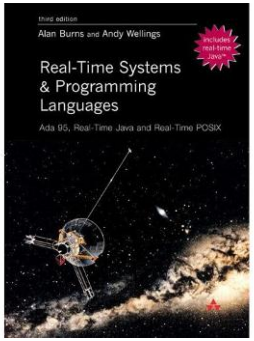
Kapitel 8

Programmiersprachen für Echtzeitsysteme

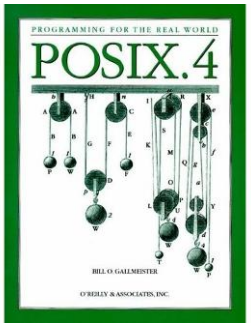
Inhalt

- Motivation
  - Anforderungen von Echtzeitsystemen
  - Geschichte
- PEARL
- Ada
- Real-Time Java
- Zusammenfassung

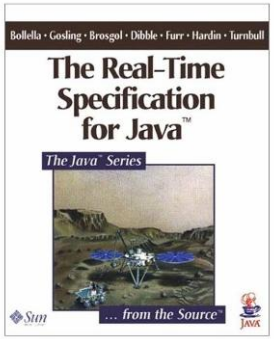
Literatur



A. Burns, A. Wellings: Real-Time Systems & Programming Languages, 2001



B. Gallmeister: POSIX.4 Programming for the Real World, 1995



G. Bollella: The Real-Time Specification for Java, 2000

Paper:

- N. Wirth: Embedded Systems and Real-time Programming, EMSOFT 2001
- Ascher Opler: Requirements for Real-Time Languages, Communications of ACM 1966

Programmiersprachen für Echtzeitsysteme

Anforderungen

Anforderungen

- Die Anforderungen an Programmiersprachen für Echtzeitsysteme fallen in verschiedene Bereiche:
  - Unterstützung bei der Beherrschung komplexer und nebenläufiger Systeme
  - Möglichkeit zur Spezifikation zeitlicher Anforderungen
  - Unterstützung der hardwarenahen Programmierung
  - Erfüllung hoher Sicherheitsanforderungen (Fehlersicherheit)
  - Möglichkeiten zum Umgang mit verteilten Systemen

Beherrschung komplexer nebenläufiger Systeme

- Anforderungen an Programmiersprachen
  - Konstrukte zur Aufteilung der Anwendung in kleinere, weniger komplexe Subsysteme
  - Unterstützung von Nebenläufigkeit (Prozesse, Threads)
  - Daten- und Methodenkapselung in Modulen zur Erleichterung der Wiederverwendbarkeit
  - Eignung für unabhängiges Implementieren, Übersetzen und Testen von Modulen durch verschiedene Personen

Einhalten zeitlicher Anforderungen

- Projektierbares Zeitverhalten
  - Möglichkeit zur Definition von Prioritäten
  - wenig (kein) Overhead durch Laufzeitsystem (z.B. Virtual Machine)
- Bereitstellung umfangreicher Zeitdienste
- Zeitüberwachung aller Wartezustände
- Möglichkeit zur Aktivierung von Prozessen
  - sofort
  - zu bestimmten Zeitpunkten
  - in bestimmten Zeitabständen
  - bei bestimmten Ereignissen

Unterstützung hardwarenaher Programmierung

- Ansprechen von Speicheradressen, z.B. „memory mapped I/O“
- Unterbrechungs- und Ausnahmebehandlung
- Unterstützung vielseitiger Peripherie
- Definition virtueller Geräteklassen mit einheitlichen Schnittstellen
- einheitliches Konzept für Standard- und Prozesse- Ein-/Ausgabe

Erfüllung hoher Sicherheitsanforderungen

- Lesbarkeit, Übersichtlichkeit, Einfachheit durch wenige Konzepte
- Modularisierung und strenge Typüberprüfung als Voraussetzung zur frühen Fehlererkennung durch Übersetzer, Binder und Laufzeitsystem
- Überprüfbare Schnittstellen (-beschreibungen) der Module
- Verifizierbarkeit von Systemen

Sicherheit fängt schon im Kleinen an

- Lexikalische Konventionen können Fehler verhindern.
- Negatives Beispiel: FORTRAN
  - In FORTRAN werden Leerzeichen bei Namen ignoriert.
  - Variablen müssen in FORTRAN nicht explizit definiert werden

