

Introduction & Languages

Uwe & Zimmer - The Australian National University

Introduction & Languages

What is a real-time system?

Definition of a Real-Time System

- The correctness of a real-time system depends on:
 1. The **logical correctness** (accuracy) of the results ... as well as ...
 2. The **time when the result is delivered.** ... both with respect to the specification.

Real-Time systems are frequently evaluated as part of a physical system.

Introduction & Languages

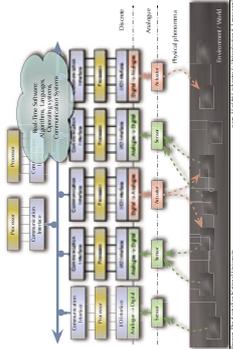
Simple example: Brake manager

Ways to define reliability

- Full fault tolerant, graceful degradation or fail safe?
- Redundancies?
- Testing?
- Verification?
- (Physical) Modularization?
- Use an algebraic / real-time logic tool? Proof correctness?
- Use a predictable runtime environment and language? Certification? Test all relevant cases?
- Assume things will still go bad? Provide fallback of

Introduction & Languages

Real-Time Systems Components



Introduction & Languages

References

Berry, Gerard
The Essential 65 Language
Piner Version 5.57
138 (2 pp. 243-271)
[Lee2000]
Lee, Edward
Comparing Needs, Time
and Real-Time Program-
ming in Ada, edition, Cam-
bridge University Press (2007)
[NN1995]
Nanni, S. & Williams, Andy
Concurrent and Real-Time
Programming in Ada
Cambridge Univer-
sity Press (2007)
[Dawes1995]
Dawes, Peter & Schaefer, S.
Ada Reference Manual, 3rd
Edition
Cambridge University Press (1995, vol. 3, 1.D.)

Introduction & Languages

What is a real-time system?

Real-Time Systems Scenarios

- All sizes and complexities: From heating regulators over mobile phones to high speed trains, aircraft, satellites, space stations) ...
 - Situated. Almost always part of or coupled to a physical system.
 - Relevant: Vital components of our traffic and communication infrastructure among many other essential systems.
 - Dangerous: Failures often lead to loss of life, or environmental damage.
- Real-Time Systems require a specific understanding and skill set.

Introduction & Languages

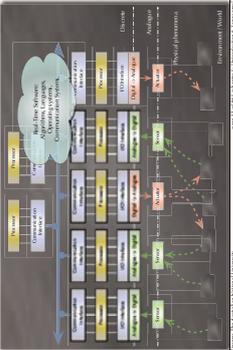
Simple example: Brake manager

Ways to implement this (hardware)

- The brakes:
- Mechanical (hydraulics), + overwrite valves
 - Digitally controlled, (no mechanical connection)
 - Brake lights
- The controllers:
- Single CPU
 - Multiple CPUs + shared memory
 - Multiple CPUs + point-to-point connections
 - Multiple CPUs + communication system (e.g. a bus system)
 - Redundant CPUs

Introduction & Languages

Embedded Real-Time Systems Components



Introduction & Languages

What is a real-time system?

Features of a Real-Time System?

- Fast context switches?
- Small size?
- Quick responses to external interrupts?
- Multitasking?
- 'Low level' programming interfaces?
- Inter-process communication tools?
- High processor utilization?
- Fast systems?

Introduction & Languages

What is a real-time system?

Typical characteristics of a Real-Time System

- Adherence to set time constraints
 - Predictability
 - Fault tolerance
 - Accuracy
 - Frequency: concurrent, distributed and employing complex hardware.
- Not too early – not too late
- Repeatable results in time and value
- Robustness in the presence of foreseeable faults
- Results are precise enough to drive e.g. a physical systems

Introduction & Languages

Simple example: Brake manager

Ways to implement this (software)

- Sequential, concurrent or distributed?
- Shared memory or message passing?
- Synchronous or asynchronous communication?
- Dynamic or fixed scheduler?
- Imperative, functional, or dataflow programming?
- Predefined communication channels
- Data driven or global clock synchronization?
- Polling interrupt driven, or event driven?
- Globally synchronous, individually asynchronous I/O channels?
- Languages/tools which lend themselves to verification and validation?
- Languages/tools which lend themselves to certification and accreditation?

Introduction & Languages

Real-Time Programming Languages

Relevant Programming Paradigms

- Control flow: Imperative → Declarative
- Declarative: Functional → (Logic) → Finite State Machines
- Allocations and bindings: Static → Dynamic
- Time: Event-driven → Discrete → Synchronous → Continuous
- Focus: Control flow-oriented → Data flow-oriented
- Degree of concurrency: Sequential → Concurrent → Distributed
- Structure: Modular → Generics → Templates → (Object-Oriented) → (Aspect-Oriented)
- Determinism: Deterministic → Non-deterministic

Introduction & Languages

What is a real-time system?

