§ 6 Programmiersprachen für die Prozessautomatisierung

Kapitel 6 - Lernziele

- Die Vorgehensweisen bei der Erstellung von Prozessautomatisierungs-programmen kennen
- Programmiersprachen nach Sprachhöhe unterscheiden können
- Die Programmiersprachen für SPSen kennen
- Einfache Beispiele in SPS-Sprachen programmieren können
- Die wichtigsten Echtzeit-Konzepte für Programmiersprachen kennen
- Wissen, wie Echtzeit-Konzepte in Ada 95 umgesetzt sind
- Ein Echtzeit-Programm in Ada 95 entwerfen können
- C/C++ bezüglich Echtzeit einschätzen können
- Wissen, worauf die Portabilität von Java beruht
- Die Echtzeit-Erweiterungen von Java verstanden haben

Folie 2

Tr1 In den vorangehenden Kapiteln 1 bis 5 wurden alle Grundlagen erläutert, die für die Realisierung von Echtzeit-Systemen notwendig sind. Es fehlt lediglich die konkrete Programmier-Umsetzung der dort vorgestellten Konzepte. Dies wird zum Abschluss der Vorlesung in Kapitel 6 behandelt. Bei Abschluss des Kapitels sollte man die unterschiedlichen Arten der Programmiersprachen für die unterschiedlichen Automatisierungscomputer kennen. Der Schwerpunkt dabei liegt auf den Speicherprogrammierbaren Steuerungen und der Echtzeit-Programmierung von PCs.
Traumüller; 13.11.2003

§ 6 Programmiersprachen für die Prozessautomatisierung

Vorgehensweisen bei der Erstellung der Programme

Speicherprogrammierbare Steuerungen
- textuelle Programmiersprachen
- graphische Programmiersprachen

Mikrocontroller
- Assembler
- niedere maschinenunabhängige Programmiersprachen

PC und IPC
- Softwarepakete
- universelle Echtzeitprogrammiersprachen

Prozessleitsysteme
- Funktionsbausteintechnologie

Folie 3

Tr2 Es gibt je nach Art oder Zweck des Automatisierungscomputers bei der Erstellung von Programmen unterschiedliche Vorgehensweisen an welchen man sich orientieren kann. Diese verschiedenen Vorgehensweisen und die dabei benutzen Programmiersprachen sollen hier besprochen werden. Der erste Schritt dazu wird in Form eines Überblicks in Kapitel 6.1 geschaffen.
Traumüller; 13.11.2003

Arten von Programmiersprachen

Klassifikation nach Art der Notation

–	textuelle Programmiersprachen	Ada, C, SPS-Anweisungsliste
–	graphische Programmiersprache	SPS-Kontaktplan

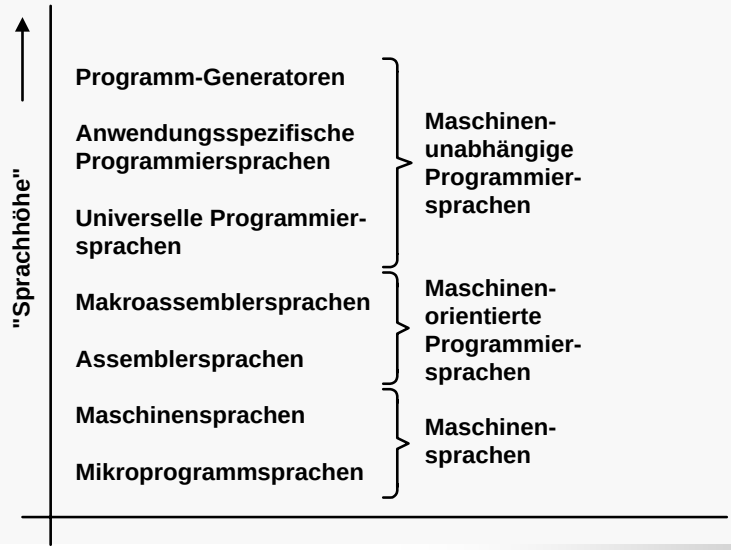
Klassifikation nach dem Programmiersprachenparadigma

–	prozedurale Programmiersprachen	C, Ada 83
–	funktionale Programmiersprachen	LISP
–	logische Programmiersprachen	PROLOG
–	objektorientierte Programmiersprachen	C++, Smalltalk, Ada 95

Klassifikation nach der Sprachhöhe

–	**hoch:**	an der Verstehbarkeit durch den Menschen orientiert
–	**nieder:**	an den Hardware-Eigenschaften eines Computers orientiert

Klassifizierung nach der “Sprachhöhe”



Mikroprogrammsprachen

– Realisierung der Ablaufsteuerungen zur Ausführung der Maschinenbefehle
  - fest verdrahtete Verknüpfungsglieder
  - Mikroprogramme
– Mikroprogramme (Firmware)
  - Speicherung in schnellen Schreib-/Lesespeicher (RAM's) oder in Festwertspeicher (ROM's)
  - nicht zugänglich für Anwenderprogrammierung
– Maschinensprachen
  - Sprachelemente: **Befehle und Daten in Form von Bitmustern**
  - Zusammenfassung als Oktal-bzw. Hexadezimalzahl
  - mühsame Handhabung
  - nicht für Anwendungsprogrammierung

Assemblersprachen

Ziel:
Vermeidung der mühsamen Handhabung der Maschinensprachen unter Beibehaltung der Eigenschaften der Maschinenbefehle

– Ersetzung der oktalen/hexadezimalen Schreibweise des Operationsteils der Befehle durch symbolische, mnemotechnisch günstige Buchstabenabkürzungen
– Einführung eines symbolischen Namens anstelle der zahlenmäßigen Darstellung des Adressteils
– Eindeutige Zuordnung zwischen den Befehlen der Assemblersprache und den Befehlen der Maschinensprache
– Abhängigkeit von gerätetechnischen Eigenschaften der jeweiligen Rechenanlage

Makroassemblersprachen (Makrosprachen)

– weiteres Hilfsmittel zur einfacheren Handhabung: Makros

Makro: Abkürzung für eine bestimmte Befehlsfolge

– Unterscheidung
- Makrodefinition
- Makroaufruf
- Makroexpansion

– Aufbau einer Makrodefinition

MAKRO Makroname (P1, P2, ..., PN)
Makrokörper
Endzeichen

– Makroaufruf

Makroname (A1, A2, ..., AN)

– Eindeutige Zuordnung von Makroassemblerbefehlen zu Befehlen der Maschinensprache

– Einem Makrobefehl entsprechen mehrere Maschinenbefehle

Beispiel Makro Dreiadressadd

Makrodefinition:
MAKRO Dreiadressadd (P_1, P_2, P_3)
LADE P_1
ADDIERE P_2
SPEICHERE P_3
ENDE

Makroaufruf:
Dreiadressadd (A,B, SUMME)

Makroexpansion:
...
LADE A
ADDIERE B
SPEICHERE SUMME
...

