§ 6 Programming Languages for Process Automation

Chapter 6 - Learning targets

- to know the modus operandi when doing process automation programming
- to be able to differ programming languages referring to the language level
- to know the programming languages for PLC
- to program simple examples using PLC-languages
- to know the most important real-time concepts for programming languages
- to know how real-time concepts are realized in Ada 95
- to be able to design a real-time program in Ada 95
- to be able to rank C/C++ referring to real-time aspects
- to know what the fundamentals of the portability of Java
- to understand the real-time extensions of Java

§ 6 Programming Languages for Process Automation

Procedures for program generation

Programmable Logic Controllers
- Textual programming languages
- Graphical programming languages

Micro controller
- Assembler
- Low machine-independent programming languages

PC and IPC
- Software package
- Universal real-time programming language

Process control system
- Functional module technology

Programming language types

Classification according to notation type

–	textual programming languages	Ada, C, PLC instruction list
–	graphical programming languages	PLC ladder diagram

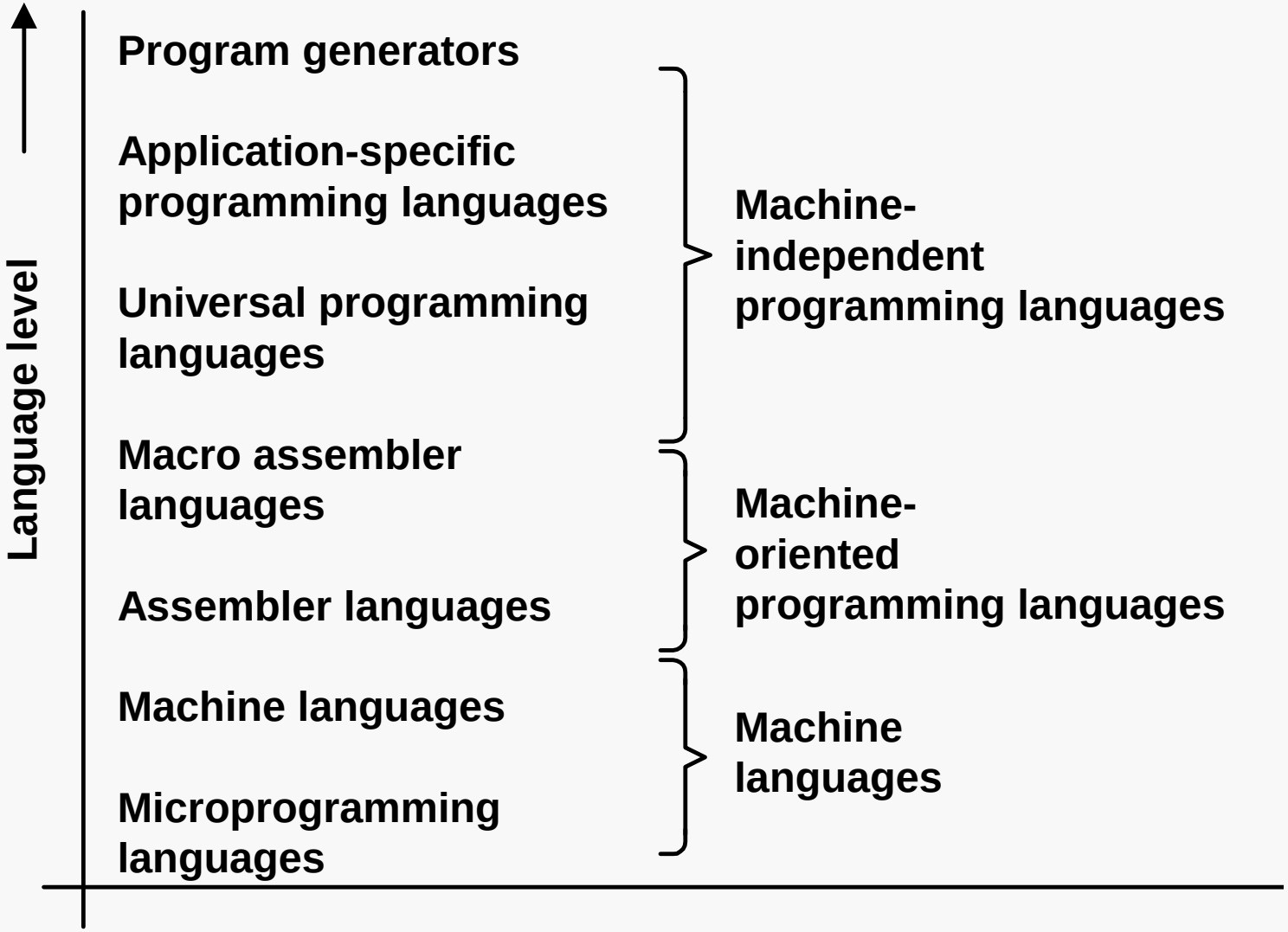
Classification according to programming language paradigm

–	procedural programming languages	C, Ada 83
–	functional programming languages	LISP
–	logical programming languages	PROLOG
–	object-oriented programming languages	C++, Smalltalk, Ada 95

Classification according to language level

–	**high:**	focus on the understandability for the user
–	**low:**	focus on the hardware characteristics of a computer

Classification according to language level



Microprogramming languages

- Sequence control realization for the execution of machine commands
  - Permanently wired logic elements
  - Microprograms
- Microprograms (firmware)
  - Storage in fast random access memory (RAM's) or in read only memory (ROM's)
  - Not accessible for mask programming
- Machine languages
  - Language elements:

    Commands and data in form of bit patterns

  - Combination of octal or hexadecimal numbers
  - Difficult handling
  - Not suitable for application programming

Assembler languages

Goals: Avoid the difficult handling of machine languages while keeping the characteristics of machine commands

- Replacement of the octal/hexadecimal notation on the operation part of the commands by symbolic, mnemonically convenient letter abbreviations
- Introduction of a symbolic name instead of the numerical depiction of the address part
- Non-ambiguous assignment of commands in the assembler languages to commands in the machine languages
- Dependency on device-technical characteristics of the corresponding computer

Macro assembler languages (macro languages)

– Further auxiliary means for simpler handling: macros

Macro: Abbreviation for a certain instruction sequence

– Distinction
  - Macro definition
  - Macro call
  - Macro expansion

– Setup of a macro definition

MACRO Macro name (P_1, P_2, ..., P_N)
Macro body
Final mark

Macro call
Macro name (A_1, A_2, ..., A_N)

– Non-ambiguous assignment of macro assembler commands to commands of the machine language

– One macro command corresponds to several machine commands

Example: macro threeaddressadd

Macro definition:

```
MACRO Threeaddressadd (P1, P2, P3)
LOAD P1
ADD P2
STORE P3
END
```

Macro call:

```
Threeaddressadd (A,B, SUM)
```

Macro expansion:

```
...
LOAD A
ADD B
STORE SUM
...
```

Difference between subprogram calls and macros

- A subprogram is only stored once but can be called and executed several times
- Each time a macro is called it is expanded

Classification of macros

- Standard macros: predefined (fix)
- User macros: the user is able to define macros for instruction sequences that are needed often

Universal programming languages

Universal $\triangleq$ **not focused on one specific application area**

Universal low programming languages **System programming languages**

- Purpose: Generation of system programs
  - Compiler
  - Operating systems
  - Editors
  - Driver programs
- Objective: 1. Utilization of the hardware characteristics
  2. Portability
- Example: C

Universal higher programming languages

- Purpose : Generation of general/common programs
- Objective : 1. Simple formulation
  2. Extensive compiler checks
  3. Portability
- Example : Ada, Java, Smalltalk

Application-specific languages

Descriptive languages, non-procedural higher languages, very high level languages

Difference to procedural languages:

– No description of the solution approach,
instead: description of the problem definition

– Limited to certain application areas:

e.g.:	SFC	Sequential Function Chart
	LD	Ladder Diagram
	IL	Instruction List
	EXAPT	For machine tool control
	ATLAS	For automatic device tester