Features of a Real-Time System?

- Fast context switches? • Should be fast anyway!
- Small size? • Should be small anyway!
- Quick responses to external interrupts? • Predictable! – not quick!
- Multitasking? • Real time systems are often multitasking systems!
- Low level programming interfaces? • Needed in many systems!
- Inter-process communication tools? • Needed for any concurrency!
- High processor utilization? • Just the opposite usually! (redundancy)
- Fast systems? • Predictable! – not 'fast'

Introduction & Languages

Simple example: Brake manager

Latencies:

- Constant.
- Significantly shorter than the driver's response time.

Reliability:

- Provide robustness under all for foreseeable and "manageable" failures.

Efficient design:

- Mass producible?

Introduction & Languages

What is a real-time system?

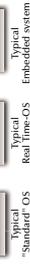
Typical Real-Time Operating System

Often implemented as an integrated run-time environment, i.e. there is 'no operating system' (as embedded systems).

RTOs provide:

- Predictability
- Passivity

- Small footprint
- Instrumentation



Introduction & Languages

Real-Time Programming Languages

Requirements for Real-Time Languages / Environments

- Predictability
 - No operations shall lead to unforeseeable timing behaviours.
- Time
 - Specified granularity, operations based on time, scheduling
- High integrity
 - Strong compiler and runtime environments detecting faults as early as possible.
 - Complete, unambiguous language definition.
- Concurrency and Distribution
 - Solid, high-level synchronization and communication primitives, automated data marshalling.
- Specific yet Scaling
 - Adapting physical interfaces into high-level data types and programming "in the very large".

Introduction & Languages

Real-Time Programming Languages

Requirements for Real-Time Language/Environments

- **Predictability**
 - No operations that could cause timing behaviours.
- **Time**
 - Synchronised (family) operations based on time, including:
 - **deadlines** (time when an operation must complete)
 - **release times** (time when an operation can start)
 - **periods** (time between successive releases)
- **High integrity**
 - **Some times used as hammers!**
- **Concurrency and Distribution**
 - Solid, high-level synchronisation and distribution mechanisms
 - **Specific yet general hammers!**
 - Mapping physical interfaces into high-level data-types and programming "in the very large".

Copyright © 2010, Pearson Education, Inc. All rights reserved. ISBN 978-0-13-200-000-0

Introduction & Languages

Ada

A crash course

... refreshing for some, x-th-language introduction for others:

- **Specification and implementation** (body parts, basic types)
- **Exceptions**
- **Information hiding** in specifications ('private')
- **Contracts**
- **Generic programming** (polymorphism)
- **Tasking**
- **Monitors and synchronisation** ('protected', 'entries', 'selects', 'accepts')
- **Groups of interfaces** ('entry families')
- **Abstract types and dispatching**

Not mentioned here: general object orientation, dynamic memory management, foreign language interfaces, marshalling, bases of imperative programming...

Copyright © 2010, Pearson Education, Inc. All rights reserved. ISBN 978-0-13-200-000-0

Introduction & Languages

A simple queue specification

```

package Queue_Pack_Simple is
  QueueSize : constant Positive := 10;
  type Element is new Positive range 1..QueueSize;
  type Marker is mod QueueSize;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;

```

Copyright © 2010, Pearson Education, Inc. All rights reserved. ISBN 978-0-13-200-000-0

Introduction & Languages

A simple queue specification

```

package Queue_Pack_Simple is
  QueueSize : constant Positive := 10;
  type Element is new Positive range 1..QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;

```

Copyright © 2010, Pearson Education, Inc. All rights reserved. ISBN 978-0-13-200-000-0

Introduction & Languages

Real-Time Programming Languages

Real-Time Languages, Operating Systems and Libraries

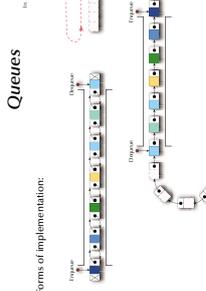
What if you wanted to use real-time languages and you need to formulate some/all real-time constraints outside the programming language?

- **Real-time operating systems:**
 - Scheduling, interrupt handling, (potentially other features) integrate from the compiler environment into the operating system.
 - Compiler level analysis is replaced by equivalent tools on the OS level. This requires additional languages, as those tools need specifications as well.
- **Libraries:**
 - Loss of all compiler-level checks.
 - Loss of all block structure and scoping.