Unterschied zwischen Unterprogramm-Aufrufen und Makros

– Unterprogramm wird nur einmal gespeichert, kann mehrfach aufgerufen und verwendet werden

– Makro wird an jeder Stelle an der es aufgerufen wird, expandiert

Klassifizierung von Makros

– Standardmakros: fest vorgegeben

– Anwendermakros: vom Anwender selbst definierbare Makros für häufig vorkommende Befehlsfolgen

Universelle Programmiersprachen

universell $\triangleq$ **nicht auf ein Anwendungsgebiet ausgerichtet**

universelle niedere Programmiersprachen — **Systemprogrammier-sprachen**

– Zweck: Erstellung von Systemprogrammen
  - Compiler
  - Editoren
  - Betriebssysteme
  - Treiberprogramme

– Ziel: 1. Ausnutzung der Hardwareeigenschaften
2. Portabilität

– Beispiel: C

universelle höhere Programmiersprachen

– Zweck: Erstellung von allgemeinen Programmen

– Ziel: 1. einfache Formulierbarkeit
2. umfangreiche Compilerprüfungen
3. Portabilität

– Beispiel: Ada, Java, Smalltalk

Anwendungsspezifische Sprachen

Deskripitive Sprachen, nicht-prozedurale höhere Sprachen, very high level languages

Unterscheidung zu prozeduralen Sprachen

– keine Beschreibung des Lösungsverfahrens, sondern Beschreibung der Problemstellung selbst
– Einschränkung auf bestimmtes Anwendungsgebiet

Bsp.:	FUP	Funktionsplan
	KOP	Kontaktplan
	AWL	Anweisungsliste
	EXAPT	für Werkzeugmaschinensteuerungen
	ATLAS	für automatische Prüfsysteme

Vorteile/Nachteile:

+	**bequeme, anwendungsspezifische Ausdrucksweise**
-	**Inflexibilität**

Programm-Generatoren (Fill-in-the-blancs-Sprachen)

– Formulierungsverfahren für Programme

– Beantwortung von Fragen in Form von Menues am Bildschirm durch den Anwender (Konfigurierung)

– Umsetzung der Antworten durch den Programm-Generator in ein ausführbares Programm

– Vorteil: keine Programmierkenntnisse notwendig

– Nachteil:
- **Einengung auf bestimmtes Anwendungsgebiet**
- **Abhängigkeit von einem bestimmten Hersteller**

Anwendungsgebiete von Programm-Generatoren im Bereich Prozessautomatisierung

– Leittechniksysteme in der Energie-und Verfahrenstechnik
  - TELEPERM-M (Siemens)
  - PROCONTROL-B (ABB)
  - CONTRONIC-P (Hartmann & Braun)

– Speicherprogrammierbare Steuerungen in Form von Anweisungsliste oder Kontaktplan
  - SIMATIC (Siemens)

§ 6 Programmiersprachen für die Prozessautomatisierung

Folie 16

Tr3 Eine mögliche Art zur Programmierung von Prozessautomatisierungs-Systemen sind höhere Programmiersprachen. Hier wird diskutiert, was solche Sprachen sind, was von diesen erwartet wird und was diese Sprachen von anderen Programmiersprachen unterscheidet, also was die typischen Eigenschaften dieser Programmiersprachen sind.
Als erstes wird die Problematik der Echtzeit-Programmierung erläutert. Als nächstes wird erklärt, wann man Assemblerprogrammierung benötigt und wann höhere Programmiersprachen einsetzbar sind.
Traumüller; 13.11.2003

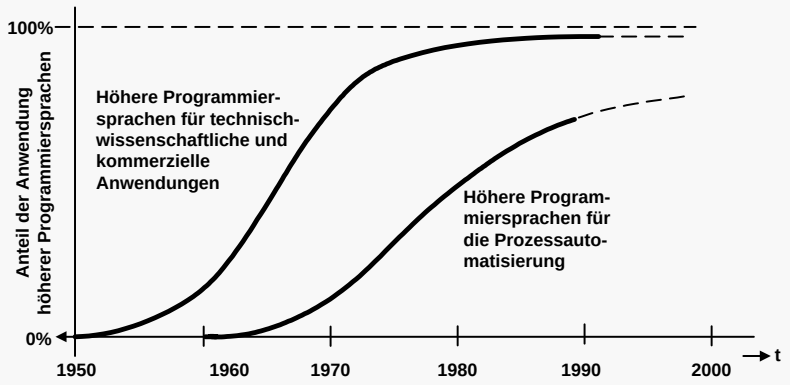
Problematik der Echtzeit-Programmierung

– Höhere Programmiersprachen wie BASIC, FORTRAN, COBOL, PASCAL sind für Echtzeit-Programmierung ungeeignet
  - keine Echtzeitsprachmittel
  - keine Einzelbitoperationen
  - keine Befehle für Prozess-Ein-/Ausgabe

– Einsatz von höheren Programmiersprachen im Vergleich zu Assemblersprachen bringt Erhöhung von Speicherbedarf und Rechenzeit mit sich
  - Produktautomatisierung: Speicherplatz und Rechenzeit kritisch
  - Anlagenautomatisierung: Rechenzeit unter Umständen kritisch

Voraussetzung
- Compiler für Zielrechner
- Echtzeit-Betriebssystem

Vergleich der Verwendung höherer Programmiersprachen



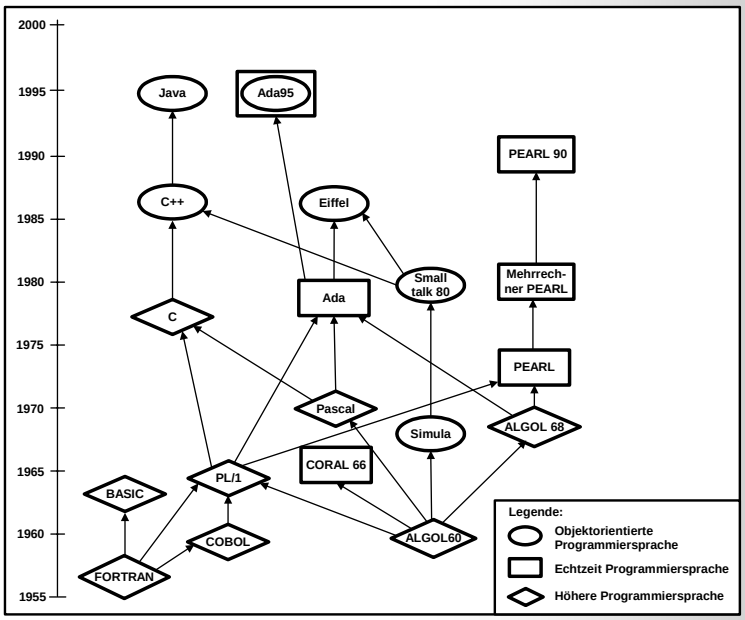
Vor- und Nachteile der Assemblerprogrammierung

+	Speicher- und Rechenzeit-Effizienz
-	höhere Programm-Entwicklungskosten
-	geringe Wartbarkeit
-	Probleme mit der Zuverlässigkeit
-	schlechte Lesbarkeit, geringer Dokumentationswert
-	fehlende Portabilität

Einsatz von Assemblersprachen

ja: Automatisierung von Geräten oder Maschinen, wo es um Serien- oder Massenanwendungen geht, kleine Programme

nein: langlebige, große zu automatisierende Prozesse



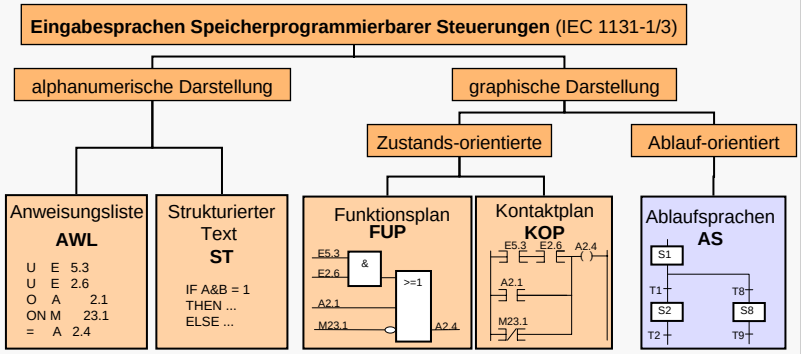
§ 6 Programmiersprachen für die Prozessautomatisierung

Programmiersprachen für SPS-Systeme (1)

– Keine feste Programmiersprache für SPS-Systeme

– Unterschiedliche Arten der Programmierung, auch abhängig vom Hersteller

– IEC 1131 definiert grafische und textuelle Grundsprachen



– IEC 1131 definiert keine Befehle!