Advantages/disadvantages:

+ comfortable, application-specific mode of expression

- inflexibility

Program generator (fill-in-the-blanks-languages)

- Formulation method for programs
- User replies to questions in the form of menus on screen (user configuration)
- Conversion of the answers in an executable program through the program generator
- Advantage: no programming skills necessary
- Disadvantage:
  - **Limitation to certain application areas**
  - **Dependency in regard to a certain manufacturer**

Application areas of program generators in the process automation field

- Instrumentation and control systems in energy and process engineering
  - TELEPERM-M (Siemens)
  - PROCONTROL-B (ABB)
  - CONTRONIC-P (Hartmann & Braun)
- Programmable logic controller in the form of instruction lists or ladder diagrams
  - SIMATIC (Siemens)

§ 6 Programming Languages for Process Automation

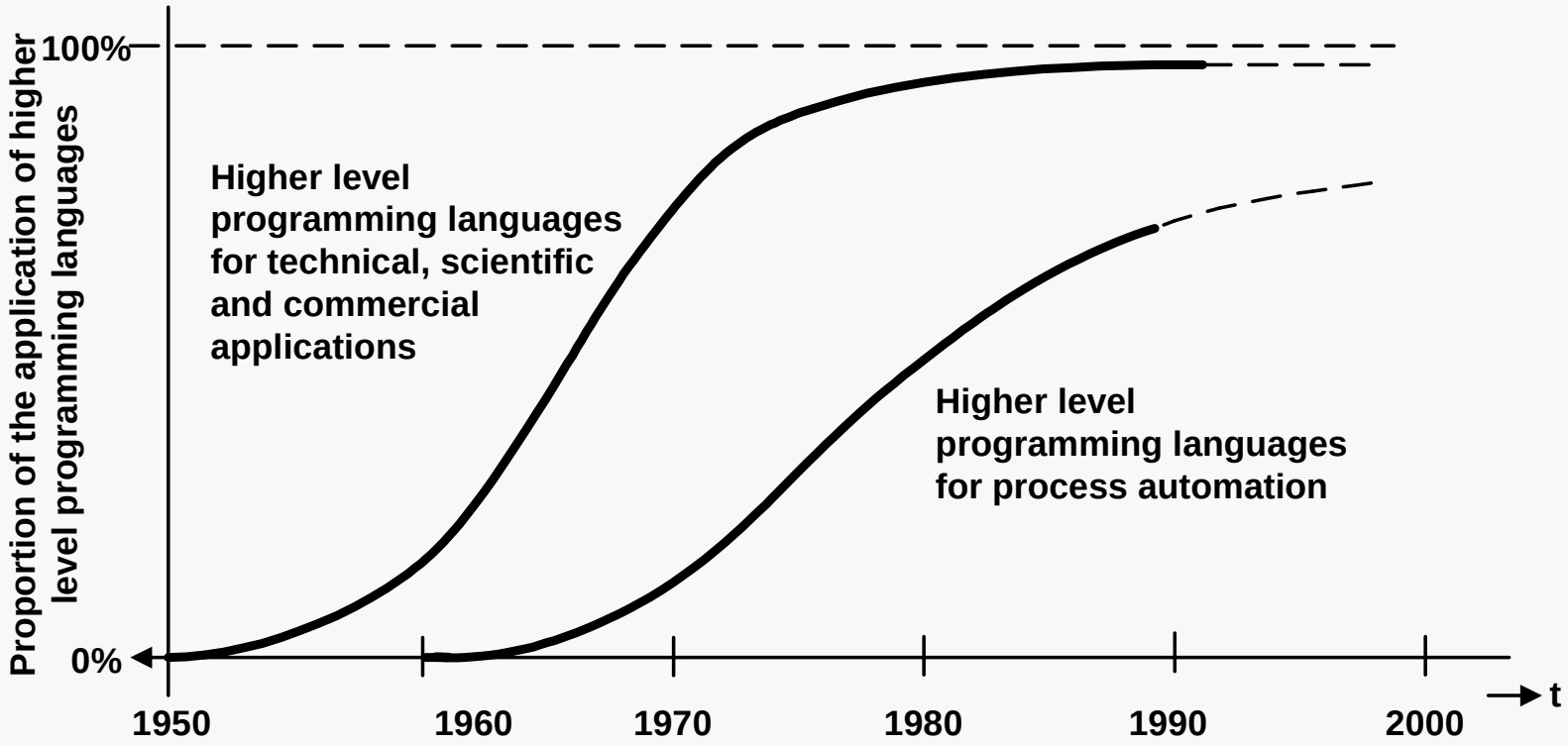
Difficulties regarding real-time programming

– High-level programming languages like BASIC, FORTRAN, COBOL and PASCAL are not suitable for real-time programming:
  - no real-time language resources
  - no single bit operations
  - no instructions for process input/output

– The operation of higher programming languages in comparison to assembler languages also causes higher demands on memory and computing times:
  - Product automation: memory usage and computing times critical
  - Plant automation: computing times might be critical

Availability
- Compiler for target computer
- Real-time operating system

Deployment comparison of higher programming languages

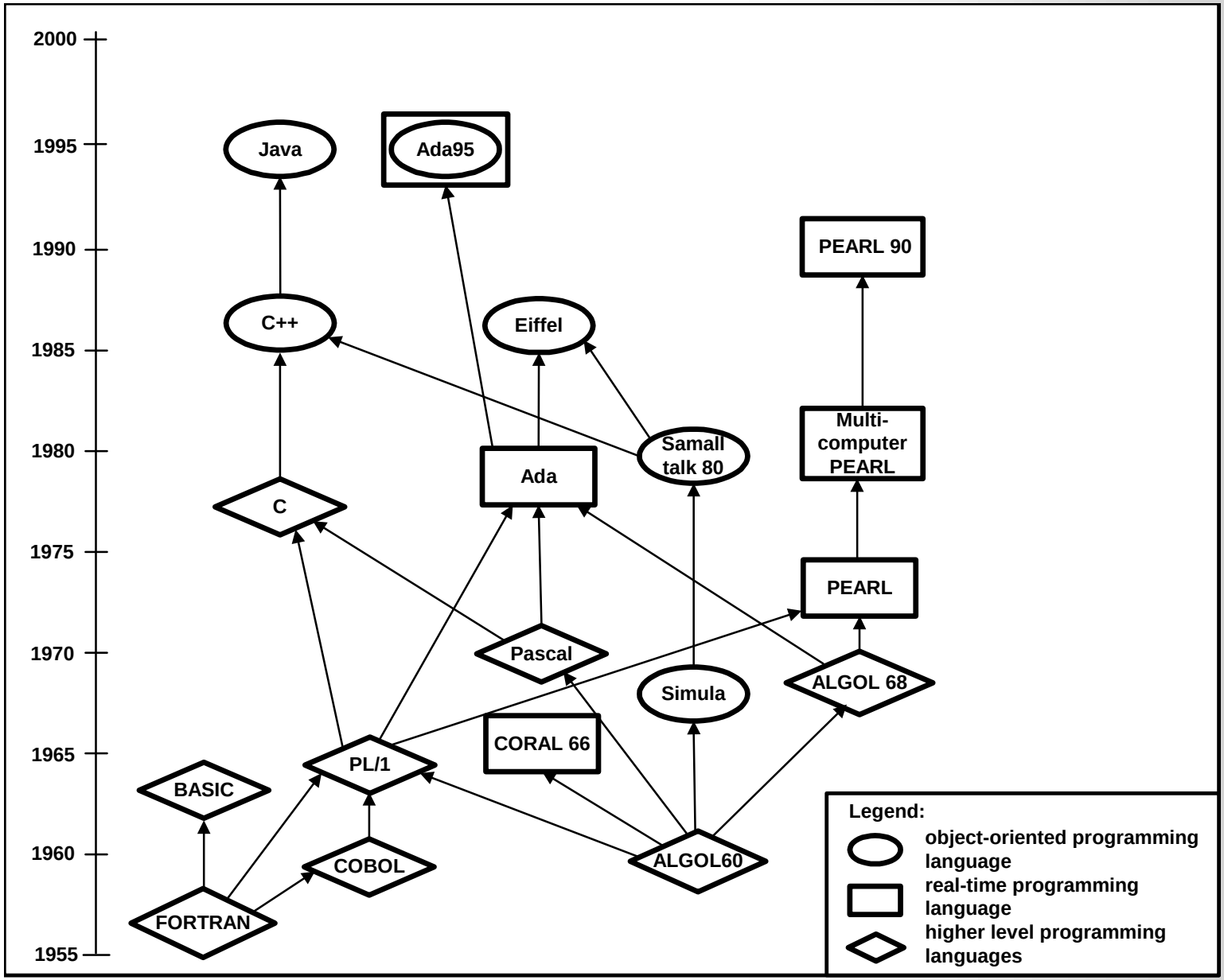


Advantages and disadvantages of assembler programming

+	memory and computing time efficiency
-	higher program development costs
-	lower maintainability
-	problems concerning reliability
-	bad readability, low documentation value
-	missing portability

