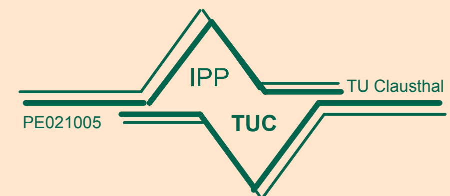
Eine Technik zur Konstruktion sicherer und zuverlässiger Echtzeitsysteme

Peter F. Elzer

Institut für Prozess- und Produktionsleittechnik (IPP)
der Technischen Universität Clausthal (TUC)
Julius-Albert-Str. 6
38678 Clausthal-Zellerfeld

email: elzer@ipp.tu-clausthal.de



Inhalt

1 Einleitung

2 Entwurfskriterien

3 Strukturierte Echtzeitprogrammierung

3.1 "klassische" strukturierte Programmierung

3.2 Strukturierter Parallelismus

3.3 Behandlung von Ablaufanomalien

3.4 Verwaltung von Ressourcen

3.5 Vermeidung von "Deadlocks"

4 Eine Testimplementation

5 Schlußfolgerungen

1 Einleitung
2 Entwurfskriterien
3 Strukturierte Echtzeitprogrammierung
3.1 "klassische" strukturierte Programmierung
3.2 Strukturierter Parallelismus
3.3 Behandlung von Ablaufanomalien
3.4 Verwaltung von Ressourcen
3.5 Vermeidung von "Deadlocks"
4 Eine Testimplementation
5 Schlußfolgerungen

Einleitung

Beim **Betrieb autonomer Systeme** sind **Sicherheit und Zuverlässigkeit von** besonders **hoher Bedeutung**, da sie - wie schon ihr Name sagt - weitestgehend ohne menschliche Intervention arbeiten müssen.

Zwei wesentliche **Aspekte** sind dabei z.B.:

- **Vorhersagbarkeit** des Systemverhaltens und
- **Deadlockfreiheit** .

Außerdem müssen Systeme aus autonomen Subsystemen **unabhängig vom aktuellen "Stand der Technik"** sein, da man nicht alle existierenden Komponenten an jeweils neu hinzukommende Teile anpassen kann.

Glücklicherweise kann man zur Lösung dieser Probleme auf die **Ergebnisse jahrzehntelanger intensiver Forschung in Automatisierungs- und Softwaretechnik** zurückgreifen.

1 Einleitung

2 Entwurfskriterien

3 Strukturierte Echtzeitprogrammierung

3.1 "klassische" strukturierte Programmierung

3.2 Strukturierter Parallelismus

3.3 Behandlung von Ablaufanomalien

3.4 Verwaltung von Ressourcen

3.5 Vermeidung von "Deadlocks"

4 Eine Testimplementation

5 Schlußfolgerungen

Entwurfskriterien

Wesentliche Entwurfsziele der vorgeschlagenen Technik waren also:

- die **Vorhersagbarkeit des Programmverhaltens zu verbessern** und
- über einen **ausreichenden Zeitraum stabil zu bleiben**.

Aber: die **Erfahrung** hat gezeigt, dass es **nicht möglich** ist, irgend eine **Komponente** von Rechnersystemen oder ein **Werkzeug** zu deren Entwicklung **für mehr als wenige Jahre oder außerhalb eines eng begrenzten Anwendungsgebietes zu normen**.

Also mußte die Technik **auf genügend hoher Ebene angesiedelt**

sowie **unabhängig** sein von: **Rechnerhardware**, **Programmiersprache** oder **Betriebssystem**.

Außerdem mußte sie sein : **konzeptionell einfach**, sowie **geeignet für eine grafische Notation** und für **Simulation**.