Copyright © 2010, Pearson Education, Inc. All rights reserved. ISBN 978-0-13-200-000-0

Introduction & Languages

Data structure example



Copyright © 2010, Pearson Education, Inc. All rights reserved. ISBN 978-0-13-200-000-0

Introduction & Languages

A simple queue specification

```

package Queue_Pack_Simple is
  QueueSize : constant Positive := 10;
  type Element is new Positive range 1..QueueSize;
  type Marker is mod QueueSize;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;

```

Copyright © 2010, Pearson Education, Inc. All rights reserved. ISBN 978-0-13-200-000-0

Introduction & Languages

A simple queue specification

```

package Queue_Pack_Simple is
  QueueSize : constant Positive := 10;
  type Element is new Positive range 1..QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;

```

Copyright © 2010, Pearson Education, Inc. All rights reserved. ISBN 978-0-13-200-000-0

Introduction & Languages

Real-Time Programming Languages

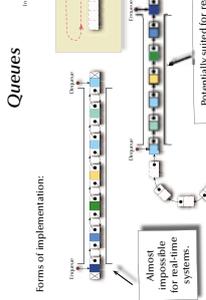
Some RT-Languages

- **Ada** (Ada2012) for general workloads.
- **Real-Time Java** (Real-Time Specification for Java 1.1) for soft real-time applications.
- **Esterel** (Esterel v7) for an alternative for high-integrity applications.
- **VHDL** for Compile real-time data flows and independent asynchronous control paths directed to hardware.
- **Timed CSP** (as used and developed since 1980) for an algebraic approach.
- **PEARL** (PEARL-90) for a traditional language, specialized on plan modelling.
- **POSIX** (POSIX 1003.1b...) for The libraries of bare bone integrals and semaphores.
- **Assemblers / C** for The languages of bare bone integrals and semaphores.

Copyright © 2010, Pearson Education, Inc. All rights reserved. ISBN 978-0-13-200-000-0

Introduction & Languages

Data structure example



Copyright © 2010, Pearson Education, Inc. All rights reserved. ISBN 978-0-13-200-000-0

Introduction & Languages

A simple queue specification

```

package Queue_Pack_Simple is
  QueueSize : constant Positive := 10;
  type Element is new Positive range 1..QueueSize;
  type Marker is mod QueueSize;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;

```

Copyright © 2010, Pearson Education, Inc. All rights reserved. ISBN 978-0-13-200-000-0

Introduction & Languages

A simple queue specification

```

package Queue_Pack_Simple is
  QueueSize : constant Positive := 10;
  type Element is new Positive range 1..QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;

```

Copyright © 2010, Pearson Education, Inc. All rights reserved. ISBN 978-0-13-200-000-0

Introduction & Languages

Languages explicitly supporting concurrency: e.g. Ada

Ada is an ISO standardized (ISO/IEC 10622:1987) 'general purpose' language with focus on "program reliability and maintenance, programming as a human activity, and efficiency".

- It provides **core language primitives** for:
 - Strong typing, contracts, separate compilation, object-orientation.
 - Concurrency, message-passing, synchronization, monitors, rps, timeouts, scheduling, priority ceiling locks, hardware mappings, fully typed network communication.
 - Strong run-time environments (incl. stand-alone execution).
- ... as well as **standardized language annexes** for:
 - Additional and time features, distributed programming, system-level programming, numeric, information systems, distributed programming, system-level programming, numeric, information systems, safety and security issues.

Copyright © 2010, Pearson Education, Inc. All rights reserved. ISBN 978-0-13-200-000-0

Introduction & Languages

Ada

Basics

- ... introducing:
- **Specification and implementation** (body parts)
- **Constants**
- **Some basic types** (integer specifics)
- **Some type attributes**
- **Parameter specification**

Copyright © 2010, Pearson Education, Inc. All rights reserved. ISBN 978-0-13-200-000-0

Introduction & Languages

A simple queue specification

```

package Queue_Pack_Simple is
  QueueSize : constant Positive := 10;
  type Element is new Positive range 1..QueueSize;
  type Marker is mod QueueSize;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;

```

Copyright © 2010, Pearson Education, Inc. All rights reserved. ISBN 978-0-13-200-000-0

Introduction & Languages

A simple queue specification

```

package Queue_Pack_Simple is
  QueueSize : constant Positive := 10;
  type Element is new Positive range 1..QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;

```

Copyright © 2010, Pearson Education, Inc. All rights reserved. ISBN 978-0-13-200-000-0

Introduction & Languages

A simple queue specification

```

package Queue_Pack_Simple is
  QueueSize : constant Positive := 10;
  type Element is new Positive range 1..QueueSize;
  type Marker is array (Marker) of Element;
  type Queue_Type is record
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
  end Queue_Pack_Simple;

```

...anything on this slide
all not perfectly clear!

Introduction & Languages

A simple queue implementation

```

package body Queue_Pack_Simple is
  procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
  end Enqueue;
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
  begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
  end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is
  (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
  (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  end Queue_Pack_Simple;

```

Side-effect free, single-expression functions can be compressed with no right-hand blocks.