Operation of assembler languages

yes:	automation of devices or machines with serial or mass application, small programs
no:	long-lasting, big processes that ought to be automated


2000
1995
1990
1985
1980
1975
1970
1965
1960
1955
Java
Ada95
PEARL 90
C++
Eiffel
Multi-
computer
PEARL
Samall
talk 80
Ada
C
PEARL
Pascal
ALGOL 68
Simula
CORAL 66
PL/1
BASIC
COBOL
ALGOL60
FORTRAN
Legend:
object-oriented programming language
real-time programming language
higher level programming languages

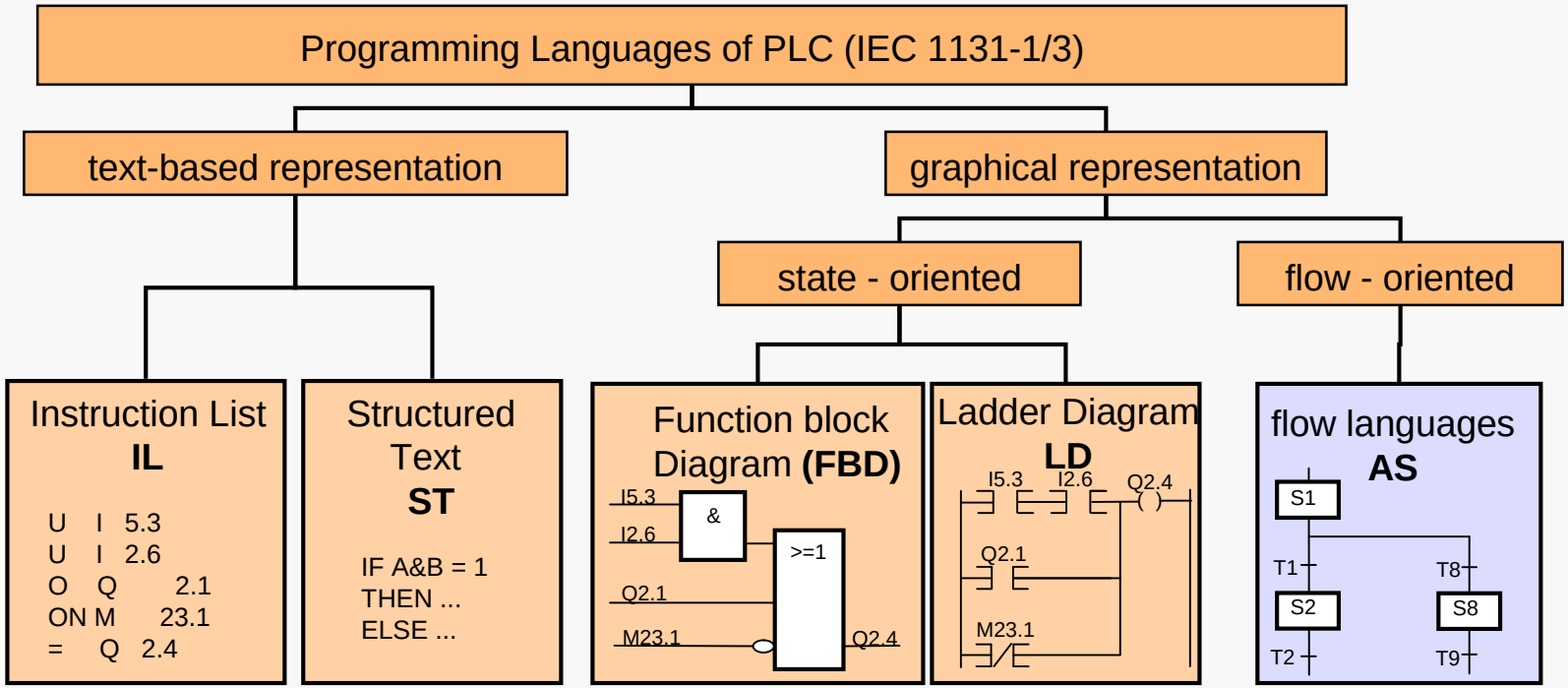
§ 6 Programming Languages for Process Automation

Programming Languages for PLC-Systems (1)

– no fixed programming languages for PLC-systems

– different kinds of programming, also depending on the manufacturer

– IEC 1131 is defining graphical and text-based basic-languages



– IEC 1131 is not defining any instructions!

ð as well german as english identifier for operators

Programming Languages for PLC-Systems (2)

– base of all programming languages are logical connections

– addition of possibilities for time processing

– suitable kind of representation depends on problem

  - state-oriented program parts are more suitable for FBD or LD
  - flow-oriented program parts are more suitable for IL or AS

– the different representations can be converged into each other

Be aware:

Some operations can only be programmed using IL (e.g. bit shifting)!

Example:
Exit 1 (exit1) and exit 2 (exit2) are only true, if either entry 3 (entry3) is true or if both entries 1 (entry1) and 2 (entry2) are simultaneously true

Instruction List (IL)

– similar to assembler

– all functionality of a PLC can be programmed using IL

– consistent structure of every statement

```
Mark(opt.):    Operator  Operand      comment(opt.)
```

– the programming is done by connecting signals

Realization of the example using AWL (chalkboard writing)

Structured Text (ST)

– high-level language, similar to Pascal

- declarations

  e.g.: alarm := on AND off;

- sub program calls

  e.g.: alarmlamp(S:=on, R:=off);

- control statements

  e.g.:
  ```
  IF alarm
      THEN alarmlamp(S:=on, R:=off);
  END_IF;
  ```

– additional language constructs for time processing and process data access

– suitable for programming large systems

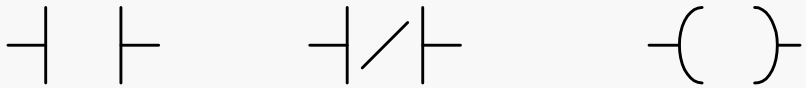
Realization of the example using ST (chalkboard writing)

Ladder Diagram (LD)

– simple illustration

– similar to the circuit diagram of the relay technology

– symbols for „normally open contact“, „normally closed contact“, und „relay coil“ (output)



normally open contact | normally closed contact | relay coil

– symbolized current flow from left to right

– I/O states are mapped onto switch-states

– the program is read from top to bottom

Realization of the example using LD (chalkboard writing)

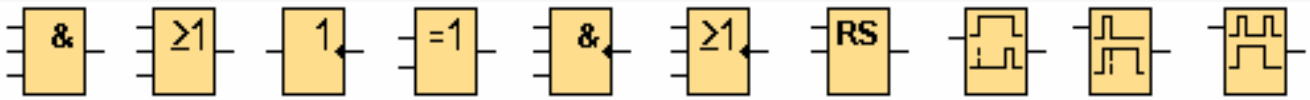
Disadvantage: difficult representation of complex mathematic functions

Function Block Diagram (FBD)

– similar to the well known symbols for function blocks (DIN40900)

– amount of symbols is not confined to logical basic elements

ð memory marks, counter, timer and freely definable blocks possible



– I/O states are directly used as Input-/Outputsignals

– clearly arranged representation

Realization of the example using FBD (chalkboard writing)

§ 6 Programming Languages for Process Automation

Lady Ada Lovelace

Ada

Name of the 1. female programmer

Named in the honor of the mathematician
Augusta Ada Byron, Countess of Lovelace, daughter of Lord Byron. Ada Lovelace (1815- 1851) worked with Charles Babbage on his “Difference-and Analytic-Engine”. The idea of programming this machine with punched cards can be traced back to Ada Lovelace.