=> Verallgemeinerung der **"strukturierten Programmierung"**

1 Einleitung

2 Entwurfskriterien

3 Strukturierte Echtzeitprogrammierung

3.1 "klassische" strukturierte Programmierung

3.2 Strukturierter Parallelismus

3.3 Behandlung von Ablaufanomalien

3.4 Verwaltung von Ressourcen

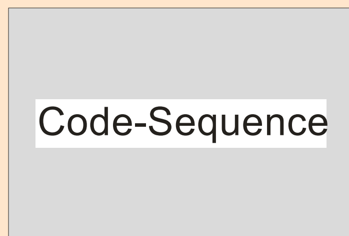
3.5 Vermeidung von "Deadlocks"

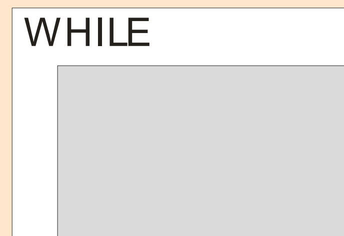
4 Eine Testimplementation

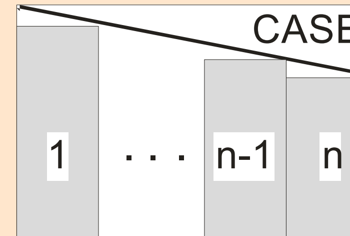
5 Schlußfolgerungen

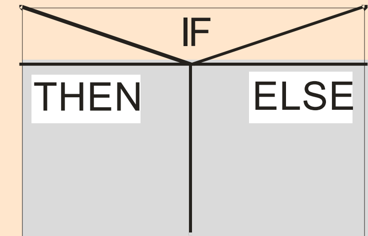
1 Einleitung
2 Entwurfskriterien
3 Strukturierte Echtzeitprogrammierung
3.1 "klassische" strukturierte Programmierung
3.2 Strukturierter Parallelismus
3.3 Behandlung von Ablaufanomalien
3.4 Verwaltung von Ressourcen
3.5 Vermeidung von "Deadlocks"
4 Eine Testimplementation
5 Schlußfolgerungen

"Klassische" strukturierte Programmierung









Ein häufig benutzter Satz der "Nassi-Shneiderman Diagramme"