- Problem in Mariner 1:
  Aus einer Schleife

```
DO 5 K = 1,3
```

  wird durch versehentliche Verwendung eines Punktes

```
DO5K=1.3
```

  eine Zuweisung an eine nicht deklarierte Variable.
  → Zerstörung der Rakete, Schaden 18,5 Millionen $

Anforderungen durch verteilte Systeme:

- Notwendigkeit vielseitiger Protokolle zur Kommunikation (Feldbus, LAN)
- Unterstützung von Synchronisation auch in verteilten Systemen
- Möglichkeit zur Ausführung von Operationen auf Daten anderer Rechner
- Konfigurationsmöglichkeit zur Zuordnung von Programmen/Modulen zu Rechnern
- Möglichkeit zur automatischen Neukonfigurierung in Fehlersituationen

Programmiersprachen für Echtzeitsysteme

Geschichte

Geschichte: 1960-1970

- 1960-1970
  - Verwendung von Assemblerprogrammen, da der Speicher sehr teuer ist
  - Programme sind optimiert → jedes Bit wird genutzt
- ab ca. 1966
  - erster Einsatz von höheren Sprachen, z.B.
    - CORAL und RTL/2
    - ALGOL 60
    - FORTRAN IV
  - Prozeduraufrufe für Echtzeitdienste des Betriebssystems
  - Probleme:
    - viel Wissen über Betriebssystem notwendig
    - wenig portabel
    - keine semantische Prüfung der Parameter durch den Übersetzer, da keine speziellen Datentypen für Prozesse, Uhren oder Semaphoren existierten) schwierige Fehlersuche

Geschichte: 1970-1980

- Existenz erster Echtzeitsprachen (nationale bzw. internationale Normen):
  - PEARL (Deutschland): Process and Experiment Automation Realtime Language
  - HAL/S (USA)
  - PROCOL (Japan)
  - RT-FORTRAN
  - RT-BASIC
- Neue Datentypen (z.B. task, duration, sema, interrupt) mit zugehörigen Operationen sind in die Sprache integriert
- Einführung einheitlicher Anweisungen vor Ein-/Ausgabe und die Beschreibung von Datenwegen

Geschichte: 1970-1980

- Vorteil:
  - Benutzerfreundliche Sprachelemente
  - Prüfung der Semantik der Parameter bei Betriebssystemaufrufen durch Übersetzer möglich
  - Weitgehende Portabilität
- Nachteil: geeignete Betriebssysteme sind nicht vorhanden Möglichkeiten
  1. Entwicklung eines eigenen Betriebssystems → hohe Entwicklungskosten
  2. Anpassung eines vorhandenen Standardbetriebssystems → Gefahr der Existenz überflüssiger Teile im Betriebssystem, eingeschränkte Portabilität

Geschichte ab 1978

- universelle sichere hohe Sprachkonzepte für alle Anwendungsbereiche
  - Standardisierung, inbesondere durch Department of Defense (DOD): Ada
  - Datentypen (z.B. task, duration, interrupt) oder systemabhängige Parameter werden in sprachlich sauberer Weise mittels Module /Packages eingebunden
- Beispiele:
  - Ada83,Ada95
  - CHILL
  - PEARL, PEARL 90, Mehrrechner.PEARL

Geschichte heute:

- Trend hin zu universellen Sprachen (z.B. C,C++ oder Java) mit Bibliotheksprozeduren für Echtzeitdienste angereichert (z.B. POSIX), aber auch modellgetriebener Entwicklung
- herstellerspezifische Speziallösungen für eingeschränkte Anwendungsbereiche, z.B.
  - Prüfsysteme
  - Standardregelungsaufgaben
  - Förderungstechnik
  - Visualisierung (Leitstand)
  - Telefonanlagen
- Beispiele:
  - SPS-Programmierung (Speicherprogrammierbare Steuerung)
  - ATLAS (Abbreviated Test Language for All Systems) für Prüfsysteme (v.a. Flugzeugelektronik)
  - ESTEREL