Characteristics of Ada

- Suitable for the development of extensive software systems (> 10^7 lines)
  - Modularization concepts
  - Separation between specification and implementation of units
  - Support to reuse of software components
  - Exception handling as part of the module's code
- Real time specific features
  - Fulfillment of absolute and relative time requirements
  - Execution of parallel activities
  - Prioritization and scheduling of parallel activities
- Extensive validation of each Ada95 compiler against compliance standards
  - Several thousand test programs

Ada 95 program units

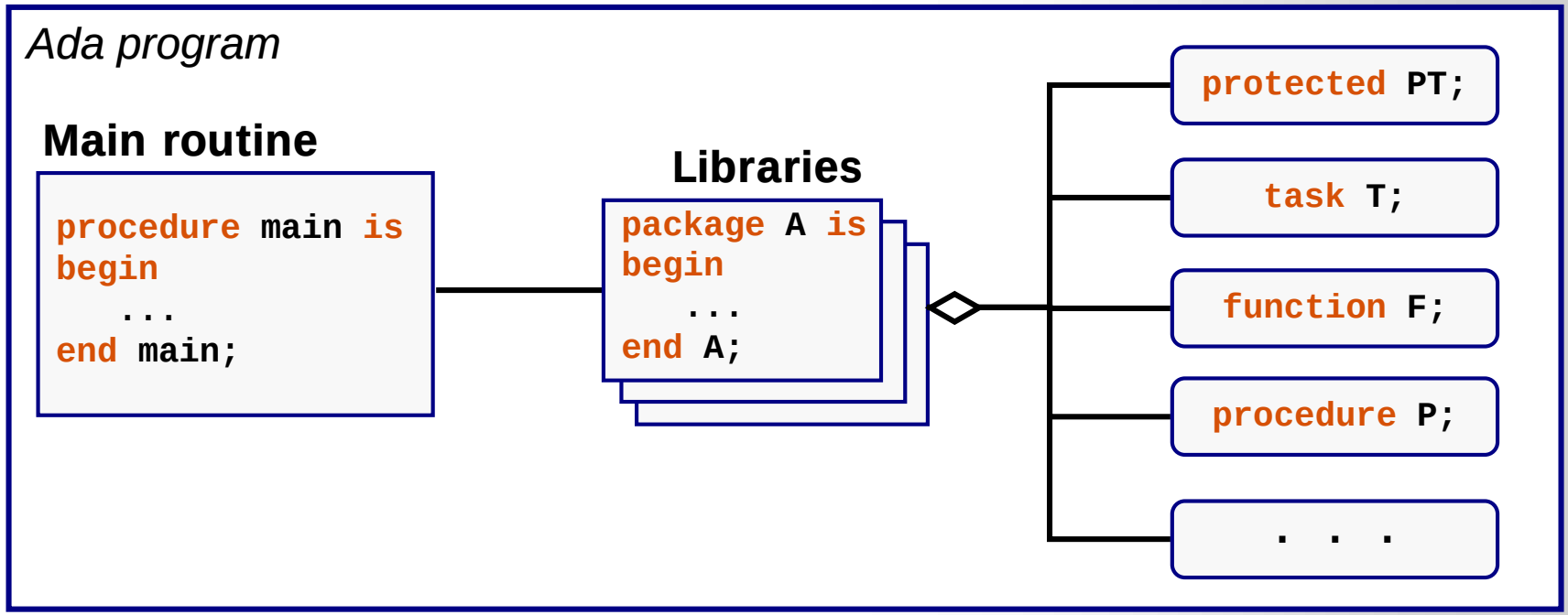
– Subprograms
  - Specific functionality encapsulated in executable pieces of code
    è Functions and procedures

– Concurrently executing programs
  - Different program sequences that are executed at the same time
    è Tasks

– Shared resources with synchronized access
  - Data structures with access protected by monitors
    è Protected variables

– Reusable libraries
  - Group of functionality related subprograms, tasks and protected units
    è Packages

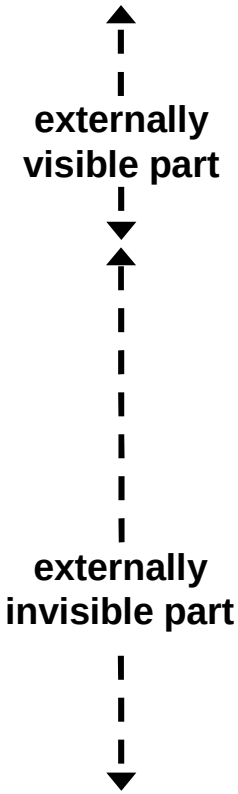
Ada 95 basic program structure



- No explicit superordinate program
  - Definition of a subprogram as main routine
- Execution of program units from the main routine
- Programs units stored in libraries or directly declared

Example of unit structure

- Specification **= external interface**
- Implementation **= realization of the program unit**

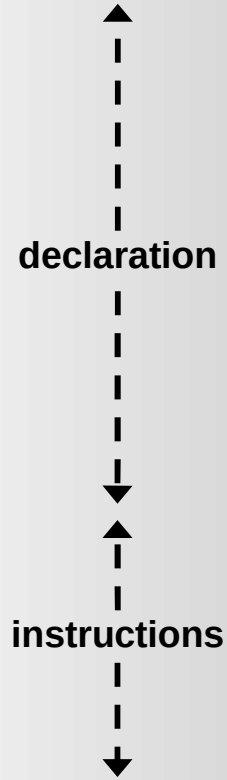


Specification (package declaration)

```
package package_name is
  declaration
  (types, subprograms,
  tasks, protected units, ...)
  private
   private part (declaration)
end package_name;
```

Implementation (package body)

```
package body package_name is
  further declaration
  subprograms' bodies
 begin
   instructions
end package_name;
```



Declaration and implementation of units

– Separation of specification and implementation in different files

Specification (File *General.ads*)

```
package General is

 protected Variable is

  function read return Float;

  private
   protectedData: Float:= 0.0;

 end Variable;

...

end General;
```

Implementation (File *General.adb*)

```
package body General is
protected body Variable is
   function read return Float is
   begin
     return protectedData;
   end read;
 end Variable;


...

end General;
```

Concurrent programming concepts

- Requirement to enable the reaction to simultaneous events
  - è Concept of parallel tasks to perform parallel activities
- Support of concurrent programming

Task unit

- Task as a concurrent, parallel running sequences of commands
- Task as a encapsulated unit
  - è Declarations made within a task are not visible outside it
- Communication between task units
  - è Message passing and via shared variables

Realization of tasks

- Declaration and implementation of tasks in a library or directly in the program unit
- Planning of tasks done at the **begin** of the superordinate program unit
- End of execution and deletion after reaching the task's **end** statement

```
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;
```

```
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 types and objects

- Task types provide a template for the instantiation of task objects
- Tasks of the same type have similar properties and functionality
- Individual task objects can be parameterized during instantiation

```
package General is

  task type monitorValue(X: Integer);

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

end General;
```

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

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

begin
  -- Main code
  ...
end main;
```

Synchronization of tasks

Synchronization concepts in Ada 95

– Logical synchronization
  - Establish a timely sequence of the tasks' execution