Introduction & Languages

A simple queue implementation

```

package body Queue_Pack_Simple is
  procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
  end Enqueue;
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
  begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
  end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is
  (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
  (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  end Queue_Pack_Simple;

```

Implementations in Ada are syntactically enclosed in a package body block.

Introduction & Languages

A simple queue implementation

```

package body Queue_Pack_Simple is
  procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
  end Enqueue;
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
  begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
  end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is
  (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
  (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  end Queue_Pack_Simple;

```

Modulo type, hence no index checks required.

Introduction & Languages

A simple queue implementation

```

package body Queue_Pack_Simple is
  procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
  end Enqueue;
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
  begin
    Queue.Top := Queue.Top + 1;
  end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is
  (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
  (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  end Queue_Pack_Simple;

```

Boolean expressions

Introduction & Languages

A simple queue implementation

```

package body Queue_Pack_Simple is
  procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
  end Enqueue;
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
  begin
    Queue.Top := Queue.Top + 1;
  end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is
  (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
  (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  end Queue_Pack_Simple;

```

Introduction & Languages

A simple queue implementation

```

package body Queue_Pack_Simple is
  procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
  end Enqueue;
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
  begin
    Queue.Top := Queue.Top + 1;
  end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is
  (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
  (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  end Queue_Pack_Simple;

```

anything on this slide
still not perfectly clear!

Introduction & Languages

A simple queue test program

```

with Queue_Pack_Simple; use Queue_Pack_Simple;
package Queue_Test_Simple is
  Item : Element;
  Enqueue (2008, Queue);
  Dequeue (Item, Queue);
  end Queue_Test_Simple;

```

Introduction & Languages

A simple queue test program

```

with Queue_Pack_Simple; use Queue_Pack_Simple;
procedure Queue_Test_Simple is
  Queue : Queue_Type;
  Item : Element;
  Enqueue (2008, Queue);
  Dequeue (Item, Queue);
  end Queue_Test_Simple;

```

Importing items from other packages is done with with-clauses, with qualifying them with the package name.

Introduction & Languages

A simple queue test program

```

with Queue_Pack_Simple; use Queue_Pack_Simple;
procedure Queue_Test_Simple is
  Item : Element;
  Enqueue (2008, Queue);
  Dequeue (Item, Queue);
  end Queue_Test_Simple;

```

A top level procedure is read as the code which needs to be executed.

Introduction & Languages

A simple queue test program

```

with Queue_Pack_Simple; use Queue_Pack_Simple;
procedure Queue_Test_Simple is
  Item : Element;
  Enqueue (2008, Queue);
  Dequeue (Item, Queue);
  end Queue_Test_Simple;

```

Variables are declared in Ada style: "Item of type Element".

Introduction & Languages

A simple queue test program

```

with Queue_Pack_Simple; use Queue_Pack_Simple;
procedure Queue_Test_Simple is
  Queue : Queue_Type;
  Item : Element;
  Enqueue (2008, Queue);
  Dequeue (Item, Queue);
  end Queue_Test_Simple;

```

Will produce a result according to the chosen initialization: Boolean="invalid data" exception if initialized to invalids. ... hm, ok... so this was rubbish...

Introduction & Languages

A simple queue test program

```

with Queue_Pack_Simple; use Queue_Pack_Simple;
procedure Queue_Test_Simple is
  Queue : Queue_Type;
  Item : Element;
  Enqueue (2008, Queue);
  Dequeue (Item, Queue);
  end Queue_Test_Simple;

```

... anything on this slide
all not perfectly clear!

Introduction & Languages

Ada Exceptions

```

... introducing
• Enumeration handling
• Exception types
• Type attributed operators

```

Introduction & Languages

A queue specification with proper exceptions

```

package Queue_Pack_Exceptions is
  QueueSize : constant Positive := 10;
  type Element is (Up, Down, Spin, Turn);
  type Marker is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
  end Queue_Pack_Exceptions;

```

Introduction & Languages

A queue specification with proper exceptions

```

package Queue_Pack_Exceptions is
  QueueSize : constant Positive := 10;
  type Element is (Up, Down, Spin, Turn);
  type Marker is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
  end Queue_Pack_Exceptions;

```

Enumeration types are first-class types and can be used, e.g. as array indices. The representation values can be controlled and do not purpose like interfacing with hardware.

Introduction & Languages

```

package Queue_Pack_Exceptions is
  QueueSize : constant Integer := 10;
  type Element is (Up, Down, Spin, Turn);
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker;
    Elements : List;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
  (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  Queue_Overflow, Queue_Underflow : exception;
end Queue_Pack_Exceptions;

```

Nothing else changes in the specifications.

Exceptions need to be declared.