1 Einleitung

2 Entwurfskriterien

3 Strukturierte Echtzeitprogrammierung

3.1 "klassische" strukturierte Programmierung

3.2 Strukturierter Parallelismus

3.3 Behandlung von Ablaufanomalien

3.4 Verwaltung von Ressourcen

3.5 Vermeidung von "Deadlocks"

4 Eine Testimplementation

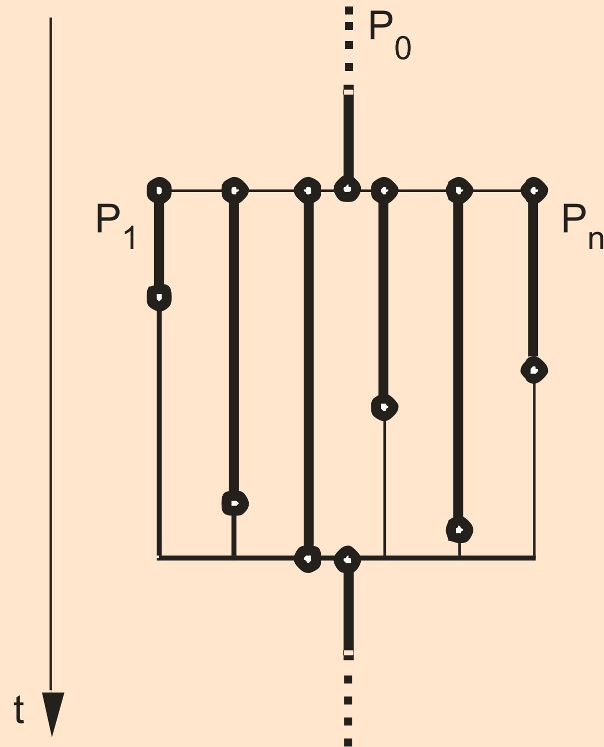
5 Schlußfolgerungen

Strukturierter Parallelismus - Syntaxvorschlag

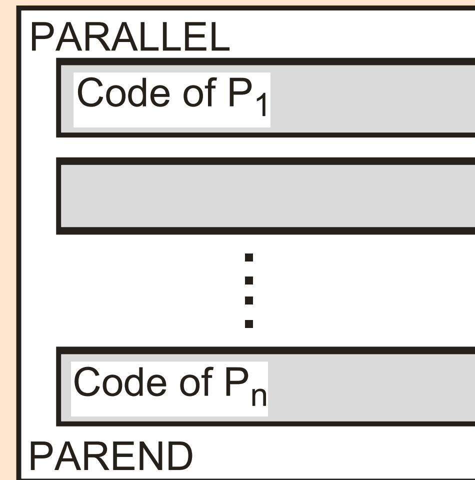
```
parallel
        ,.{execute ppc-name [with resource-list]
        [priority priority-specification]}...
parend
```

Bemerkung: "ppc-name" spezifiziert den "Protoprozess", dessen Code hier ausgeführt werden soll.

Strukturierter Parallelismus - vorgeschlagene grafische Notation



Parallele Ausführung von Prozessen



Grafische Notation

1 Einleitung

2 Entwurfskriterien

3 Strukturierte Echtzeitprogrammierung

3.1 "klassische" strukturierte Programmierung

3.2 Strukturierter Parallelismus

3.3 Behandlung von Ablaufanomalien

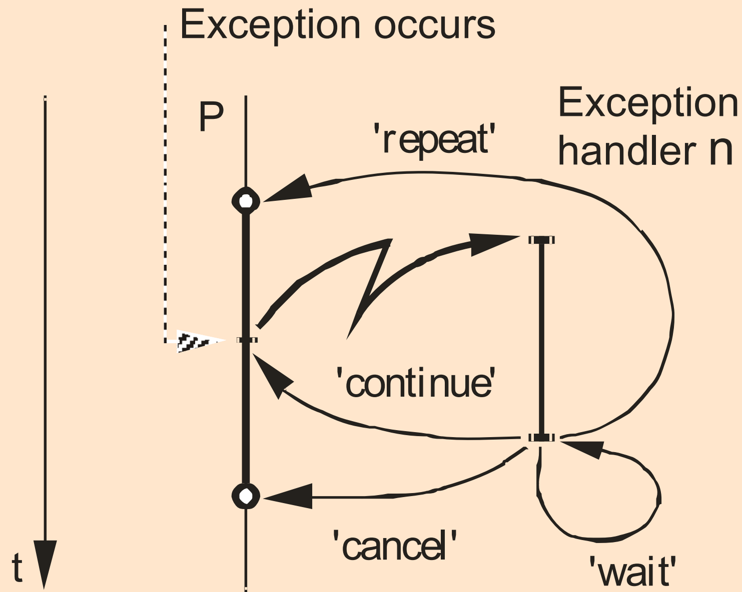
3.4 Verwaltung von Ressourcen

3.5 Vermeidung von "Deadlocks"

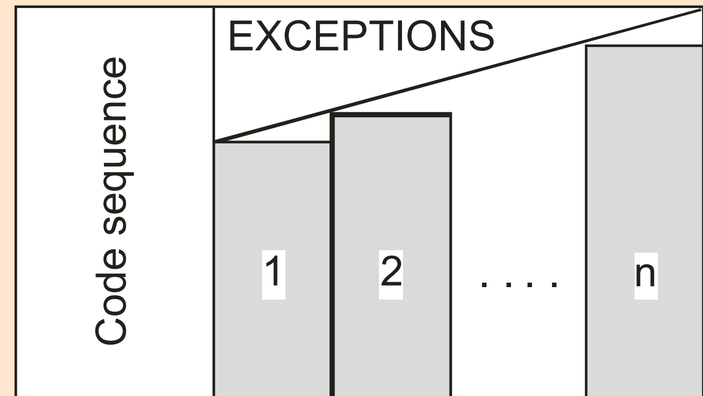
4 Eine Testimplementation

5 Schlußfolgerungen

Behandlung von Ablaufanomalien
- vorgeschlagene grafische Notation



Vier mögliche Reaktionen auf eine Programmablaufanomalie



Grafische Notation

Behandlung von Ablaufanomalien - Syntaxvorschlag

```
begin main-code
      [exceptions,.{on exception-name
            [(exception-handler)]
                  {wait|continue|repeat|nil}}...
      exend]
end
```