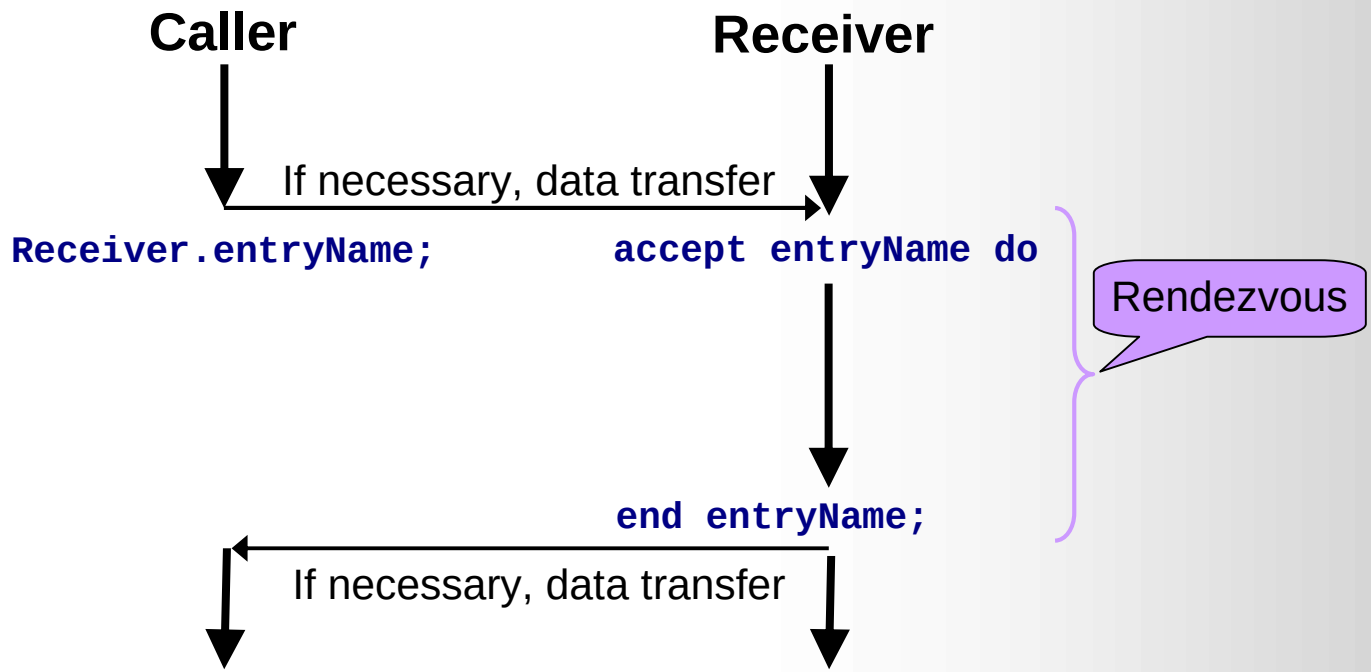
è Rendezvous concept

– Resource oriented synchronization
  - Establish rules to access shared resources

è Protected units

The rendezvous concept (1)

- Handshake method
  - Definition of timely synchronization points between the tasks
  - Mutual wait of the tasks
- Rendezvous has on the calling side the form of a procedure call
- Definition of calling declaration (entry) and calling point (accept)



The rendezvous concept (2)

- Declaration of entry points declared in the task's specification
- Calls and accept statements in the tasks' implementation code
- Function principle: task calls the entry point another task

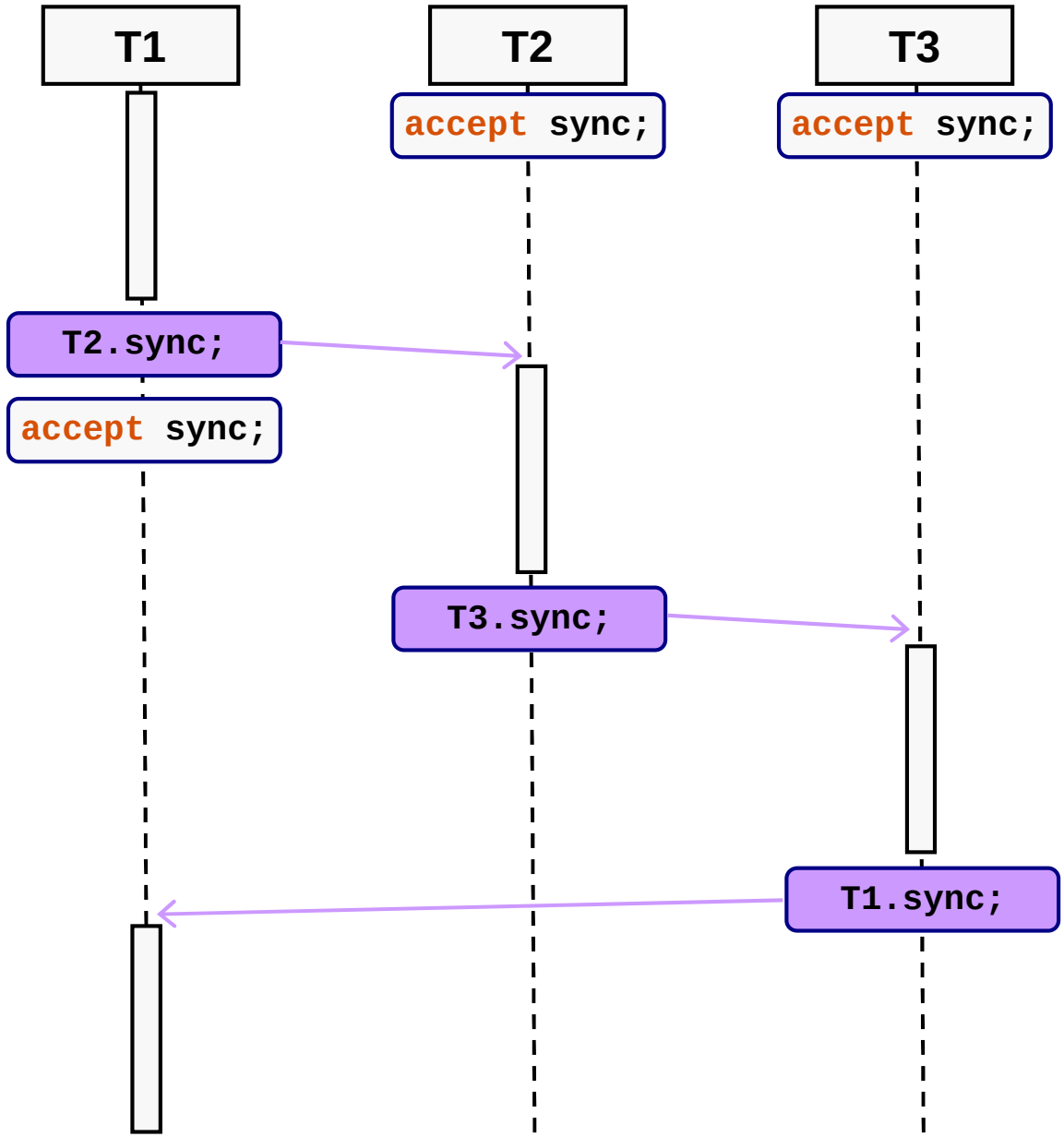
```
task T1 is
 entry sync;
end T1;

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;
```

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

task body T3 is
 begin

    accept sync;
    ...