A queue implementation with proper exceptions

```

package body Queue_Pack_Exceptions is
  procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
  begin
    if Is_Full (Queue) then
      raise Queue_Overflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Marker_Succ (Queue.Free);
    end Enqueue;
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
  begin
    if Is_Empty (Queue) then
      raise Queue_Underflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Marker_Succ (Queue.Top);
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
end Queue_Pack_Exceptions;

```

All types come with a long list of built-in operators. Symantically-expressed attributes.

Type attributes often make code more generic. 'Succ' works for instance on enumeration types as well... '4' does not.

Introduction & Languages

```

package Queue_Pack_Exceptions;
with Ada.Text_IO;
procedure Queue_Test_Exceptions is
  Item : Element;
begin
  Enqueue (Turn, Queue);
  Dequeue (Item, Queue);
  exception
  when Queue_Underflow => Put ("Queue underflow");
  when Queue_Overflow => Put ("Queue overflow");
end Queue_Test_Exceptions;

```

anything on this slide still not perfectly clear?

Introduction & Languages

```

package Queue_Pack_Private is
  QueueSize : constant Integer := 10;
  type Element is (Up, Down, Spin, Turn);
  type Queue_Type is limited private;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
  Queue_Overflow, Queue_Underflow : exception;
private
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker;
    Elements : List;
  end record;
end Queue_Pack_Private;

```

private splits the specification in a public and a private section.

The private section is only here so that the specifications can be separately compiled.

Introduction & Languages

```

package Queue_Pack_Exceptions is
  QueueSize : constant Positive := 10;
  type Element is (Up, Down, Spin, Turn);
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker;
    Elements : List;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
  (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  Queue_Overflow, Queue_Underflow : exception;
end Queue_Pack_Exceptions;

```

anything on this slide still not perfectly clear?

A queue implementation with proper exceptions

```

package body Queue_Pack_Exceptions is
  procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
  begin
    if Is_Full (Queue) then
      raise Queue_Overflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Marker_Succ (Queue.Free);
    end Enqueue;
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
  begin
    if Is_Empty (Queue) then
      raise Queue_Underflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Marker_Succ (Queue.Top);
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
end Queue_Pack_Exceptions;

```

anything on this slide still not perfectly clear?

Introduction & Languages

```

package Queue_Pack_Exceptions is
  QueueSize : constant Positive := 10;
  type Element is (Up, Down, Spin, Turn);
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker;
    Elements : List;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is (Queue.Is_Full);
  (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  Queue_Overflow, Queue_Underflow : exception;
end Queue_Pack_Exceptions;

```

This package provides access to internal structures which can lead to inconsistent access.

anything on this slide still not perfectly clear?

Introduction & Languages

```

package Queue_Pack_Private is
  QueueSize : constant Integer := 10;
  type Element is (Up, Down, Spin, Turn);
  type Queue_Type is limited private;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
  Queue_Overflow, Queue_Underflow : exception;
private
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker;
    Elements : List;
  end record;
end Queue_Pack_Private;

```

Limited disables assignment and comparison. A comparison now e.g. not be able to make a copy of a Queue_Type value.

A queue implementation with proper exceptions

```

package Queue_Pack_Exceptions is
  procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
  begin
    if Is_Full (Queue) then
      raise Queue_Overflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Marker_Succ (Queue.Free);
    end Enqueue;
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
  begin
    if Is_Empty (Queue) then
      raise Queue_Underflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Marker_Succ (Queue.Top);
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
end Queue_Pack_Exceptions;

```

Introduction & Languages

```

package Queue_Pack_Exceptions;
with Ada.Text_IO;
procedure Queue_Test_Exceptions is
  Item : Element;
begin
  Enqueue (Turn, Queue);
  Dequeue (Item, Queue);
  exception
  when Queue_Underflow => Put ("Queue underflow");
  when Queue_Overflow => Put ("Queue overflow");
end Queue_Test_Exceptions;

```

Introduction & Languages

```

...introducing
• Private declarations
  are needed to compile specifications.
  yet not accessible for a user of the package.
• Limited private types as entity cannot be assigned or compared

```

Introduction & Languages

```

package Queue_Pack_Private is
  QueueSize : constant Integer := 10;
  type Element is (Up, Down, Spin, Turn);
  type Queue_Type is limited private;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
  Queue_Overflow, Queue_Underflow : exception;
private
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker;
    Elements : List;
  end record;
end Queue_Pack_Private;

```

Alternatively, "=" and "<" operations can be replaced with type-specific versions which do not result in errors. Operations can be allowed.

A queue implementation with proper exceptions

```

package body Queue_Pack_Exceptions is
  procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
  begin
    if Is_Full (Queue) then
      raise Queue_Overflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Marker_Succ (Queue.Free);
    Queue.Is_Empty := False;
  end Enqueue;
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
  begin
    if Is_Empty (Queue) then
      raise Queue_Underflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Marker_Succ (Queue.Top);
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
end Queue_Pack_Exceptions;

```

Raises exceptions break the control flow and "propagate" to the closest "exception handler" in the call chain.