Programmiersprachen für Echtzeitsysteme

PEARL

Daten

- Process and Experiment Automation Real-Time Language
- DIN 66253
- Ziele:
  - Portabilität
  - Sicherheit
  - sichere und weitgehend rechnerunabhängige Programmierung
- lauffähig z.B. unter UNIX, OS/2, Linux
- Versionen: BASIC PEARL (1981), Full PEARL (1982), Mehrrechner PEARL (1988), PEARL 90 (1998)
- http://www.irt.uni-hannover.de/pearl/

Eigenschaften

- strenge Typisierung
- modulbasiert
- unterstützt (prioritätenbasiertes) Multitasking
- E/A-Operationen werden von eigentlicher Programmausführung separiert
- Synchronisationsdienste: Semaphore, Bolt-Variablen
- Zugriff auf Unterbrechungen
- erleichterte Zeitverwaltung

Erläuterung für folgendes Beispiel

- **Modularität**: Anwendungen können in einzelne Module aufgeteilt werden (`MODULE`, `MODEND`).
- Aufspaltung in System- und Problemteil:
  - **Systemteil** (`System;`): Definition von virtuellen Geräten für alle physischen Geräte, die das Modul benutzt. Der Systemteil muss auf den entsprechenden Computer angepasst sein
    → Hardwareabhängigkeit
  - **Problemteil** (`PROBLEM;`): eigentlicher, portabler Programmcode
- Sonstige Notationen typisch für prozedurale Sprachen:
  - Kommentare `!,/*…*/`
  - Semikolon zur Terminierung von Anweisungen

Grundstruktur

```
/*Hello World*/
MODULE Hello;
    SYSTEM;
        termout: STDOUT;

    PROBLEM;
        DECLARE x FLOAT;
        T: TASK MAIN;
        x := 3.14;                    !PI
        PUT 'Hello' TO termout;
    END;
MODEND;
```

Datentypen

- Variablen werden durch `DECLARE` deklariert und mittels `INIT` initialisiert.
- Durch das Schlüsselwort `INV` werden Konstanten gekennzeichnet.
- Die temporalen Variablen bieten eine Genauigkeit von einer Millisekunde
- Die Genauigkeit der Datentypen kann angegeben werden
- Zeiger auf Datentypen werden unterstützt

Schlüsselwort	Bedeutung	Beispiel
FIXED	Ganzzahlige Variable	-2
FLOAT	Gleitkommazahl	0.23E-3
CLOCK	Zeitpunkt	10:44:23.142
DURATION	Zeitdauer	2 HRS 31 MIN 2.346 SEC
CHAR	Folge von Bytes	'Hallo'
BIT	Folge von Bits	'1101'B1