⇨ sowohl deutsche als auch englische Bezeichner für Operatoren

Folie 21

Tr4 In Kapitel 6.3 wird diskutiert, wie die Programmierung von SPS-Systemen durchgeführt werden kann und welche Sprachmittel dafür eingesetzt werden können. Wie bereits in vorangehenden Kapiteln ausführlich dargestellt sind Speicherprogrammierbare Steuerungen ein in der Industrie häufig eingesetzter und damit sehr wichtiger Automatisierungscomputer. Sie weisen dabei auch in der Programmierung einige Besonderheiten zur bekannten Programmierung von normalen PCs auf.
Traumüller; 13.11.2003

Programmiersprachen für SPS-Systeme (2)

– Basis aller Programmiersprachen sind Logikverknüpfungen
– Erweiterung um Möglichkeiten der Zeitverarbeitung
– Darstellungsart je nach Aufgabenstellung unterschiedlich geeignet
  • Zustandsorientierte Programmteile besser in FUP oder KOP
  • Ablauforientierte Programmteile besser in AWL oder AS
– Darstellungsarten lassen sich ineinander überführen

ABER:

Bestimmte Operationen nur in AWL programmierbar (Bit-Schiebeoperationen)

Einführungsbeispiel:
Ausgang 1(A1) und Ausgang 2 (A2) sind nur dann gesetzt, wenn entweder Eingang 3 (E3) gesetzt ist oder wenn die beiden Eingänge1 (E1) und 2 (E2) gleichzeitig gesetzt sind.

Anweisungsliste (AWL) / Instruction List (IL)

– Assemblerähnliche Programmiersprache
– Alle Funktionen einer SPS uneingeschränkt programmierbar
– Einheitlicher Aufbau einer Befehlszeile

```
Marke(opt.):   Operator  Operand     Kommentar(opt.)
```

– Programmierung durch Verknüpfung von Signalen

Umsetzung des Beispiels in AWL (Tafelanschrieb)

```
Start:    U(    E1
          U     E2
            )
          O     E3
          =     A1
          =     A2
```


Strukturierter Text (ST)

– Höhere, Pascal-ähnliche Sprache
  • Zuweisungen
    z.B.: alarm := ein AND aus;
  • Unterprogrammaufrufe
    z.B.: alarmleuchte(S:=ein, R:=aus);
  • Kontrollanweisungen
    z.B.: IF alarm
    THEN alarmleuchte(S:=ein, R:=aus);
    END_IF;

– Zusätzliche Sprachmittel für Zeitverarbeitung und Prozessdatenzugriff
– Für die Programmierung umfangreicher Systeme geeignet

Umsetzung des Beispiels in ST (Tafelanschrieb)

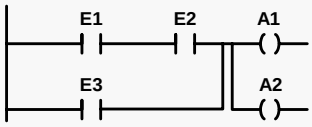
```
A1 :=    E3 OR (E1 AND E2);
A2 :=    A1;
```


Kontaktplan (KOP) / Ladder Diagramm (LD)

– Einfache Darstellung
– Lehnt sich an Stromlaufplan der Relaistechnik an
– Symbole für „Schließer“, „Öffner“, und „Relaisspule“

– Symbolisierter Stromfluss von links nach rechts
– I/O-Zustände werden auf Schalterzustände abgebildet
– Programm wird von oben nach unten gelesen

Umsetzung des Beispiels in KOP (Tafelanschrieb)



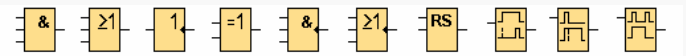
Nachteil: Komplexe mathematische Funktionen schlecht darstellbar

Funktionsbausteinsprache (FBS) / Funktionsplan (FUP)

– Lehnt sich an bekannte Symbole für Funktionsbausteine (DIN40900) an

– Symbolumfang nicht auf logische Grundelemente beschränkt

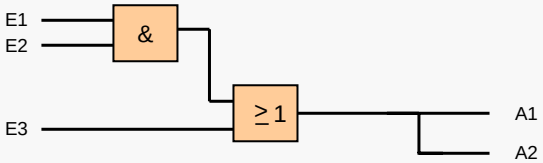
⇨ Merker, Zähler, Zeitgeber und frei definierbare Blöcke möglich



– I/O-Zustände werden direkt als Ein-/Ausgangssignale verwendet

– Übersichtliche Darstellung

Umsetzung des Beispiels in FUP (Tafelanschrieb)



§ 6 Programmiersprachen für die Prozessautomatisierung

Folie 28

Tr5 Die Programmiersprache Ada 95 hat sowohl Echtzeit- als auch objektorientierte Eigenschaften. An Hand ihrer Geschichte soll gezeigt werden wie die Entwicklung einer solchen Programmiersprache von Statten geht. Zudem werden an ihrem Beispiel Sprachkonstrukte betrachtet, die hinsichtlich Echtzeit-Eigenschaften besonders wichtig sind.

Traumüller; 13.11.2003

Lady Ada Lovelace

Ada
Name der 1. Programmiererin
Zu Ehren der Mathematikerin Augusta Ada Byron, Countess of Lovelace, Tochter von Lord Byron benannt. Ada Lovelace (1815- 1851) arbeitete mit Charles Babbage an seiner “Difference-and Analytic-Engine”. Auf Ada Lovelace geht die Idee zurück, diese Maschine mit Lochkarten zu programmieren.

Eigenschaften von Ada

- Eignung für sehr umfangreiche Softwaresysteme bis über 10^7 Zeilen
  - Modularisierungskonzept
  - Trennung von Spezifikation und Implementierung
  - Unterstützung der Wiederverwendung von Softwarekomponenten
  - Fehlerbehandlungskonzepte als Teil des Module-Codes
- Echtzeit-spezifische Eigenschaften
  - Unterstützung absoluter und relativer Zeitanforderungen
  - Ausführung paralleler Abläufe
  - Priorisierung und Scheduling von parallelen Abläufen
- Umfangreiche Prüfung jedes Ada-Compilers auf die Einhaltung der Norm
  - Mehrere tausend Testprogramme

Ada95 Programmeinheiten

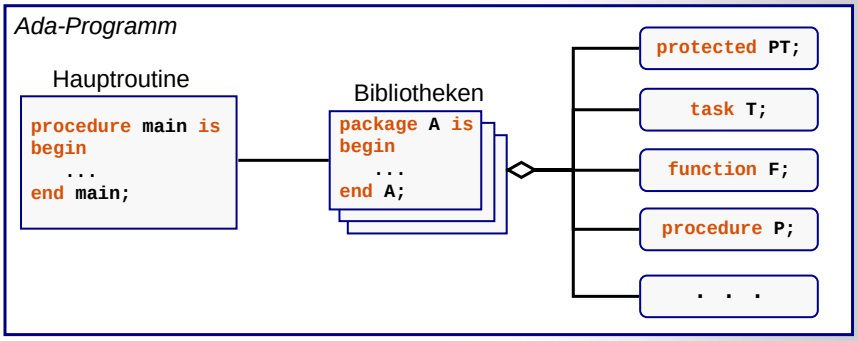
– Unterprogramme
  - Durch Aufruf ausführbare Anweisungsreihenfolge
    ➔ **Funktionen (function) und Prozeduren (procedure)**