Introduction & Languages

```

package Queue_Pack_Exceptions;
with Ada.Text_IO;
procedure Queue_Test_Exceptions is
  Item : Element;
begin
  Enqueue (Turn, Queue);
  Dequeue (Item, Queue);
  exception
  when Queue_Underflow => Put ("Queue underflow");
  when Queue_Overflow => Put ("Queue overflow");
end Queue_Test_Exceptions;

```

An exception handler has a choice to handle pass, or re-raise the exception, or a different exception.

Control flow is continued after the exception handler in case of a handled exception.

Introduction & Languages

```

package Queue_Pack_Private is
  QueueSize : constant Integer := 10;
  type Element is new Positive range 1..1000;
  type Queue_Type is limited private;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
  Queue_Overflow, Queue_Underflow : exception;
private
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker;
    Elements : List;
  end record;
end Queue_Pack_Private;

```

Introduction & Languages

```

package Queue_Pack_Private is
  QueueSize : constant Integer := 10;
  type Element is new Positive range 1..1000;
  type Queue_Type is limited private;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
  Queue_Overflow, Queue_Underflow : exception;
private
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker;
    Elements : List;
  end record;
end Queue_Pack_Private;

```

anything on this slide still not perfectly clear?

A generic protected queue specification

```

generic
  Element is private;
  type Index is mod 0; -- Modulo defines size of the queue.
  package Queue_Pack_Protected_Generic is
    type Queue_Type is limited private;
    protected type Protected_Queue is
      entry Enqueue (Item : out Element);
      procedure Dequeue_Queue;
      function Is_Empty return Boolean;
      function Is_Full return Boolean;
    private
      Queue : Queue_Type;
      entry, procedure, function
    private
      type List is array (Index) of Element;
      Top, Free : Index := Index'First;
      Is_Empty : Boolean := True;
      Elements : List;
    end Protected_Queue;
  end Queue_Pack_Protected_Generic;

```

Queue is protected for safe concurrent access. Three categories of access routines are distinguished by the keywords entry, procedure, function

ENTRANCE TO A GENERIC PACKAGE BODY

A generic protected queue implementation

```

generic
  Element is private;
  type Index is mod 0; -- Modulo defines size of the queue.
  package Queue_Pack_Protected_Generic is
    type Queue_Type is limited private;
    protected type Protected_Queue is
      entry Enqueue (Item : out Element);
      procedure Dequeue_Queue;
      function Is_Empty return Boolean;
      function Is_Full return Boolean;
    private
      Queue : Queue_Type;
      entry, procedure, function
    private
      type List is array (Index) of Element;
      Top, Free : Index := Index'First;
      Is_Empty : Boolean := True;
      Elements : List;
    end Protected_Queue;
  end Queue_Pack_Protected_Generic;

```

... anything on this slide still not perfectly clear?

ENTRANCE TO A GENERIC PACKAGE BODY

A generic protected queue test program

```

with Ada_Task_Identification;
with Ada_Text_IO;
protected Generic;
procedure Queue_Test_Protected_Generic is
  type Queue_Size is mod 3;
  package Queue_Pack_Protected_Character is
    new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
  use Queue_Pack_Protected_Character;
  Queue : Protected_Queue;
  type Task_Index is range 1 .. 3;
  task type Producer;
  task type Consumer;
  Producers : array (Task_Index) of Producer;
  Consumers : array (Task_Index) of Consumer;
begin
  null;
end Queue_Test_Protected_Generic;

```

ENTRANCE TO A GENERIC PACKAGE BODY

A generic protected queue test program

```

with Ada_Task_Identification;
with Ada_Text_IO;
protected Generic;
procedure Queue_Test_Protected_Generic is
  type Queue_Size is mod 3;
  package Queue_Pack_Protected_Character is
    new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
  use Queue_Pack_Protected_Character;
  Queue : Protected_Queue;
  type Task_Index is range 1 .. 3;
  task type Producer;
  task type Consumer;
  Producers : array (Task_Index) of Producer;
  Consumers : array (Task_Index) of Consumer;
begin
  null;
end Queue_Test_Protected_Generic;

```

... anything on this slide still not perfectly clear?

ENTRANCE TO A GENERIC PACKAGE BODY

A generic protected queue specification

```

generic
  Element is private;
  type Index is mod 0; -- Modulo defines size of the queue.
  package Queue_Pack_Protected_Generic is
    type Queue_Type is limited private;
    protected type Protected_Queue is
      entry Enqueue (Item : out Element);
      procedure Dequeue_Queue;
      function Is_Empty return Boolean;
      function Is_Full return Boolean;
    private
      Queue : Queue_Type;
      entry, procedure, function
    private
      type List is array (Index) of Element;
      Top, Free : Index := Index'First;
      Is_Empty : Boolean := True;
      Elements : List;
    end Protected_Queue;
  end Queue_Pack_Protected_Generic;

```

Procedures are mutually exclusive to all other access routines.