    T1.sync;

 end T3;
```

è Execution sequence T1, T2, T3, T1, … is guaranteed in the code above

Dynamic run of the multitask program example



The select statement

– Alternative sequence with the select statement
  - Conditional synchronization
  - Timed waiting

Conditional synchronization

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

```
select
  Receiver.entryName;

 else
  -- Alternative code
end select;
```

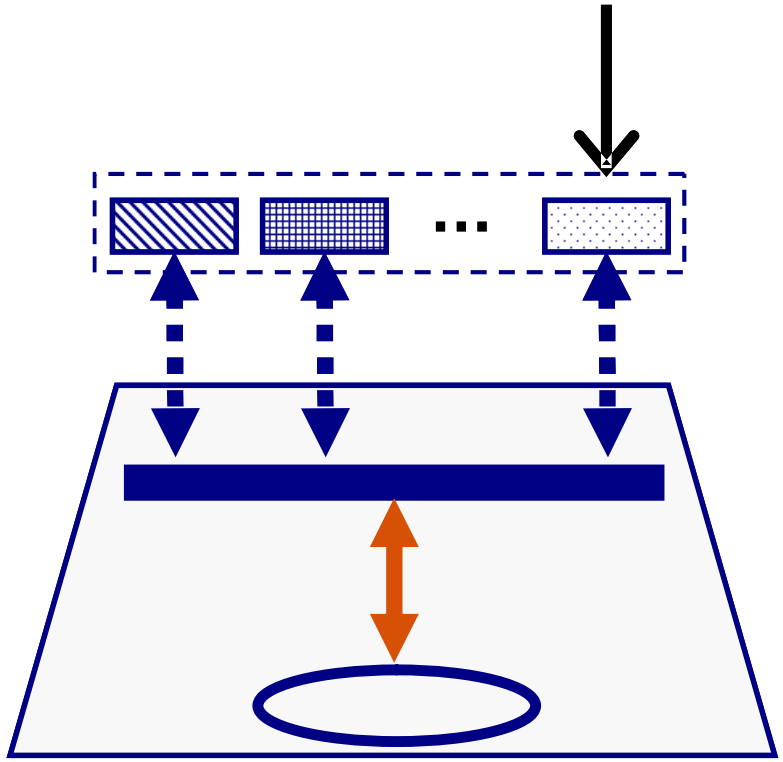
Timed waiting

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

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

Protected units and resource synchronization (1)

- Problem of accessing shared variables
- Monitors for exclusive access



- Interface as **protected operations**
- Access to the shared variable controlled by a intern **monitor**
  - Protected operations are not executed in parallel
  - Sequential ordering of simultaneous requests

è Tasks access the shared variable exclusively

Protected units and resource synchronization (2)

```
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;
```

- Protected operations are mutually exclusive
- A **procedure** can manipulated the shared variable in any way

  è Functions have only read access

- The **entries** are protected operations
- All variables of a protected unit have to be declared **private**

Communication between tasks (1)

- Synchronous communication
  - Extension of the rendezvous synchronization with data exchange
    - Mutual wait
    - Data exchange during the rendezvous
    - Data only valid within the rendezvous

Example of synchronous communication

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

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

è Message passing principle

Communication between tasks (2)

- Asynchronous communication
  - Data exchange via shared variables
    - ú Protected operation for access
    - ú No waiting/ blockage during communication

Example of asynchronous communication

```
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;
```

è Shared variables and protected operations

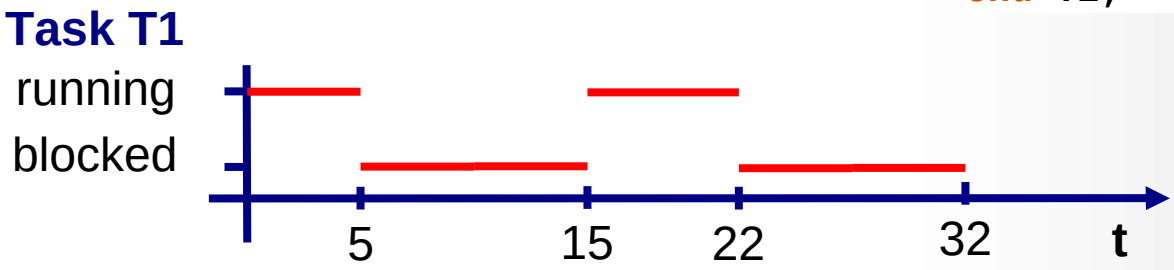
Time operations

- Possible delay of the execution
  - Until a defined point in time
  - For a fixed time period
- Ada 95 statements **delay** and **delay until**
- Time units and operations defined in the core packages
  - For standard applications, **second**
  - For real time applications **ms**, **µs**, **ns**
  - Operations
    - Add and subtract time variables
    - Read the current system time
- During a delay, the processor is occupied by another process ready to run
- After a delay, a task may have to wait to receive the processor
  - Preemption by a **higher priority task**

Example of timed operations

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

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

```
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;
```



- Task T1 activates read_sensor **after a 10ms time delay**
  - ð The subprogram activation time is relative to the previous execution
- Task T2 activates the read_sensor **every 10ms**
  - ð The subprogram is activated at fixed times

Modularized exception handling

- Exceptions due to irregular operations possible
  - è Treatment of exceptions (exception handling)

Ada 95 exception handling concept:

- Exception handling blocks are implemented as part of the unit's code
- If an exception occurs, control is given to the exception handling blocks
- Several exceptions can be handled in a single handling block
- Any exception not handled is propagated to the calling unit, if there is any
- Exceptions are raised by the runtime environment or by the program code
  - è Runtime environment: predefined exceptions
  - è Program code: custom defined exceptions

Exception handling example

```
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;
```

- Handling blocks implement the exception handling
  - Sensor exceptions handled within the function **readSensor**; others are propagated
  - Activation of specific code by constraint exceptions
  - **Predictable** system behavior also under exception conditions

Real time extensions to Ada 95 (1)

– Tasks with dynamic priorities
  - **Base priority** is established on its specification
  - **Active priority** can differ from the above

    è Priority inheritance

– **Hint:** In Ada, a lower number means a lower priority!

Specification

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

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

Implementation

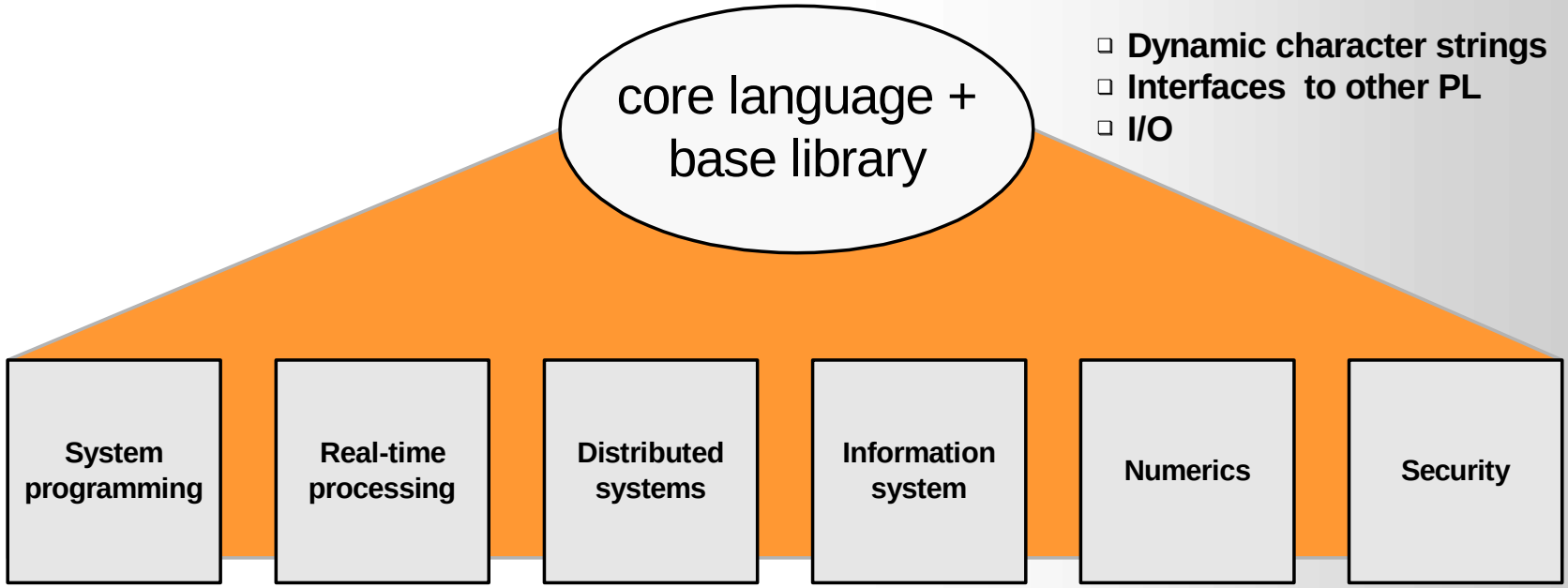
```
task body T1 is
begin
 T2.sync;
end T1;