– Parallele Abläufe
  - Abgeschlossene, parallel ausführbare Programmteile
    ➔ **Rechenprozesse (tasks)**

– Gemeinsam genutzte Betriebsmittel mit synchronisiertem Zugriff
  - Datenstrukturen mit Zugriffskontrolle durch Monitore
    ➔ **Geschützte Variablen (protected variables)**

– Wiederverwendbare Bibliotheken
  - Zusammenstellung unterschiedlicher, sachlich zusammengehöriger Unterprogramme, Tasks und geschützten Typen
    ➔ **Pakete (packages)**

Aufbau von Ada95-Programmen



– Kein explizites Hauptprogramm
  ⇨ Bestimmung eines Unterprogramms als Hauptroutine

– Aufruf von Programmeinheiten aus der Hauptroutine heraus

– Programmeinheiten in Bibliotheken abgelegt oder direkt deklariert

Aufbau von Ada95-Programmeinheiten

- Spezifikation **= Schnittstelle nach außen**
- Rumpf **= Realisierung der Programmeinheit**

Paketspezifikation (package declaration)

nach außen sichtbarer Teil

```
package package_name is

   Vereinbarungen
   (Types, Unterprogramme,Tasks,...)

   private
    privater Teil (Vereinbarungen)

end package_name;
```

Vereinbarungen

Paketrumpf (package body)

nach außen nicht sichtbar

```
package body package_name is

   weitere Vereinbarungen,
   Unterprogramm-Rümpfe,

   begin

     Anweisungen

end package_name;
```

Anweisungen

Beispiel: Realisierung von Programmeinheiten

- Trennung von Spezifikation und Implementierung in separaten Dateien

Spezifikation (Datei *General.ads*)

```
package General is

 protected Variable is

  function read return Float;

  private
   protectedData: Float:= 0.0;
 end Variable;

...

end General;
```

von außen sichtbar

Nach außen nicht sichtbar

Implementierung (Datei *General.adb*)

```
package body General is

protected body Variable is

   function read return Float is
   begin
     return protectedData;
   end read;
 end Variable;

...

end General;
```

Implementierung der Anweisungen

Konzepte der Parallelprogrammierung

- Notwendigkeit auf gleichzeitig stattfindende Ereignisse reagieren zu können
  - ⇨Konzept „paralleler“ Rechenprozesse zur Bearbeitung paralleler Abläufe
- Direkte Unterstützung von Rechenprozesses in Ada durch eigenes Sprachmittel “Task”

Programmeinheit “Task”

- Task als parallel ablaufende, abgeschlossene Befehlssequenz
- Task als gekapselte Einheit
  - ⇨ Deklarationen innerhalb einer Task sind nicht nach außen sichtbar
- Kommunikation zwischen Tasks unterschiedlich realisierbar
  - ⇨ „message passing“ und „shared variables“

Realisierung von Tasks

- Spezifikation und Implementierung von Tasks in Unterprogrammen oder in Paketen möglich
- Taskeinplanung sobald „begin“ der übergeordneten Einheit erreicht wird
- Ausführungsende und Terminierung wenn “end”-Anweisung der Task erreicht

```
package package_name is
  task T1;
  task T2;
end package_name;

package body package_name is
  task body T1 is
  begin
    ...
  end T1;
  task body T2 is
  begin
    ...
  end T2;
end package_name;
```

Einplanung von T3

```
with package_name; use package_name;

procedure main_procedure_name is

  task T3;

  task body T3 is
    ...
  end T3;

begin
  ...
end main_procedure_name;
```


Task-Typen und Task-Objekte

- Task-Typen definieren Vorlagen für die Instanziierung von Task-Objekten
- Tasks des selben Typs besitzen ähnliche Eigenschaften und Funktionalität
- Individuelle Task-Objekte durch Parametrierung bei der Instanziierung

Einbindung des Pakets „General“

```
package General is

  task type monitorValue(X: Integer);

  task body monitorValue(X: Integer) is
  begin
    -- Implementierung
    ...
  end monitorValue;

end General;
```

Task-Typ Spezifikation / Implementierung

```
With General; use General;
procedure main is

  M1 : monitorValue(1);
  M2 : monitorValue(2);
  M3 : monitorValue(3);

begin
  -- Programmcode
  ...
end main;
```

Deklaration von Task-Objekten

Synchronisierung von Tasks

Synchronisierungskonzepte in Ada95

- Logische Synchronisierung
  - Definition einer Ablaufreihenfolge der Tasks

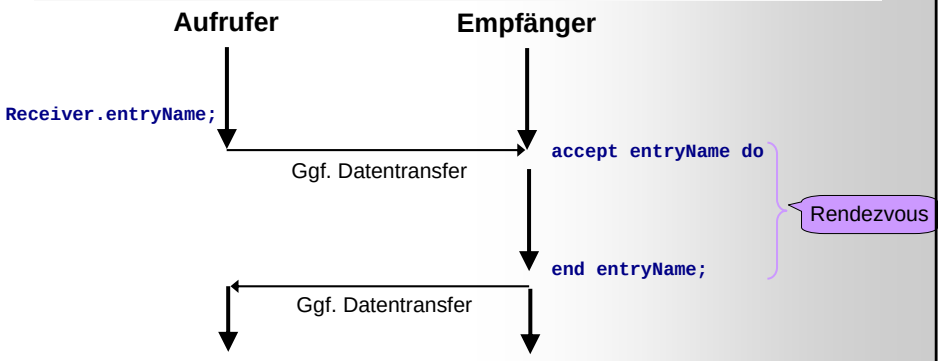
➔ **Rendezvous-Konzept**

- Betriebsmittel-orientierte Synchronisierung
  - Regeln für den Zugriff auf gemeinsam genutzte Betriebsmittel

➔ **Geschützte Betriebsmittel (protected units)**

Rendezvous-Konzept (1)

- Handshake-Verfahren
  - Festlegung zeitlicher Synchronisierungspunkte zwischen Tasks
  - Beiderseitiges Warten der Tasks
- Rendezvous hat aus Aufrufersicht die Gestalt eines Prozeduraufrufs
- Definition von Aufrufvereinbarung (entry) und Aufrufpunkt (accept)



Rendezvous-Konzept (2)

- Deklaration der Aufrufvereinbarung in Task-Spezifikation
- Aufrufe und Aufrufpunkte im Task-Rumpf implementiert
- Funktionsprinzip: Task ruft Aufrufpunkt einer anderen Task auf

Aufrufvereinbarung

```
task T1 is
 entry sync;
end T1;

task body T1 is
 begin

  ...

  T2.sync;
  accept sync;

end T1;
```

```
task T2 is
 entry sync;
end T2;

task body T2 is
 begin

   accept sync;
   ...

   T3.sync;

 end T2;
```