Rationale: Procedures can modify the protected queue. Hence, they need a guarantee for exclusive access.

ENTRANCE TO A GENERIC PACKAGE BODY

A generic protected queue implementation

```

package body Queue_Pack_Protected_Generic is
  protected body Protected_Queue is
    entry Enqueue (Item : out Element) when not Is_Full is
      Queue.Elements (Queue.Free) := Item; Queue.Free := Index'Succ (Queue.Free);
      Queue.Is_Empty := False;
    end Enqueue;
    and Enqueue (Item : out Element) when not Is_Empty is
      begin
        Queue := Queue.Elements (Queue.Top); Queue.Top := Index'Succ (Queue.Top);
        Queue.Is_Empty := Queue.Top = Queue.Free;
      end procedure Empty_Queue;
    end Protected_Queue;
  end Queue_Pack_Protected_Generic;

```

ENTRANCE TO A GENERIC PACKAGE BODY

A generic protected queue test program

```

with Ada_Task_Identification;
with Ada_Text_IO;
protected Generic;
procedure Queue_Test_Protected_Generic is
  type Queue_Size is mod 3;
  package Queue_Pack_Protected_Character is
    new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
  use Queue_Pack_Protected_Character;
  Queue : Protected_Queue;
  type Task_Index is range 1 .. 3;
  task type Producer;
  task type Consumer;
  Producers : array (Task_Index) of Producer;
  Consumers : array (Task_Index) of Consumer;
begin
  null;
end Queue_Test_Protected_Generic;

```

If more than one instance of a specific task is declared, the task must be opposed to a concrete task is declared.

ENTRANCE TO A GENERIC PACKAGE BODY

A generic protected queue test program

```

with Ada_Task_Identification;
with Ada_Text_IO;
protected Generic;
procedure Queue_Test_Protected_Generic is
  type Queue_Size is mod 3;
  package Queue_Pack_Protected_Character is
    new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
  use Queue_Pack_Protected_Character;
  Queue : Protected_Queue;
  type Task_Index is range 1 .. 3;
  task type Producer;
  task type Consumer;
  Producers : array (Task_Index) of Producer;
  Consumers : array (Task_Index) of Consumer;
begin
  null;
end Queue_Test_Protected_Generic;

```

ENTRANCE TO A GENERIC PACKAGE BODY

A generic protected queue specification

```

generic
  Element is private;
  type Index is mod 0; -- Modulo defines size of the queue.
  package Queue_Pack_Protected_Generic is
    type Queue_Type is limited private;
    protected type Protected_Queue is
      entry Enqueue (Item : out Element);
      procedure Dequeue_Queue;
      function Is_Empty return Boolean;
      function Is_Full return Boolean;
    private
      Queue : Queue_Type;
      entry, procedure, function
    private
      type List is array (Index) of Element;
      Top, Free : Index := Index'First;
      Is_Empty : Boolean := True;
      Elements : List;
    end Protected_Queue;
  end Queue_Pack_Protected_Generic;

```

Functions are mutually exclusive to all other access routines.

Rationale: The compiler enforces these functions to be side-effect-free with respect to the protected data. Hence concurrent access can be granted among functions without risk.

ENTRANCE TO A GENERIC PACKAGE BODY

A generic protected queue implementation

```

package body Queue_Pack_Protected_Generic is
  protected body Protected_Queue is
    entry Enqueue (Item : out Element) when not Is_Full is
      Queue.Elements (Queue.Free) := Item; Queue.Free := Index'Succ (Queue.Free);
      Queue.Is_Empty := False;
    end Enqueue;
    and Enqueue (Item : out Element) when not Is_Empty is
      begin
        Queue := Queue.Elements (Queue.Top); Queue.Top := Index'Succ (Queue.Top);
        Queue.Is_Empty := Queue.Top = Queue.Free;
      end procedure Empty_Queue;
    end Protected_Queue;
  end Queue_Pack_Protected_Generic;

```

Guard expressions follow the implementation of entries.

Tasks are automatically blocked or released depending on the state of the guard.

Function Is_Full is called on entering an entry or procedure.

(not Queue.Is_Empty and then Queue.Top = Queue.Free) (no point to re-check them at any other time).

end Protected_Queue; (no point to re-check them at any other time).

end Queue_Pack_Protected; (no point to re-check them at any other time).