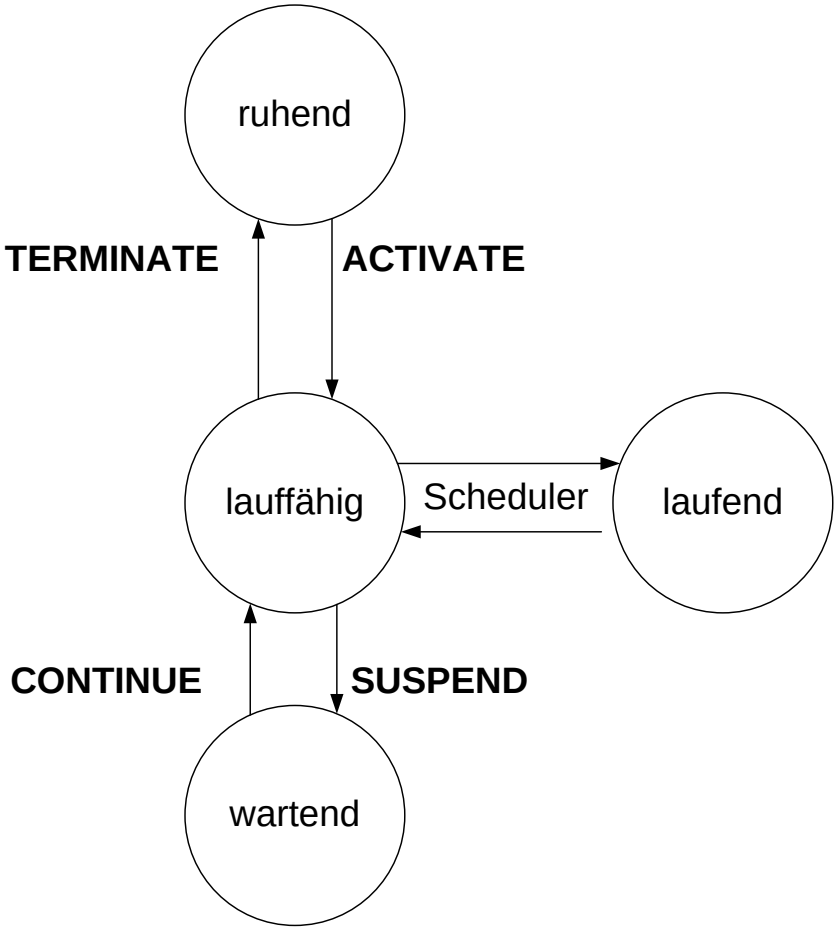
Prozessmodell

- Initial sind alle Prozesse bis auf MAIN ruhend
- Zustandswechsel sind unter Angabe einer exakten Zeit möglich:

```
AFTER 5 SEC ACTIVATE Task1;

AT 10:15:0 ALL 2 MIN
     UNTIL 11:45:0
     ACTIVATE Student;
```

- Scheduling präemptives, prioritätenbasiertes Schedulingverfahren mit Zeitscheiben (Round-Robin)
- Zuweisung der Prioritäten durch den Benutzer
- Zeitscheibenlänge abhängig vom Betriebssystem



Prozess-Synchronisation

- Zur Synchronisation bietet PEARL Semaphore und Bolt-Variablen:
  - Semaphore (Datentyp: SEMA):
    - Deklaration wie bei einer Variablen
    - Operationen REQUEST und RELEASE zum Anfordern und Freigeben des Semaphores
    - Mittels der Operation TRY kann versucht werden den Semaphore nicht blockierend anzufordern
    - Es werden keine Möglichkeiten zur Vermeidung von Prioritätsinversion geboten
  - Bolt-Variablen (Datentyp: BOLT):
    - Bolt-Variablen besitzen wie Semaphoren die Zustände belegt und frei und zusätzlich einen 3. Zustand: Belegung nicht möglich
    - RESERVE und FREE funktionieren analog zu Semaphore-Operationen REQUEST bzw. RELEASE
    - exklusive Zugriffe mit RESERVE haben Vorrang von (nicht exklusiven) Zugriffen mit ENTER (Freigabe mit LEAVE)
    - Eine elegante Formulierung des Leser-Schreiber-Problems mit Schreiberpriorisierung ist damit möglich

Lösung: Leser-Schreiber-Problem mit Schreiberpriorisierung

```
PROBLEM;
     DECLARE content CHAR(255); ! Speicher: 255 Bytes
     DECLARE key BOLT;                ! Default: frei

    LESER1: TASK;
          DECLARE local1 CHAR(255);
          ENTER key; local1=content; LEAVE key;
          …
    END;

    LESER2: TASK;
          DECLARE local2 CHAR(255);
          ENTER key; local2=content; LEAVE key;
          …
    END;

    SCHREIBER1: TASK;
          DECLARE newcontent1 CHAR(255);
          …
          RESERVE key; content=newcontent1; FREE  key;
    END;

    SCHREIBER2: TASK;
          DECLARE newcontent2 CHAR(255);
          …
          RESERVE key; content=newcontent2; FREE  key;
    END;
```

Unterbrechungen

- Durch das Schlüsselwort WHEN können Prozesse aktiviert oder fortgesetzt werden.
- Es ist möglich Unterbrechungen durch DISABLE/ENABLE zu sperren bzw. freizugeben.
- Beispiel: Student 2 weckt Student 1 beim Eintreffen der Unterbrechung auf.

```
MODULE Vorlesung:
   System;
        alarm: IR*2;

   PROBLEM;
        SPECIFY alarm INTERRUPT;

   student2: TASK PRIORITY 20;
        WHEN alarm ACTIVATE student1;
        DISABLE alarm;
        ...
        ENABLE alarm;
   END;
MODEND;
```