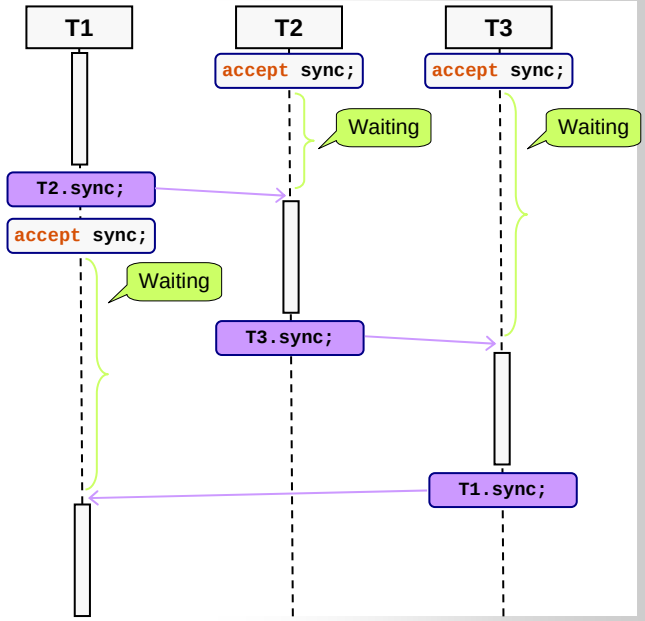
Aufrufpunkt

```
task T3 is
 entry sync;
end T3;

task body T3 is
 begin

   accept sync;
   ...

   T1.sync;

 end T3;
```

Aufruf eines anderen Aufrufpunkts

➔ **Beispiel realisiert feste Ablaufreihenfolge T1, T2, T3, T1….**

Beispielablauf



Select-Anweisung

– Realisierung alternativer Abläufe mittels “select”
  - Selektive Synchronisierung
  - Zeitbedingtes Warten

Selektive Synchronisierung

```
select
  accept entryName1;
 or
  accept entryName2;
  ...
 else
 -- Alternative code
end select;
```

Ausführung, wenn kein Aufruf vorliegt

```
select
  Receiver.entryName;
 else
 -- Alternative code
end select;
```

Sofortige Ausführung wenn kein Aufruf möglich

Zeitbedingtes Warten

```
select
  accept entryName1;
 or
  accept entryName2;
  ...
 or
  delay time_out;
 -- Alternative code
end select;
```

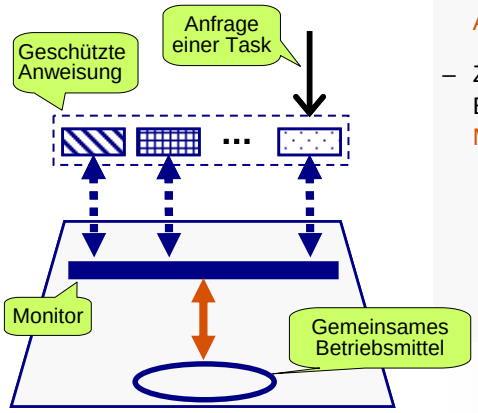
Ausführung wenn kein Aufruf innerhalb “time_out”

```
select
  Receiver.entryName;
 or
  delay time_out;
 -- Alternative code
end select;
```

Ausführung wenn kein Rendezvous innerhalb “time_out” möglich

Geschützte Betriebsmittel (1)

- Problem: Zugriff auf gemeinsam genutzte Betriebsmittel
- Monitor-Konzept für exklusiven Zugriff



- Schnittstelle als geschützte Anweisung
- Zugriff auf gemeinsames Betriebsmittel durch internen Monitor geregelt
  - Geschützte Anweisungen werden nicht parallel ausgeführt
  - Sequenzielle Abarbeitung von gleichzeitigen Anfragen

➔ **Exklusiver Zugriff auf gemeinsame Betriebsmittel**

Geschützte Betriebsmittel (2)

Geschützte Anweisungen

```
protected S is
  entry P;
  procedure V;
  function G return Boolean;

private
  Sem : Boolean := True;
end S;

protected body S is
  entry P when Sem = True is
    Sem := False;
  end P;

  procedure V is
  begin
    Sem := True;
  end V;
  ...
end S;
```

Aufruf-bedingung

- Geschützte Anweisungen schließen sich gegenseitig aus
- Eine Prozedur kann gemeinsames Betriebsmittel beliebig verändern

➔ **Funktionen besitzen nur Lesezugriff**

- Aufrufpunkte als geschützte Anweisungen
- Alle Variablen von geschützten Einheiten müssen als „private“ deklariert werden

Kommunikation zwischen Tasks (1)

- Synchrone Kommunikation
  - Erweiterung der Rendezvous-Synchronisierung um Datenaustausch
    - Gegenseitiges Warten
    - Datenaustausch im Zuge des Rendezvous
    - Daten nur innerhalb des Rendezvous gültig

Beispiel einer synchronen Kommunikation

```
task body T1 is
begin
  T2.entryName(I:Integer);
end T1;
```

Entry-Aufruf mit Datentransfer

```
task body T2 is
begin
  accept entryName(I:Integer)do
    ...
  end entryName;
end T1;
```

T1 während des Rendezvous blockiert

➔ **Prinzip des “message passing”**

Kommunikation zwischen Tasks (2)

- Asynchrone Kommunikation
  - Datenaustausch mittels gemeinsamer Variablen
    - Geschützte Operationen für Zugriff
    - Kein Warten/Blockieren bei Kommunikation

Beispiel einer asynchronem Kommunikation

```
protected body M is
 procedure write(I:Integer);
 function read return Integer;
private
 Val:Integer;
end T1;
```

```
task body T1 is
begin
  M.write(I:Integer);
end T1;
```

```
task body T2 is
temp: Integer;
begin
 temp := M.read;
 ...
end T1;
```

➔ **“shared variable” und “protected operations”**

Zeitoperationen

- Verzögerung des Ablaufs
  - Bis zu einem bestimmten Zeitpunkt
  - Für eine feste Zeitdauer
- Ada-Befehle: “delay” bzw. “delay until”
- Zeiteinheiten und Operationen im Standardpaket definiert
  - Zeiteinheiten für Standard-Anwendungen: Sekunden (seconds)
  - Zeiteinheiten für Echtzeit-Anwendungen: ms, µs, ns
  - Operationen
    - Addition und Subtraktion von Zeitvariablen
    - Auslesen der Systemzeit
- Während einer Verzögerung führt der Prozessor eine andere Task aus
- Nach Ablauf einer Verzögerung evtl. weitere Verzögerungen bis zur erneuten Prozessorzuteilung

  ⇨ Ausführungsunterbrechung durch höherpriore Task

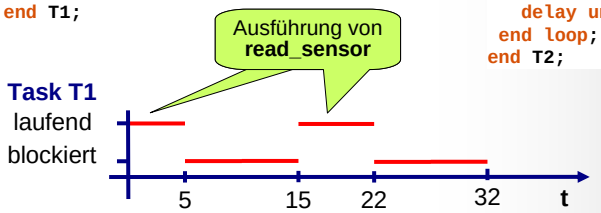
Beispiele für Zeitoperationen

```
task T1 is
 ms: Duration := 0.001;

begin
 loop
  read_sensor;
  delay 10*ms;
 end loop;
end T1;
```

Ausführungs-dauer nicht konstant

```
task T2 is
 ms: Duration := 0.001;
 next_call: Time;

begin
 loop
   next_call := Clock + 10*ms;
   read_sensor;
   delay until next_call;
 end loop;
end T2;
```

aktuelle Zeit



- Aufruf von “read_sensor” durch Task T1 10ms nach der letzten Ausführung

  ⇨ Zeit zwischen Aufrufen hängt von der Ausführungsdauer ab
- Aufruf von “read_sensor” alle 10ms durch Task T2