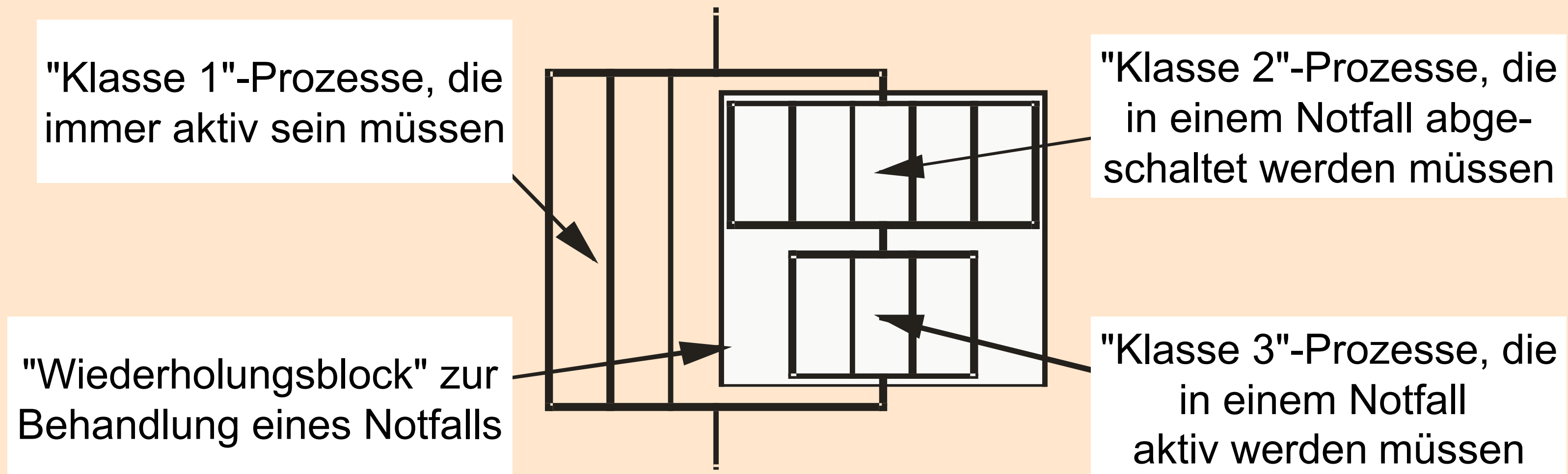
Behandlung von Ablaufanomalien

- Begründung der vier Abschlußcodes:

- **`wait`** (z.B. auf die Verfügbarkeit einer Ressource)
- **`continue`** (z.B. wenn die betreffende Ablaufanomalie im gerade gültigen Kontext bedeutungslos ist)
- **`repeat`** (z.B. wenn die betreffende Ablaufanomalie als vorübergehend betrachtet werden kann)
- nil (wenn der Prozeß abgebrochen werden soll)

Behandlung von Ablaufanomalien - Anwendungsbeispiel 1:

Die Notfallbehandlungsgruppe

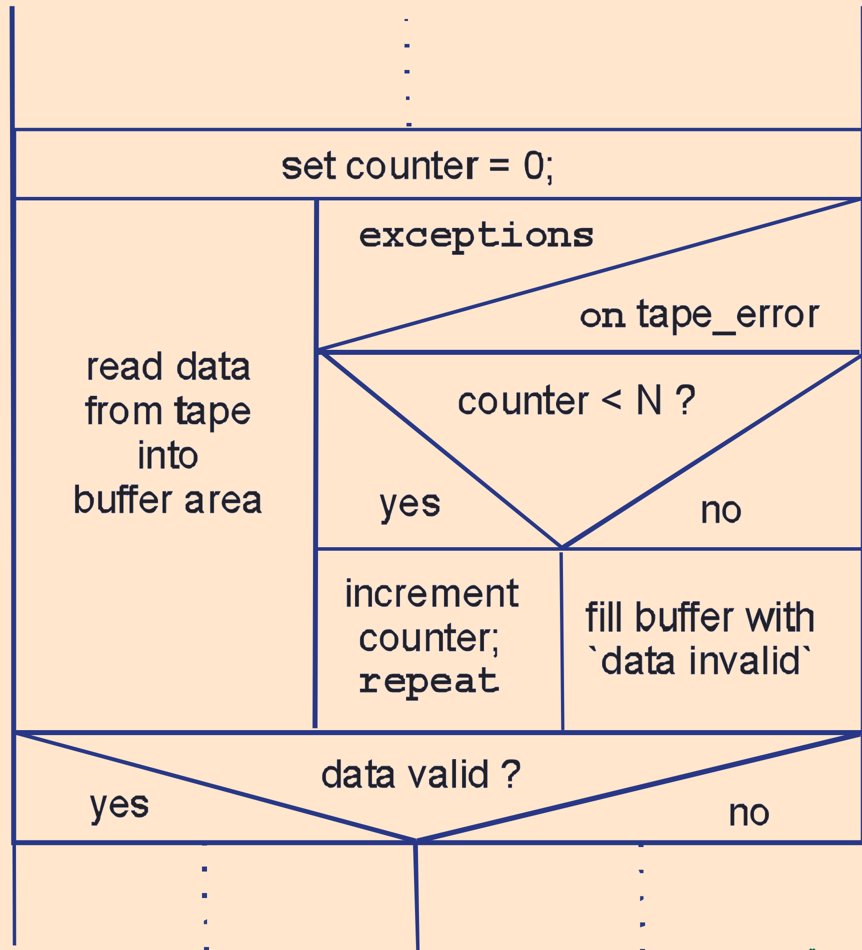


Behandlung von Ablaufanomalien - Anwendungsbeispiel 2:

Ein Magnetband-treiber

Daten sollen von einem Band gelesen werden.

Wenn ein Fehler auftritt, soll der Lesevorgang wiederholt werden, aber nicht öfter als n-mal.



1 Einleitung

2 Entwurfskriterien

3 Strukturierte Echtzeitprogrammierung

3.1 "klassische" strukturierte Programmierung

3.2 Strukturierter Parallelismus

3.3 Behandlung von Ablaufanomalien

3.4 Verwaltung von Ressourcen

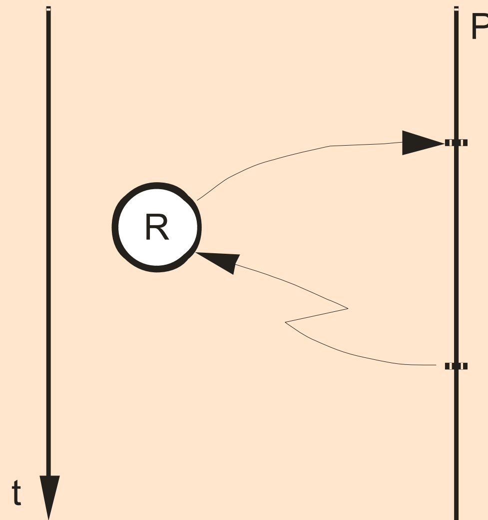
3.5 Vermeidung von "Deadlocks"

