oder Ausgabe sollte mit Hilfe des RST-Formates geprüft werden, ob ein Fehler aufgetreten ist (Beispiel 8).

Dadurch wird verhindert, daß ein Programm infolge eines E/A-Fehlers beendet wird.

8.2 Zeilenumschaltung bei Eingabe

Bei jedem Einlesen von Daten, bei dem danach auf eine neue Zeile umgeschaltet werden muß, sollte das dazu notwendige SKIP nicht am Ende der Einleseanweisung für die Daten, sondern in einer neuen Anweisung stehen, die nur positioniert, aber keine Daten einliest (Beispiel 8).

Da das Einlesen mit dem RST-Format erfolgen soll und daher die Einlese-Anweisung bei Auftreten eines Fehlers sofort verlassen wird, wird im Fehlerfall nicht auf eine neue Zeile umgeschaltet und der Einlesepuffer nicht gelöscht, was zu mehrfachen Einleseversuchen mit den gleichen Daten führen kann.

```
                                        statt
GET zahl FROM datei                       |  GET zahl FROM datei
         BY RST(lesefehler),F(30);        |           BY RST(lesefehler),F(30);
GET FROM datei BY RST(dateiende),SKIP; |
```

Beispiel 8: Einlesen aus Datei mit Prüfung auf Fehler

Inhaltsverzeichnis