  ⇨ Unterprogramm wird zu festen Zeitpunkten aufgerufen

Modularisierte Ausnahmebehandlung

– Ausnahmefehler auf Grund irregulärer Operationen möglich
  ⇨ Ausnahmebehandlung (exception handling)

“Exception Handling” Konzept in Ada95

– Ausführung eines “Exception Handling Block” bei Auftreten einer Exception
– Exception Handling Blocks direkt in der Programmeinheit implementiert
– Mehrere Exceptions können in einem Handling Block behandelt werden
– Nicht behandelte Exceptions werden an übergeordnete Einheit übergeben
– Exceptions durch Laufzeitumgebung oder Programmcode ausgelöst
  ➔ Laufzeitumgebung: vordefinierte Exceptions
  ➔ Programmcode: Anwender-definierte Exceptions

Beispiel für “Exception Handling”

```
procedure main is
begin
 A:= readSensor(1);
 B:= readSensor(2);

 rate:= A/B;
 -- Could be a division by 0
 exception
   when Constraint_Exception =>
   -- Handles divisions by 0
      ...
   when others =>
   -- Handles all other exceptions
      ...
end main;
```

```
function readSensor (X:Integer)
return Float is
begin
   temp := read(X);
   if (temp < 0.0) then
     raise Sensor_Exception;
     -- Custom defined exception
   else
     return temp;
   end if;
   exception
     when Sensor_Exception =>
     -- Handles sensor exceptions
        ...
end readSensor;
```

– Spezifische Blocks mit implementierten “Exception Handler”
  • “Sensor Exceptions” werden innerhalb der Funktion behandelt, andere werden übergeben
  • Ausführung von speziellen Code bei Auftreten einer „Constraint_Exception“
  • Vorhersagbares Systemverhalten auch bei Ausnahmefällen

Echtzeiterweiterungen in Ada95 (1)

- Tasks mit dynamischen Prioritäten
  - Basispriorität wird auf Basis der Spezifikation festgelegt
  - Tatsächliche Priorität kann davon abweichen
    - ➔ **Bsp. bestimmt durch "Priority inheritance protocol"**
- Hinweis: In Ada bedeutet eine niedrigere Zahl eine niedrigere Priorität!

Spezifikation

```
task T1 is
pragma Priority(10);
end T1;

task T2 is
pragma Priority(1);
entry sync;
end T1;
```

Implementierung

```
task body T1 is
begin
 T2.sync;
end T1;

task body T2 is
begin
 accept sync do
   ...
 end sync;

 Set_Priority(20);
end T2;
```

T1 ruft T2 auf

Basispriorität

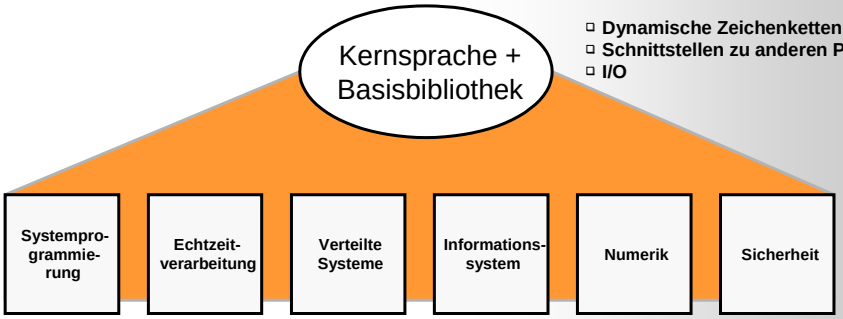
T2 wird mit der Priorität von T1 ausgeführt

Dynamischer Prioritätswechsel

Echtzeiterweiterungen in Ada95 (2)

- Scheduling-Verfahren
  - Scheduling-Verfahren kann frei festgelegt werden
    - Pragma-Anweisung "Task_Dispatching_Policy"
    - Standard-Verfahren: Feste Prioritäten, FIFO bei gleicher Priorität
  - Weitere Verfahren können implementiert und dann verwendet werden
- Zugriff auf geschützte Einheiten per "Priority Ceiling Protocol"
  - Prioritätsschranke von geschützten Einheiten kann in deren Spezifikation definiert werden
    - Default-Wert: Höchste Systempriorität
  - Prioritätsvererbung garantiert Deadlock-Freiheit!

Spracherweiterungen in Ada 95



- Inline-Maschinen-sprache
- Interrupts
- ....

- Flexibles Scheduling,
- flexible Prioritäten,
- flexible Warte-schlangen
- ....

- Partitionen
- Inter-Partitions-Kommunikation,
- statische und dynamische RPCs
- ....

- Dezimale Darstellung
- "PIC"-Darstellung
- ....

- Kontrolle über die interne numerische Darstellung,
- komplexe Zahlen
- ....

- Coderevision und -inspektion
- Einschränkung der verfügbaren Sprachkonstrukte
- ....

§ 6 Programmiersprachen für die Prozessautomatisierung

Tr6 In diesem Kapitel werden jeweils die Geschichten und die Sprachkonzepte der beiden Sprachen C und C++ besprochen. Anschließend kommt das wichtige Thema der Eignung dieser Sprachen für die Echtzeitprogrammierung und dahingehend eine Bewertung von C und C++ vorgenommen.
Traumüller; 13.11.2003

Entstehungsgeschichte von C und C++

1978	Entwicklung des Betriebssystems UNIX durch Dennis Richie in der Programmiersprache C
1986	Erweiterung von C für die objektorientierte Programmierung zur Programmiersprache C++

Entwicklungsziele von C und C++

C:
- Maschinennahe effiziente Systemprogrammiersprache
- flexibel wie Assembler
- Kontrollflussmöglichkeiten höherer Programmiersprachen
- universelle Verwendbarkeit
- begrenzter Sprachumfang

C++:
- Erweiterung von C um Objektorientierung
- Beibehaltung der Effizienz von C
- Verbesserung der Produktivität und Qualität

Hybride Programmiersprache

Concurrent C: Erweiterung von C um Konzepte zur Echtzeitverarbeitung

Sprachkonzepte von C

–	Vier Datentypen:	char, int, float, double
–	Zusammenfassung von Daten:	Vektoren, Strukturen
–	Kontrollstrukturen:	if, switch, while, do-while, for, continue, break, exit, goto
–	Ein-/Ausgabe:	Bibliotheksfunktionen
–	große Vielfalt an Bit-Manipulationsmöglichkeiten	
–	schwaches Typkonzept	
–	getrennte Kompilierbarkeit von Sourcefiles	
–	schwaches Exception-Handling	
–	keine expliziten Möglichkeiten für Parallelverarbeitung	

Programmaufbau

```
Einfügen bestimmter Bibliotheken bzw. Dateien
Festlegen der Namen für Konstanten und Makros

Deklaration von globalen Variablen;

main ( )
{
        Variablen, die innerhalb der Funktion main bekannt sind,
        müssen deklariert werden;
        Anweisungen;
}

Funktionstype Funktionsname (Liste der Parameter)
{
        Variablen, die innerhalb der Funktion bekannt sind, müssen hier
        deklariert werden;
        verschiedene Anweisungen;
        return (Wert);
}
```