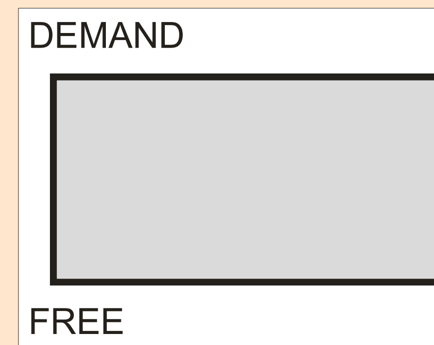
4 Eine Testimplementation

5 Schlußfolgerungen

Verwaltung von Ressourcen

Benutzung von Ressourcen - vorgeschlagene grafische Notation





Prozess P benutzt Ressource R während eines Zeitraums

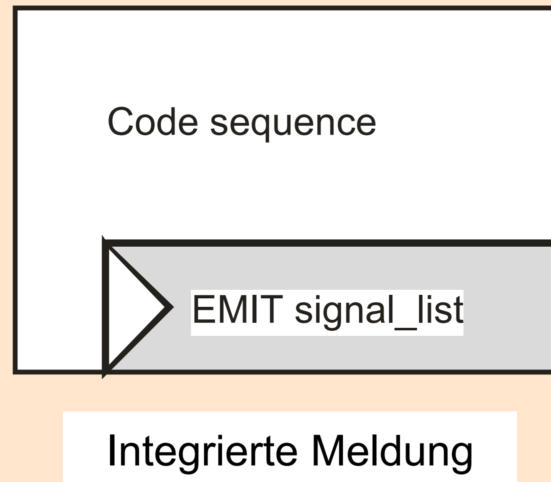
Grafische Notation

Syntaxvorschlag :

```
demand resource-expression [main-code] free
```

Verwaltung von Ressourcen

Integrierte Meldung - vorgeschlagene grafische Notation



Integrierte Meldung

Syntaxvorschlag

```
emit signal-list [main-code] emend
```

1 Einleitung
2 Entwurfskriterien
3 Strukturierte Echtzeitprogrammierung
3.1 "klassische" strukturierte Programmierung
3.2 Strukturierter Parallelismus
3.3 Behandlung von Ablaufanomalien
3.4 Verwaltung von Ressourcen
3.5 Vermeidung von "Deadlocks"
4 Eine Testimplementation
5 Schlußfolgerungen

Vermeidung von Deadlocks

Der **Todfeind** eines autonomen Systems ist der **"deadlock"** ("Systemabsturz"), der üblicherweise eine "reboot" erfordert.

Es scheint, dass nur wenige wissen, dass die **damit verbundenen Entwurfsprobleme schon vor Jahrzehnten gelöst wurden**. Ein Ansatz, der aus diesen Arbeiten hervorging, wird **Haberman** zugeschrieben und kann folgendermaßen zusammengefasst werden:

Ordne alle Ressourcen, die während der Ausführung eines Programmsystems benötigt werden, **in einer linearen Liste**. **Benutze sie nur in dieser Reihenfolge** und gib sie in umgekehrter Reihenfolge wieder frei. Dann ist das Programmsystem "deadlockfrei".

Die vorgeschlagene Technik erlaubt es, die Einhaltung dieser Regel auf einfache Weise zu unterstützen.

Vermeidung von Deadlocks

Der **Gesamtsystementwickler spezifiziert eine lineare Liste aller Ressourcen**, die jemals im System benutzt werden.

Die **Deklaration eines Protoprozesses enthält eine Untermenge dieser Liste**, welche diejenigen **Ressourcen** enthält, **die während der Ausführung** dieses **Prozesses** benötigt werden:

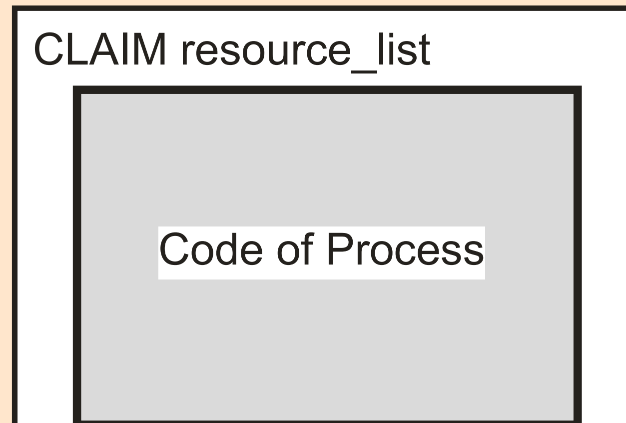
```
ppc-declaration ::= ppc-name task
      [claim resource-list] [complete-code] taskend
```