Programmiersprachen für Echtzeitsysteme

Ada

Einleitung

- 1970 von Jean Ichbiah (Firma Honeywell Bull) entworfen
- Durch das Department of Defense (DOD) gefördert
- Mitglied der Pascal Familie
- Häufige Verwendungen für Systeme mit hohen Anforderungen an die Sicherheit.
- Bis 1997 mussten alle Systeme im Rahmen von DOD-Projekten mit einem Anteil von mehr als 30% neuen Code in ADA implementiert werden.
- Versionen: Ada 83, Ada 95
- Freie Compiler sind verfügbar: z.B. http://www.adahome.com
- http://www.ada-deutschland.de/

Eigenschaften

- Sicherheit durch
  - sehr **strenges Typsystem**
  - zahlreiche Prüfungen zur Laufzeit: z.B. zur Erkennung von Speicherüberläufen, Zugriff auf fremden Speicher, Off-by-One-Fehlern
  - Verhinderung von Fehlern bei nebenläufiger Programmierung (durch Rendezvous-Konzept, geschützte Typen)
- Unterstützung der modularen Programmierung (insbesondere auch **information hiding**, also Aufteilung separate Schnittstellen und Implementierung)
- Unterstützung der Ausnahmebehandlung
- Eignung zur Implementierung generischer Systeme
- Ab Ada 95:
  - objektorientierte Programmierung
  - dynamische Polymorphie
- Offener Standard: http://www.adaic.org/standards/95lrm/html/RM-TOC.html

Strukturierung

- Die Programme können beliebig in Blöcke/ Unterprogramme/ Pakete/ Tasks aufgeteilt werden.
- Pakete und Tasks müssen, Unterprogramme können in eine Spezifikation (head) und einen Rumpf (body) aufgeteilt werden
  - Kopf: Definition der Schnittstellen und Variablen auf die andere Pakete/Tasks/Unterprogramme zugreifen können
  - Rumpf: private spezifiziert lokale Objekte, deren Realisierung verborgen bleiben
  - Der Anweisungsteil des Pakets wird einmalig beim Abarbeiten der Paketdeklaration ausgeführt
- Benutzung von Paketen:
  - Durch den Befehl `WITH` kann ein Paket benutzt werden.

```
PACKAGE <name> IS
   <sichtbare Vereinbarungen>;
   [PRIVATE <Vereinbarungen>;]
END <name>


PACKAGE BODY <name> IS
   <lokale Vereinbarungen>;
BEGIN
   <Anweisungen>
   [EXCEPTION
   <Ausnahmebehandler>]
END <name>;
```

Generische Einheiten

- Durch das Schlüsselwort `GENERIC` können Unterprogramme/Pakete als Programmschablonen implementiert werden.
- Parameter sind Objekte und Objekttypen
- Freie Parameter werden bei der Übersetzung durch aktuelle Parameter ersetzt (entspricht Templates in C++)

Spezifikation:

```
GENERIC
   TYPE sometype IS PRIVATE;

PACKAGE queue_handling IS
   TYPE queue (maxlength: NATURAL)
   IS PRIVATE;

PROCEDURE enqueue (q: IN OUT queue;
   elem: IN sometype);
   ...

PROCEDURE dequeue ...

PRIVATE
   SUBTYPE index IS CARDINAL RANGE
        0..1000;
   …
```

Benutzung:

```
DECLARE
   PACKAGE int_queue
   IS NEW queue_handling (INTEGER);
   …
```

Prozesse

- Prozesse (Datentyp TASK) werden wie Variablen behandelt:
  - Verwendung als Komponenten von Feldern oder Records möglich.
  - Verwendung als Parameter erlaubt.
- Der Spezifikationsteil darf ausschließlich die Deklaration von Eingängen (Schlüsselwort `ENTRY`) enthalten.
  - Ein Eingang ist ein Bestandteil eines Tasks, der von außen aufgerufen werden kann.
  - Es ist zu jedem Zeitpunkt immer nur höchstens ein Eingang aktiviert. In der Zwischenzeit eintreffende Aufrufe werden in einer Warteschlange eingereiht.

```
TASK [TYPE] name IS
   ENTRY ename (<Parameter>);
   ENTRY ...
END name;


TASK BODY name IS
   <deklarationen>
BEGIN
   …
   ACCEPT ename(<Parameter>) DO
        …
   END ename;
   …
   EXCEPTION [<exception handler>]]
END name;
```