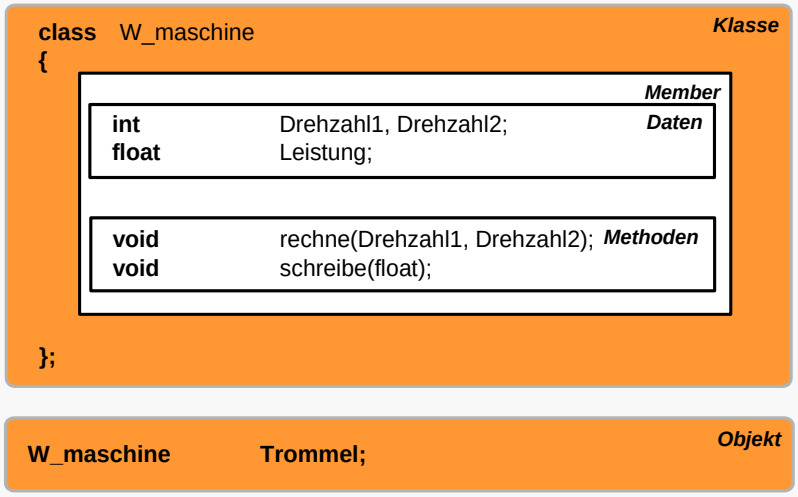

Beispiel

```
#include <stdio.h>                    // Standard-Ein-/Ausgabebibliothek

main ( )                              // Kennzeichnung des Programm-
{                                     // anfangs

        printf („Mein erstes Programm/n“)        // Ausgabeanweisung

}
```


Sprachkonzepte von C++

- Klasse (class)
  - Datenstruktur mit Daten und Methoden (memberfunction)
- Konstruktor (constructor)
  - Anlegen einer Instanz einer Klasse (Objekt)
- Destruktor (destructor)
  - Freigabe von Klassenobjekten
- Überladen von Funktionen (Overloading)
- Datenkapselung (encapsulation)
- Vererbung (inheritance)
- Polymorphismus
  - Auslösung unterschiedlicher Verarbeitungsschritte durch Botschaften

Struktur eines C++-Programms



Eignung von C und C++ für die Echtzeitprogrammierung (1)

- C und C++ enthalten keine Echtzeit-Sprachmittel
- Einsatz von Echtzeit-Betriebssystemen zur Realisierung von Echtzeitsystemen

Aufruf von Betriebssystemfunktionen im C-Programm

- Bereitstellung von Bibliotheken

Eignung von C und C++ für die Echtzeitprogrammierung (2)

Programmiersprachen C und C++

- ↑ Am häufigsten verwendete Programmiersprache für Echtzeit-Anwendungen
  - große Anzahl und Vielfalt an Unterstützungswerkzeugen
  - gut ausgebaute Programmierumgebungen
  - für die meisten Mikroprozessoren sind Compiler verfügbar
  - Anschluss an Echtzeitbetriebssysteme wie QNX, OS9, RTS, VxWorks

⇨ **Vorsicht bei objektorientierten Sprachmitteln**
- **nicht-deterministisches Laufzeitverhalten**
- **schlechte Speicherplatzausnutzung**

§ 6 Programmiersprachen für die Prozessautomatisierung

Folie 64

Tr7 Java ist im Gegensatz zu C, C++ oder Ada eine relativ junge Programmiersprache. In diesem Kapitel werden kurz die Entstehungsgeschichte und dann die Sprachkonzepte von Java erläutert. Danach untersucht die Vorlesung die Eignung von Java für die Echtzeitentwicklung und schließlich wird die neuere Entwicklung Rael-Time Java behandelt. Real-Time Java ist eine bewusste Erweiterung der Sprache Java mit bestimmten Spracheigenschaften, die zur Echtzeitprogrammierung benötigt werden.

Traumüller; 13.11.2003

Entstehungsgeschichte von Java

1990 Konzept der Programmiersprache Java durch die Fa. Sun (James Gosling, Bill Joy)

Ziel: Programmiersprache für die Unterhaltungselektronik (interaktives Fernsehen)

Namensgebung nach der Kaffeesorte Java

1995 Neuorientierung der Entwicklungsrichtung zu einer Sprache, um Programme im World Wide Web zu übertragen und auszuführen

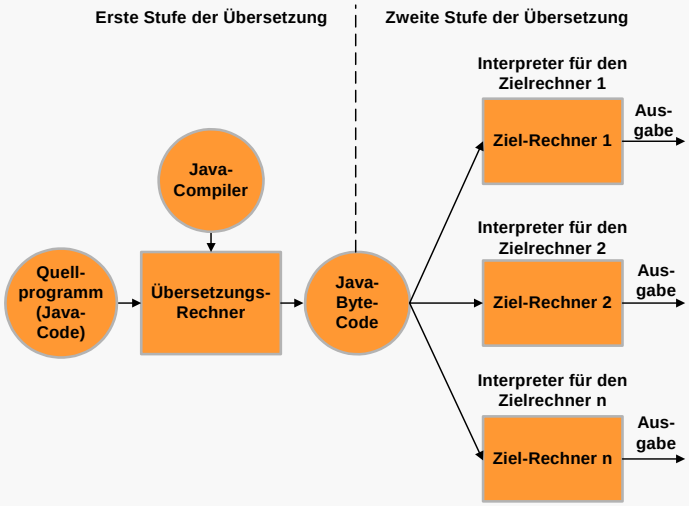
Frei verfügbar für nicht-kommerzielle Zwecke

Sprachkonzepte von Java

- Objektorientierte Konzepte
- Interpretierung des Codes
  - schneller Entwicklungszyklus
  - schlechtes Laufzeitverhalten und hoher Speicherplatzbedarf
  - höhere Portabilität
- Bereitstellung eines Speichermanagers
- Verzicht auf herkömmliche Zeigertechnik
- Strikte Typprüfung zur Kompilier- und Laufzeit
- Leichtgewichtsprozesse
- GUI-Klassenbibliothek

Portabilität

Die Portabilität von Java wird durch zweistufiges Übersetzungsverfahren erreicht.



Unterschiede zu C++

- Keine Preprozessoranweisungen wie #define oder #include
- keine typedef-Klauseln
- Structures und Unions in Form von Klassen
- keine Funktionen
- keine Mehrfachvererbung
- kein goto
- kein Überladen von Operatoren
- umfangreiche Klassenbibliothek
  - Basisklassen (Object, Float, Integer)
  - GUI-Klassen
  - Klassen für Ein-und Ausgabe
  - Klassen für Netzwerkunterstützung

Eignung von Java für die Entwicklung von Echtzeitsystemen

Anwendungsfelder

- rapid prototyping im Client/Server-Bereich
- multimediale Darstellungen (Video, Sound, Animation)
- Intranetapplikationen
- Echtzeitanwendungen

⇨ **Speicherverwaltung (garbage collection)**
⇨ **hoher Speicherbedarf**
⇨ **schlechtes Laufzeitverhalten**

Echtzeitsprachmittel in Java

- Eingabe und Ausgabe von Prozesswerten
  - vergleichbar mit C/ C++
- Parallelität
  - keine Prozessunterstützung
  - Leichtgewichtsprozesse
  - Zeitscheibenverfahren
- Synchronisierung
  - Monitore
  - Semaphorvariablen
- Interprozesskommunikation
  - nur für Leichtgewichtsprozesse über gemeinsame Daten
- Bitoperationen
  - vergleichbar mit C/ C++

Java-Beispiele

Java Applikation

```
class HelloWorldApplication
{
        public static void main(String argv[ ])
        {
                System.out.println(„Hello World!“);
        }
}
```

Java Applet

```
import java.applet.*;
import java.awt.Graphics;
public class HelloWorldApplet extends Applet
{
        public void paint(Graphics g)
        {
                g.drawString(„Hello World!“),5,25);
        }
}
```


Real-Time Java

Erweiterungen der Sprache Java, um Echtzeiteigenschaften Echtzeitanforderungen realisieren zu können.