Die **"resource-expression" in** einer **"demand-clause" enthält** die-jenige **Untermenge** an Ressourcen, **die notwendig ist, um den "main-code"** dieser Anweisung auszuführen.

Der **Präprozessor prüft**, ob alle diese Listen konsistent benutzt werden.

Vermeidung von Deadlocks

Der Protoprozess - vorgeschlagene grafische Notation



Protoprozess mit "Resource Claim"

Eigene Experimente haben den Autor gelehrt: **dies ist eine sehr strenge Regel und nicht immer einhaltbar. Jedoch wirkt** die daraus resultierende **Disziplin während des Systementwurfs außerordentlich erzieherisch**.

1 Einleitung
2 Entwurfskriterien
3 Strukturierte Echtzeitprogrammierung
3.1 "klassische" strukturierte Programmierung
3.2 Strukturierter Parallelismus
3.3 Behandlung von Ablaufanomalien
3.4 Verwaltung von Ressourcen
3.5 Vermeidung von "Deadlocks"
4 Eine Testimplementation
5 Schlußfolgerungen

Eine Testimplementation

Ursprünglich war die vorgeschlagene Technik für ein interaktives grafisches Entwurfswerkzeug entwickelt worden. Es war aber lange Zeit nicht möglich, finanzielle Unterstützung für ein entsprechendes Projekt zu erhalten.

Deshalb entschloss sich der Verfasser im Jahr 2004 zu einer kleinen Test-implementation in Form eines studentischen Projekts.

Ausgewählte Zielsprachen:

1. **"C"**, weil es derzeit in Echtzeitsystemen praktisch "allgegenwärtig" ist,
2. **"PEARL"** (Process and Experiment Automation Real-Time Language), weil es speziell für die Programmierung von Echtzeitsystemen entwickelt wurde.

Andere Kandidatensprachen waren "Java" and "Ada".

Es wäre auch denkbar gewesen, eine objektorientierte Sprache zu verwen-den, aber die Vorhersagbarkeit des Zeitverhaltens von Echtzeitsystemen auf objektorientierter Basis steht noch nicht ausser Zweifel.

Eine Testimplementation

Aus Gründen der Verfügbarkeit wurden folgende Betriebssysteme gewählt:

1 **Real-Time C** mit Siemens **RMOS-32**©
auf einem **Siemens M7**© Prozessrechner

2 **PEARL** mit Betriebssystem **RTOS UH**©
auf einem Emulator für Windows© PC´s.

Da der C-Präprozessor nicht leistungsfähig genug war, wurde **"M4"**© eingesetzt, ein **UNIX**© **Makroprozessor**, der auch für Windows© verfügbar ist. Jedoch mußte die vorgeschlagene Syntax leicht an die Fähigkeiten dieses Werkzeugs angepaßt werden.

Eine Testimplementation   Demonstrationsprogramm

Zwei Prozesse:

der erste
berechnet das Quadrat einer beliebigen eingegebenen Zahl

und sendet ein Signal an
den zweiten,

der das Ergebnis druckt.