Lebenszyklus eines Prozesses

- Start:
  - Prozesse werden automatisch beim Abarbeiten der Deklaration aktiv, aber erst am Ende des Deklarationsteils gestartet.
  - Durch die Blockstruktur können Prozessaufrufe geschachtelt auftreten.
- Beendigung:
  - Es gibt nur die Operation `ABORT` zum Datentyp TASK (gewaltsames Beenden)
  - Prozesse terminieren automatisch beim Erreichen des Blockendes, falls sie nicht auf das Ende von untergeordneten Prozesse warten müssen.
  - Der umfassende Prozess wird durch implizite Synchronisation des Betriebssystems erst beendet, wenn alle in ihm deklarierten und damit alle gestarteten Prozesse beendet sind.
  - Ein Block wird erst verlassen, wenn alle in ihm vereinbarten Prozesse beendet sind.

Partitionen

- Seit Ada 95 werden auch Partitionen unterstützt.
- Eigenschaft einer Partition:
  - Partitionen haben einen eigenen Adressraum
  - Partitionen können Prozesse enthalten
  - Die Programme können durch Partitionen auf verschiedenen Rechnern ausgeführt werden
  - Aktive Partitionen enthalten Prozesse und main()
  - Passive Partitionen enthalten nur Daten und/oder Unterprogramme
  - Eine Partition wird erst beendet, wenn all ihre Prozesse beendet sind
  - Partitionen werden von außen oder durch einen sogenannten Environment-Task angestoßen, bei deren Abarbeitung, die in ihr enthaltene Main-Prozedur aufgerufen wird
  - Zur Kommunikation zwischen Partitionen können RPC oder gemeinsame Daten einer dritten Partition benutzt werden.

Prozess-Synchronisation: Rendezvous-Konzept

- Ada bietet mit Rendezvous ein Konzept zur synchronen Kommunikation:
  - Definition eines Eingangs (`ENTRY`) in einem Prozess
  - `ACCEPT`-Anweisung zu den Eingängen in den Prozessen
  - Der Aufruf des Eingangs eines anderen Prozesses erfolgt wie ein Prozeduraufruf mit Parametern.
  - Die Ausführung erfolgt erst, wenn beide Prozesse bereit sind: der externe Prozess den Aufruf durchführt und der eigentliche Prozess die `ACCEPT`-Anweisung erreicht.
  - Sowohl der aufrufende als auch der aufgerufene Prozess warten, bis die Anweisungen im `ACCEPT`–Block durchgeführt sind.
  - Alternatives Warten durch `SELECT` mit Guards (`WHEN`).
  - Eine zeitliche Begrenzung der Wartezeit (watchdog) ist möglich.
- Eine ausführliche Beschreibung ist unter http://www.ada-deutschland.de/AdaTourCD2004/ada_dokumentation/paralleleprozesse/10_6_rendezvous.html zu finden.

Beispiel: Realisierung eines gemeinsamen Speichers (Leser-Schreiber-Problem mit Schreiberpriorität)

- Grundgerüst des Codes:
  - Deklaration eines generischen Datentyps `item`
  - Das Paket `sharedmemory` biete nach außen die beiden Funktionen `readProc` und `writeProc` an.
  - Intern wird der Speicher in der Variablen `item` gesichert.
  - Zusätzlich besitzt das Paket einen Prozess `control`, der den Zugriff auf die Variable `value` überwacht.

```
GENERIC
        TYPE item IS PRIVATE

PACKAGE sharedmemory IS
        PROCEDURE readProc(x: OUT item)
        PROCEDURE writeProc(x: IN item)
END;

PACKAGE BODY sharedmemory IS
        value: item;

  TASK control IS
        ... (siehe folgende Folien)
  END control;
  PROCEDURE readProc(x:OUT item) IS
  BEGIN
        ... (siehe folgende Folien)
  PROCEDURE writeProc(x:IN item) IS
  BEGIN
        ... (siehe folgende Folien)
END sharedmemory;
```