Erweiterungen

- **1. Scheduling**
  Sicherstellung, dass Sequenzen von eingeplanten Objekten rechtzeitig und berechenbar ausgeführt werden
- **2. Speichermanagement**
  Erweiterung des Speichermodells um Echtzeitanwendungen deterministisches Verhalten zu ermöglichen
- **3. Synchronisation**
  Spezifikation der Zuteilungsalgorithmen mit Unterstützung der „priority inheritance“ und „priority ceiling“-Methoden und unter Vermeidung des Problems der Prioritäteninversion
- **4. Asynchrone Ereignis-Behandlung**
  Gewährleistung, dass das Programm mit einer großen Anzahl gleichzeitiger Ereignisse (bis zu mehreren 10000) umgehen kann.

Erweiterungen (2)

- **5. Asynchrone Ausführungsunterbrechung (ATC)**
  Möglichkeit der Ausführungsunterbrechung eines Threads bei Eintreffen eines asynchronen Ereignisses (z.B. Timerablauf)
- **6. Asynchrone Thread-Terminierung**
  Gewährleistung einer planmäßigen Terminierung und Speicherfreigabe von Threads ohne Deadlockgefahr
- **7. Physikalischer Speicherzugriff**
  Spezielle API (Application Programming Interface) für einen direkten Speicherzugriff
- **8. Exceptions**
  Definition von neuen Exceptions (Ausnahmen) und neuer Behandlung von Exceptions im Zusammenhang mit ATC und Speicherreservierung

Frage zu Kapitel 6.2

Für die Steuerung der Zentralverriegelung in einem Kfz soll ein Steuergerät mit einem Mikrocontroller eingesetzt werden.

Erläutern Sie, warum es vorteilhaft sein kann, die Software in einer Assemblersprache zu erstellen.

Antwort

Der erzeugte Code aus einem Assemblerprogramm ist effizienter und benötigt weniger Speicherplatz.

Dadurch können die Kosten für die Serienproduktion (eventuell) geringer ausfallen, da ein billigerer Mikrocontroller und weniger Speicher benötigt werden.

Bei großen Stückzahlen fällt dies wesentlich mehr ins Gewicht als die höheren Entwicklungskosten, die durch die Programmierung in einer Assemblersprache anfallen.

Frage zu Kapitel 6.4

Nennen Sie einige wesentliche Unterschiede zwischen der Echtzeit-programmiersprache Ada 95 und der SPS-Sprache FBS in Hinblick auf

Antwort

	Ada	FBS
Notation	textuelle Sprache	grafische Sprache
Sprachhöhe	Universelle höhere Sprache	Sprache vor allem für Steuerungen
Einsetzbarkeit	Geeignet für große Projekte	Geeignet für kleine Projekte
Echtzeiteigenschaften	Umfassende Echtzeit-Spracheigenschaften	Nur eingeschränkte Spracheigenschaften für die Echtzeitprogrammierung

Frage zu Kapitel 6.4

Ada 95 wird auf Grund einiger seiner Spracheigenschaften als Echtzeit-Progammiersprache angesehen. Warum ist Ada besonders Echtzeit-tauglich? Da.....

Antwort

- ☐ Ada keine Objekt-Orientierung besitzt.
- ☑ Ada Rechenprozesse unterstützt.
- ☐ Ada schneller wie andere Programmiersprachen ist.
- ☐ Ada interpretiert wird.
- ☐ Ada eine hybride Programmiersprache ist.
- ☑ Ada ein Rendezvous-Konzept bietet.
- ☑ Ada über Methoden für Laufzeitüberprüfungen und eine Ausnahmebehandlung verfügt.

Frage zu Kapitel 6.4

Folgender Ablauf soll in Ada in Form zweier Tasks implementiert werden:

Ein Passant wartet auf ein Taxi und fährt mit diesem zur Kirche. Danach bezahlt er die Fahrt und geht seines Weges.

Nachfolgend finden Sie 3 Codeauszüge für dieses Problem. Welcher ist der Richtige und was machen die anderen?

Frage zu Kapitel 6.4 - Codeauszüge

Variante 1

```
task passant;

task body passant is
begin
     -- do something
     taxi.drive;
     -- do something
end passant;


task taxi is
     entry drive;
end taxi;

task body taxi is
begin
     -- do something
     accept drive;
     drive.to(church);
     -- drive away
end taxi;
```

f - Realisiert nur zeitliche Synchronisierung ohne gemeinsame Codeausführung

Variante 2

```
task passant;

task body passant is
begin
     -- do something
     taxi.drive;
     -- do something
end passant;


task taxi is
      entry drive;
end taxi;

task body taxi is
begin
     -- do something
     accept drive do
      drive.to(church);
     end drive;
    -- drive away
end taxi;
```

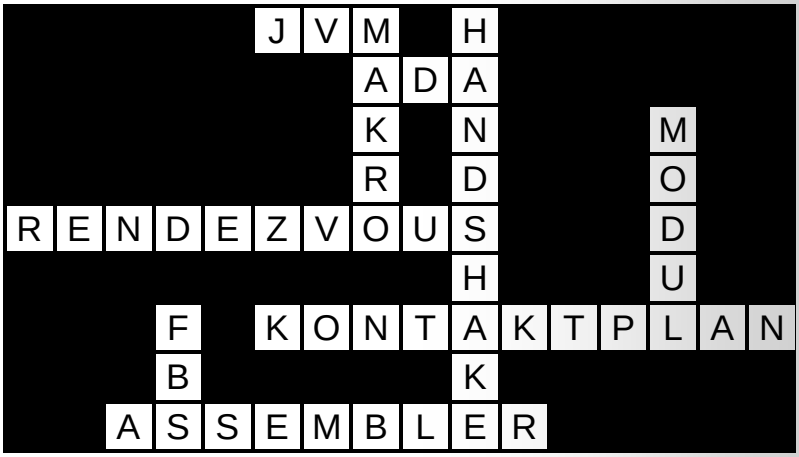
✓- Realisiert Synchronisierung mit gemeinsamer Codeausführung

Variante 3

```
task passant is
     entry taxi;
end passant;

task body passant is
begin
     -- do something
     accept taxi;
     -- do something
end passant;


task taxi is
     entry passant;
end taxi;

task body taxi is
begin
     -- do something
     accept passant;
     drive.to(church);
     -- drive away
end taxi;
```

f - Realisiert keine gegenseitige Synchronisierung

Kreuzworträtsel zu Kapitel 6



Kreuzworträtsel zu Kapitel 6

Waagerecht

1	Grundlage für die Plattformunabhängigkeit von Java (3)
4	Echtzeit-Programmiersprache (3)
6	Ada-Konzept zur Synchronisierung (10)
8	Schaltplan-ähnliche Darstellung für SPS-Programme (11)
9	Maschinennahe Programmiersprache (9)

Senkrecht

2	Hilfsmittel zur Vereinfachung der Maschinensprache (5)
3	Engl. Bezeichnung für eine gemeinsame Ausführung eines Programmstücks auf Empfängerseite (9)
5	Paket in Ada (5)
7	Abkürzung für eine Blockschaltbild-orientierte SPS-Programmiersprache (3)