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

 Set_Priority(20);
end T2;
```

Real time extensions to Ada 95 (2)

– Scheduling methods
  - The method used to schedule tasks can be specified
    - The **pragma Task_Dispatching_Policy**
    - Default method: **fixed priorities; FIFO within the priorities**
  - Further scheduling methods can be implemented and used

– Access to protected units and the Priority Ceiling Protocol
  - Ceiling priority of protected units can be defined in its specification
    - Default value: **system's highest priority**
  - Ceiling priority rules guarantee that there is **no chance of deadlock**

Further language extensions in Ada 95



- Dynamic character strings
- Interfaces to other PL
- I/O

- Inline-machine-language
- Interrupts
- ....

- Flexible scheduling
- Flexible priorities
- Flexible queues
- ....

- Partitions
- Inter-partition communication
- Static and dynamic RPCs
- ....

- Decimal format
- "PIC" format
- ....

- Control over internal numeric formats
- Complex numbers
- ....

- Code revision and code inspection
- Reduction of available language constructs
- ...

§ 6 Programming Languages for Process Automation

History of C and C++

1978 Development of the operating system UNIX by Dennis Richie in the programming language C.

1986 Extension of C for the object-oriented programming to the programming language C++.

Development objectives for C and C++

C:
- efficient system programming language for hardware
- flexible like assembler
- control flow possibilities of higher programming languages
- universal usability
- restricted language size

C++:
- C is enhanced with object orientation
- the efficiency of C is kept
- improvement of productivity and quality

Hybrid programming language

Concurrent C: C is enhanced with concepts dealing with real-time processing.

Language concepts of C

- Four data types: char, int, float, double
- Combination of data : vectors, structures
- Control structures: if, switch, while, do-while, for, continue, break, exit, goto
- Input/Output: Library functions
- Large variety of bit manipulation possibilities
- Weak type concept
- Separate compiling of source files
- Weak exception handling
- No explicit tools for parallel processing

Program setup

Insertion of certain libraries or files;
Definition of names for constants and macros;

Declaration of global variables;

```
main ( )
{
        Variables that are known within the function “main” have to be
        declared;
        Instructions;
}
```

Function type function name (list of parameters)

```
{
        Variables that are known within the function have to be declared
        here;
        Different instructions;
        Return (value);
}
```

Example

```
#include <stdio.h>                    // Standard input/output library

main ( )                              // Marker for program start

{

        printf („My first program“)   // Output instruction

}
```

Language concepts of C++

- Class
  - Data structure with data and methods (member function)
- Constructor
  - Creation of an instance of a class (object)
- Destructor
  - Release of class objects
- Overloading of functions
- Encapsulation of data
- Inheritance
- Polymorphism
  - Triggering of different processing steps by messages

Structure of a C++ program

Class

```
class   Wash_machine
{
```

Member Data

```
int         rotspeed1, rotspeed2;
float       capacity;
```

Methods

```
void        calculate (rotspeed1,rotspeed2);
void        type (float);
};
```

Object

```
Wash_machine        Drum;
```

Suitability of C and C++ for real-time systems (1)

- C and C++ contain no real-time language constructs.
- Deployment of real-time operating system to realize real-time systems.

Call of operation system functions in a C coded program

- Libraries are provided.

Suitability of C and C++ for real-time systems (2)

Programming languages C and C++

é Most often used program languages for real-time applications
- Great number and variety on support tools
- Well extended programming environment
- Compilers are available for most micro processors
- Connection to real-time operating systems like QNX, OS9, RTS, VxWorks

ð Caution with object-oriented language tools
- **non-deterministic run-time behavior**
- **inefficient memory space usage**

§ 6 Programming Languages for Process Automation

History of Java

1990 Concept of the programming language Java by the Sun Corp. (James Gosling, Bill Joy)

Objective: Programming language for entertainment electronics (interactive TV)

Named after coffee beans known as Java

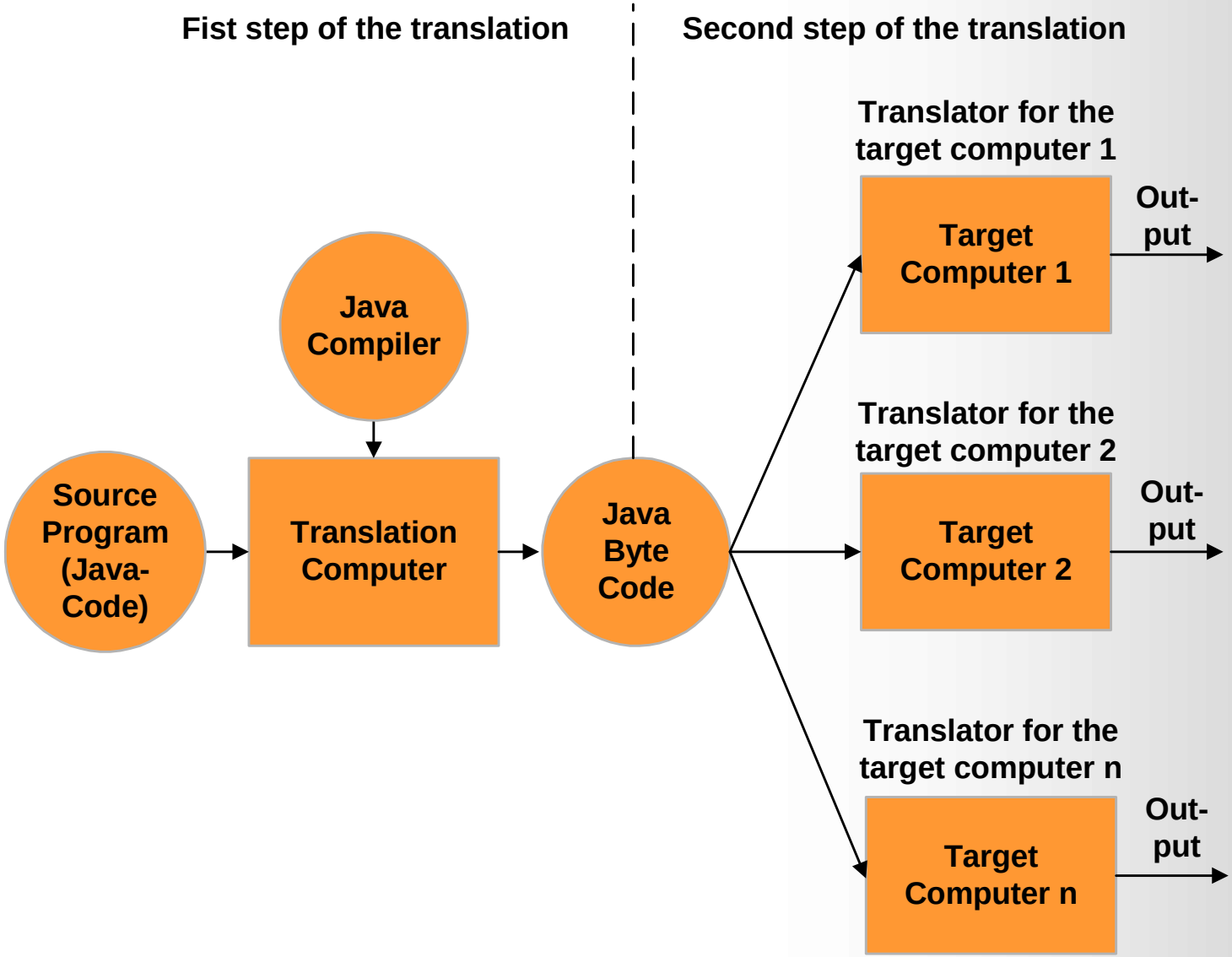
1995 Reorientation of the development direction towards a language that could be used for transmitting and carrying out programs in the World Wide Web.

Freely available for non-commercial purposes

Language concepts of Java

- Object-oriented concepts
- Interpretation of the code
  - fast development cycle
  - bad run-time behavior and high demand on storage space
  - higher portability
- A storage manager is provided.
- Conventional pointer methods were not integrated.
- Strict type control at compile and run-time
- Lightweight processes
- GUI class library

Portability of Java through a two-stage translation method



Differences to C++

- No pre-processor instructions like #define or #include
- No typedef clauses
- Structures and unions in the form of classes
- No functions
- No multiple inheritance
- No goto
- No overloading of operators
- Extensive class libraries
  - Base classes (object, float, integer)
  - GUI classes
  - Classes for input/output
  - Classes for network support

Suitability of Java for the development of real-time systems

Application fields

– rapid prototyping in client/server area

– multimedia presentations
(video, sound, animation)

– intranet applications

– real-time applications

- **Storage management (garbage collection)**
- **High demand on storage space**
- **Bad run-time behavior**

Real-time language constructs in Java

- Input and output of process values
  - comparable with C/ C++
- Parallelism
  - no process support
  - lightweight processes
  - Round Robin Method
- Synchronization
  - monitors
  - semaphore variables
- Inter-process communication
  - only for lightweight processes on common data
- Bit operations
  - comparable with C/ C++

Examples for Java

Java application