Beispiel: Fortsetzung

- Schnittstelle des Prozesses `control`: der Prozess bietet insgesamt drei Funktionen als Rendezvous an: `start`, `write`, `stop`
- Die Prozedur `readProc` benutzt die Schnittstelle `start` zum Signalisierung des Lesebeginns und `stop` zur Signalisierung der Beendigung.
- Die Prozedur `writeProc` benutzt die Schnittstellenfunktion `write`.
- Unterschied zwischen `read` und `write`: mehrere Leser dürfen gleichzeitig auf die Daten zugreifen, aber nur ein Schreiber.

```
TASK control IS
        ENTRY start;
        ENTRY stop;
        ENTRY write(x:in item);
END control;

PROCEDURE readProc(x:OUT item) IS
BEGIN
        control.start;
        x:=value;
        control.stop;
END read;

PROCEDURE writeProc(x:IN item) IS
BEGIN
        control.write(x);
END write;
```

Beispiel (Fortsetzung): Code des Prozesses `control`

- Die Anzahl der aktuellen Leser wird in der Variable `readers` gespeichert
- Bevor ein Prozess lesend auf den Speicher zugreifen darf, muß er erstmalig beschrieben werden
- Im Anschluß führt der Prozess eine Endlosschleife mit folgenden Möglichkeiten aus:
  1. Falls kein Schreiber auf den Schreibzugriff wartet (`WHEN write'count=0`), so wird ein Lesewunsch akzeptiert und die Anzahl der Leser erhöht, sonst wird der Wunsch bis zur Ausführung des Schreibwunsches verzögert (**Schreiberpriorität**).
  2. Beendet ein Leser den Zugriff, so wird die Anzahl erniedrigt.
  3. Falls kein Leser mehr aktiv ist (`WHEN readers=0`), werden Schreibwünsche akzeptiert, ansonsten wird dieser verzögert.
- Entscheidend: Die Auswahl zwischen den Rendezvous-Alternativen erfolgt nicht deterministisch (durch Würfeln)

```
TASK BODY control IS
    readers: integer :=0;
BEGIN
    ACCEPT write(x:IN item) DO
        value:=x;
    END;
    LOOP
        SELECT
            WHEN write'count=0 =>
               ACCEPT start;
            readers:=readers+1;
        OR
            ACCEPT stop;
            reader:=readers-1;
        OR
            WHEN readers=0 =>
               ACCEPT write(x:IN item) DO
               value:=x;
            END write;
        OR
            DELAY 3600.0;
            exit;
        END SELECT;
    END LOOP;
END control;
```

Wechselseitiger Ausschluss

- In Ada95 bietet zum wechselseitigen Ausschluss geschützte Typen (`PROTECTED TYPE`):
  - Die Objekte können Typen und Daten sowie die benötigten Operationen (Funktionen, Prozeduren, Eingänge) enthalten.
  - Das Laufzeitsystem sichert, dass Prozeduren in einem `PROTECTED TYPE` exklusiv ausgeführt werden.
  - Auf lesende Funktionen (`FUNCTION`) in einem `PROTECTED TYPE` können mehrere Prozesse gleichzeitig zugreifen.
  - Prioritätsvererbung wird bei geschützten Typen unterstützt.
  - Beim Auftreten von Ausnahmen wird der Block verlassen und die Belegung automatisch aufgehoben

→ Vorgehen ähnelt Monitoren

Beispiel: Realisierung eines Semaphors

```
PROTECTED TYPE sema (init:
        INTEGER := 1) IS
   ENTRY P;
   PROCEDURE V;
   PRIVATE
     count: INTEGER := init;
END sema;


PROTECTED BODY sema IS
   ENTRY P WHEN count > 0 IS
   BEGIN
        count := count - 1;
   END P;

PROCEDURE V IS
   BEGIN
        count := count + 1;
   END V;
END sema;
```

Benutzung:

```
s : sema;
...
s.P;
... -- Exklusive Anweisungen
s.V;
```

Ausnahmen

- Ausnahmen können in Anweisungen, bei Deklarationen und im Rendezvous auftreten
- Der Benutzer kann Ausnahmen selbst definieren: `exc1: EXCEPTION`
- Ausnahmen können durch `RAISE` ausgelöst werden, die Behandlung erfolgt typischerweise am Ende des Rahmens.
- Beim Auftreten einer Ausnahme wird der Rahmen verlassen und die entsprechende Behandlung gestartet.
- Ist keine Behandlung angegeben, so wird die Ausnahme an den umgebenden Rahmen weitergeleitet (exception propagation), bis eine Behandlung oder ein Programmabbruch erfolgt