Eine Testimplementation In PEARL eingebettete Makros

```
Task(Erzeuger) Claim(Sig);
      FOR i FROM 0 TO MaxWert REPEAT
            Try;
                  Emit(Sig);
                        Wert = i * i;
                  EmEnd;
            Exceptions(EXCEPTION_SignalInUse);
                  Wait;
            ExEnd;
      END;
TaskEnd;
```

Eine Testimplementation

Expansion in PEARL

```
Erzeuger:TASK;
    FOR i FROM 0 TO MaxWert REPEAT
        BEGIN;
            DCL (RTS_ContinueAt1,RTS_ThrownException1,RTS_Wait1) FIXED INIT(0,0,0);
            RTS_TryLabel1:
            CASE RTS_ThrownException1 ALT(0);
                IF RTS_Wait1 == 0 THEN
                    IF NOT(TRY(EmitSema_Sig)) THEN
                        RTS_ContinueAt1 = 1;
                        RTS_ThrownException1 = 1;
                        GOTO RTS_TryLabel1;
                        RTS_ThrowLabel1_1:;
                    FIN;
                ELSE
                    REQUEST EmitSema_Sig;
                FIN;
                Wert = i * i;
                RELEASE ResourceSema_Sig;
            ALT(1);
                RTS_Wait1 = 1;
                RTS_ContinueAt1 = 0;
                RTS_ThrownException1 = 0;
                GOTO RTS_TryLabel1;
            FIN;
            RTS_ExEndLabel_1:
        END;
    END;
    RELEASE RTS_TaskSema(1);
END;
```

Try

Emit

Exception

Wait

Expansionsfaktor: 2

Eine Testimplementation   In C eingebettete Makros

```
Task(Erzeuger) Claim(Sig) {
      int i;
      for (i = 0; i <= MaxWert; ++i) {
            Try;
                  Emit(Sig);
                        Wert = i * i;
                  EmEnd;
            Exceptions(EXCEPTION_SignalInUse);
                  Wait;
            ExEnd;
      }
} TaskEnd;
```

Eine Testimplementation

Expansion in C

```
void _FAR _FIXED Erzeuger(void) {
  do {
    int i;
    for (i = 0; i <= MaxWert; ++i) {
      do {
        int RTS_ContinueAt = 0, RTS_ThrownException = 0, RTS_Wait = 0;
        RTS_TryLabel1:
        switch (RTS_ThrownException) {
          case(0):
            do {
              if (RTS_Wait1 == 0) {
              if (RmGetbinSemaphore(RM_CONTINUE, EmitSema_Sig)!=RM_OK) {
              RTS_ContinueAt = 1;
              RTS_ThrownException = 1;
              goto RTS_TryLabel1;
              RTS_ThrowLabel1_1:;
            }
          } else {
            RmGetBinSemaphore(RM_WAIT, EmitSema_Sig);
          }
        } while (0);
        Wert = i * i;
        RmReleaseBinSemaphore(ResourceSema_Sig);
        .
        .
        .
```

Try

Emit

Expansionsfaktor: 8/3

1 Einleitung
2 Entwurfskriterien
3 Strukturierte Echtzeitprogrammierung
3.1 "klassische" strukturierte Programmierung
3.2 Strukturierter Parallelismus
3.3 Behandlung von Ablaufanomalien
3.4 Verwaltung von Ressourcen
3.5 Vermeidung von "Deadlocks"
4 Eine Testimplementation
5 Schlußfolgerungen

Schlußfolgerungen

An Hand einiger Testprobleme konnte gezeigt werden, dass die vorgeschlagenen "Echtzeitstruktogramme" wirklich funktionieren.

Sogar die **Studenten**, die die Testimplementation realisierten, **erkannten an,** dass sie **leicht zu verstehen** sind und **wenige Quellzeilen ergeben**.

Die **Testimplementation bestätigte, dass sie für mehrere Programmiersprachen** und **Betriebssysteme** brauchbar sind.

Die **Implementation** benötigte **nur wenige Mannmonate** !

Der Verfasser ist deshalb der Meinung, dass die **vorgeschlagene Technik** in der Zukunft ein **nützlicher Beitrag zur Konstruktion sicherer und zuverlässiger autonomer System** sein wird.

Ende

PS: wenn Interesse daran besteht, die im Rahmen der Studienarbeit entwickelten Makros innerhalb der Fachgruppe verfügbar zu machen, will ich mich gerne darum kümmern.