```
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

Extension in the Java language, to realize real-time requirements

Extensions (1)

- **1. Scheduling:**
  Ensure the timely or predictable execution of sequences of *schedulable objects*
- **2. Memory management:**
  Extend the memory model in order to allow real-time code to deliver deterministic behaviour
- **3. Synchronisation:**
  Specification of the dispatching algorithms; avoidance to the priority inversion problem; support to priority inheritance and priority ceiling policies.
- **4. Asynchronous Event Handling:**
  Ensure that the program can cope with a large number (ten of thousands) of simultaneous events

Extensions (2)

- **5. Asynchronous Transfer of Control (ATC):**
  Possibility to transfer the control from a thread upon an asynchronous event, e.g. a timer going off

- **6. Asynchronous Thread Termination :**
  Ensure an orderly clean up and termination of threads without danger of deadlocks

- **7. Physical Memory Access :**
  Special API to directly access memory areas

- **8. Exceptions:**
  Definition of new exceptions and new treatment of exceptions surrounding ATC and memory allocators

Question referring to Chapter 6.2

The control of a central locking system in a car is to be realized using a micro controller based control device.

Give reasons why it might be advantageous to implement the software in an assembler language.

Answer

Code generated from an assembler program is very efficient and requires less memory space.

Thus, the cost for the series production can be lowered, as a cheaper micro controller and less memory is required.

If the product is produced in high quantities this is more important than the higher development cost caused by the assembler programming.

Question referring to Chapter 6.4

State some of the differences between the real-time programming language Ada95 and the PLC language FBD concerning:

Answer

	Ada	FBD
Notation	text-based language	graphical language
Language level	universal high level language	language specific for control application
Application areas	suitable for large projects	suitable for small projects only
Real-time features	extensive real-time features	only restricted real-time capabilities

Question referring to Chapter 6.4

Due to its language features, Ada 95 is called a real-time programming language. Why is Ada so suitable for real-time issues? Because...

Answer

- f ¨ Ada has no object-orientation.
- ü Ada supports Tasks.
- f ¨ Ada is faster than other programming languages.
- f ¨ Ada is been interpreted.
- f ¨ Ada is a hybrid programming language.
- ü Ada supports a rendezvous concept.
- ü Ada offers methods for run-time checking and exception handling.

Question referring to Chapter 6.4

The following procedure should be implemented in Ada using two tasks:

A pedestrian is waiting for a taxi and wants to get to the church. He pays the ride and enters the church.

On the next slide there a three different code examples for this problem. Which is the correct one and what are the others doing?

Question referring to Chapter 6.4 - code examples

Alternative 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 - Implements only the timing synchronization without joint code execution

Alternative 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;
```

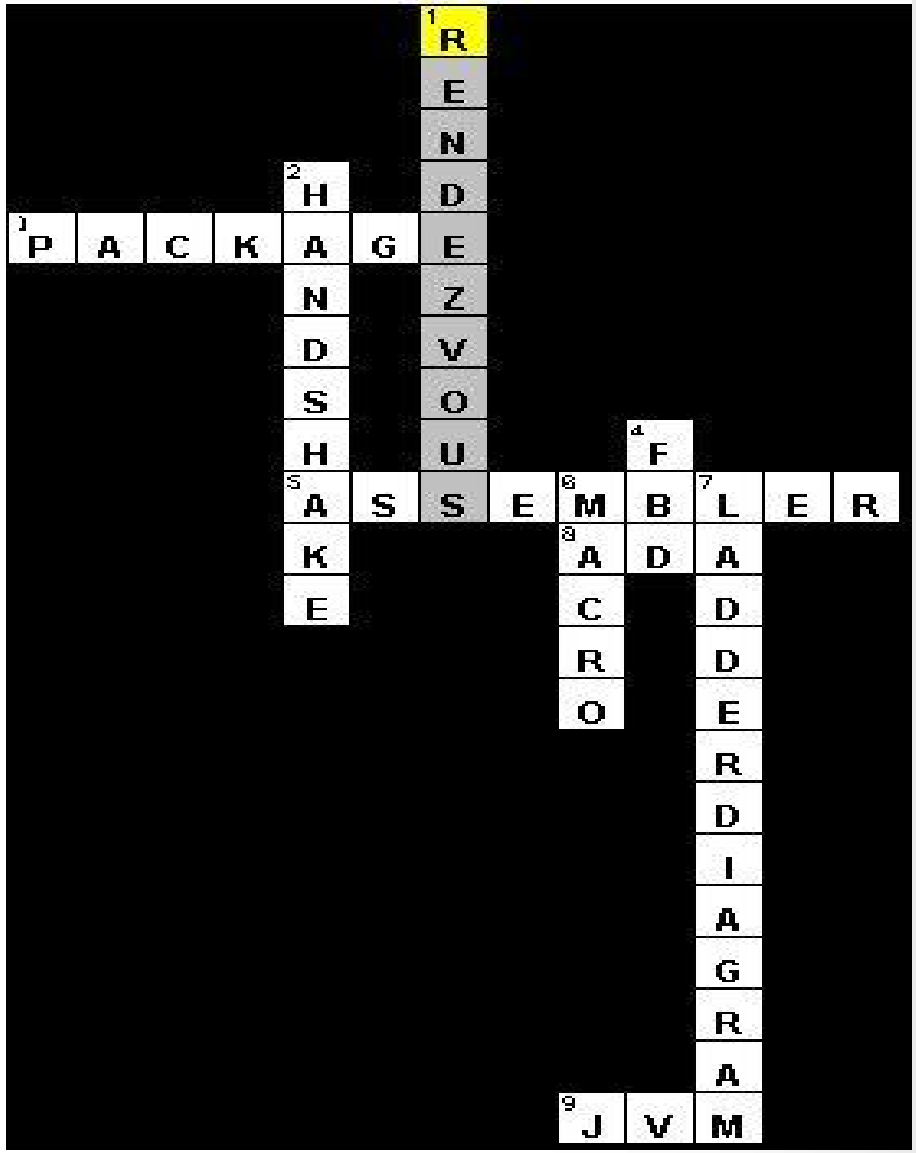
ü- Implements the timing synchronization with joint code execution

Alternative 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 - Does not implement the mutual synchronization

Crosswords to Chapter 6



Crosswords to Chapter 6

Across

3 Program unit of Ada (7)

5 Low-level programming language (9)

8 Real time programming language (3)

9 Basis for the machine independence of Java (3)

Down

1 Ada concept for synchronization (10)

2 Definition of timely synchronization points between the tasks (9)

4 Abbreviation of a PLC programming language which uses building blocks (3)

6 Auxiliary mean to simplify assembler languages (5)

4 Representation of a PLC program similar to a circuit diagram (6,7)