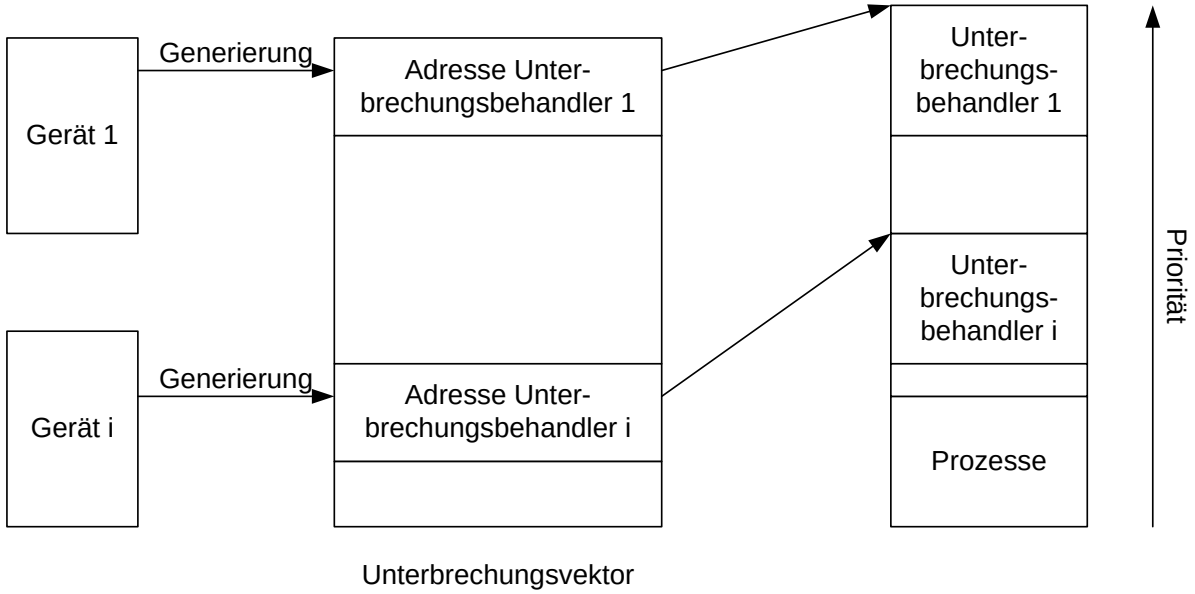
- Syntax der Behandlung

```
EXCEPTION
      WHEN exceptionname =>
           <Anweisungsfolge>;
      ...
      WHEN OTHERS =>
           <Anweisungsfolge>;
```

- Mit `OTHERS` können beliebige Ausnahmen behandelt werden
- Es gibt viele vordefinierte Ausnahmen:
  - `CONSTRAINT_ERROR`
  - `NUMERIC_ERROR`
  - `PROGRAMM_ERROR`
  - `STORAGE_ERROR`
  - `TASKING_ERROR`

Unterbrechungen

- Zur Behandlung von Unterbrechungen können PROTECTED PROCEDURES verwendet werden.
- Diese Prozeduren werden mit hoher Priorität (abhängig vom Betriebssystem, höher als Prozesspriorität) exklusiv ausgeführt.
- Die Zuordnung der Prozeduren zu den Unterbrechungen erfolgt statisch oder dynamisch
- Die möglichen Unterbrechungen sind im implementierungsabhängigen Paket `Ada.Interrupt.Names` beschrieben.



Unterbrechungen: statisch vs. dynamisch

```
PROTECTED PROCEDURE alarm IS
    PROCEDURE response;
        -- parameterlos
    PRAGMA ATTACH_HANDLER
        (response, Alarm_ID);
END alarm;
PROTECTED PROCEDURE BODY alarm IS
    PROCEDURE response IS
    ...
    END response;
END alarm;
```

```
PROTECTED PROCEDURE alarm IS
    PROCEDURE response;
        --parameterlos
    PRAGMA INTERRUPT_HANDLER
        (response);
END alarm;
PROTECTED PROCEDURE BODY alarm IS
    -- wie oben
    -- spaeterer Prozeduraufruf:
    ATTACH_HANDLER(alarm.response,
    Alarm_ID);
    …
END alarm;
```