ENTRANCE TO A GENERIC PACKAGE BODY

A generic protected queue test program

```

with Ada_Task_Identification;
with Ada_Text_IO;
protected Generic;
procedure Queue_Test_Protected_Generic is
  type Queue_Size is mod 3;
  package Queue_Pack_Protected_Character is
    new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
  use Queue_Pack_Protected_Character;
  Queue : Protected_Queue;
  type Task_Index is range 1 .. 3;
  task type Producer;
  task type Consumer;
  Producers : array (Task_Index) of Producer;
  Consumers : array (Task_Index) of Consumer;
begin
  null;
end Queue_Test_Protected_Generic;

```

Multiple instances of a task can be maintained e.g. by declaring an array of this task type.

Tasks are started right when such an array is incremented.

ENTRANCE TO A GENERIC PACKAGE BODY

A generic protected queue test program

```

with Ada_Task_Identification;
with Ada_Text_IO;
protected Generic;
procedure Queue_Test_Protected_Generic is
  type Queue_Size is mod 3;
  package Queue_Pack_Protected_Character is
    new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
  use Queue_Pack_Protected_Character;
  Queue : Protected_Queue;
  type Task_Index is range 1 .. 3;
  task type Producer;
  task type Consumer;
  Producers : array (Task_Index) of Producer;
  Consumers : array (Task_Index) of Consumer;
begin
  null;
end Queue_Test_Protected_Generic;

```

The executable code for a task is provided in its body.

ENTRANCE TO A GENERIC PACKAGE BODY

A generic protected queue specification

```

generic
  Element is private;
  type Index is mod 0; -- Modulo defines size of the queue.
  package Queue_Pack_Protected_Generic is
    type Queue_Type is limited private;
    protected type Protected_Queue is
      entry Enqueue (Item : out Element);
      procedure Dequeue_Queue;
      function Is_Empty return Boolean;
      function Is_Full return Boolean;
    private
      Queue : Queue_Type;
      entry, procedure, function
    private
      type List is array (Index) of Element;
      Top, Free : Index := Index'First;
      Is_Empty : Boolean := True;
      Elements : List;
    end Protected_Queue;
  end Queue_Pack_Protected_Generic;

```

Entries are mutually exclusive to all other access routines and also provide one guard per entry to ensure mutual exclusion.

The guard expressions are defined in the implementation part.

Rationale: Entries can be blocking even if the procedure is not. Hence a separate task waiting queue is provided per entry.

ENTRANCE TO A GENERIC PACKAGE BODY

A generic protected queue implementation

```

package body Queue_Pack_Protected_Generic is
  protected body Protected_Queue is
    entry Enqueue (Item : Element) when not Is_Full is
      Queue.Elements (Queue.Free) := Item; Queue.Free := Index'Succ (Queue.Free);
      Queue.Is_Empty := False;
    end Enqueue;
    and Enqueue (Item : out Element) when not Is_Empty is
      begin
        Item := Queue.Elements (Queue.Top); Queue.Top := Index'Succ (Queue.Top);
        Queue.Is_Empty := Queue.Top = Queue.Free;
      end procedure Empty_Queue;
    end Protected_Queue;
  end Queue_Pack_Protected_Generic;

```

Anything on this slide still not perfectly clear?

ENTRANCE TO A GENERIC PACKAGE BODY

A generic protected queue test program

```

with Ada_Task_Identification;
with Ada_Text_IO;
protected Generic;
procedure Queue_Test_Protected_Generic is
  type Queue_Size is mod 3;
  package Queue_Pack_Protected_Character is
    new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
  use Queue_Pack_Protected_Character;
  Queue : Protected_Queue;
  type Task_Index is range 1 .. 3;
  task type Producer;
  task type Consumer;
  Producers : array (Task_Index) of Producer;
  Consumers : array (Task_Index) of Consumer;
begin
  null;
end Queue_Test_Protected_Generic;

```

Often there are statements for the "task task" queue explicitly stated by a null statement.

ENTRANCE TO A GENERIC PACKAGE BODY

A generic protected queue test program

```

with Ada_Task_Identification;
with Ada_Text_IO;
protected Generic;
procedure Queue_Test_Protected_Generic is
  type Queue_Size is mod 3;
  package Queue_Pack_Protected_Character is
    new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
  use Queue_Pack_Protected_Character;
  Queue : Protected_Queue;
  type Task_Index is range 1 .. 3;
  task type Producer;
  task type Consumer;
  Producers : array (Task_Index) of Producer;
  Consumers : array (Task_Index) of Consumer;
begin
  null;
end Queue_Test_Protected_Generic;

```

subtype Some_Characters is Character range 'a' .. 'f';

task body Producer is

for Ch in Some_Characters loop

Put_Line ("Task " & Image (Current_Task) & " finds the queue to be " & (if Queue.Is_Empty then "empty" else "not empty") & "

and

if not Is_Full then "fill" else "not full") & "

and prepares to add: " & Character'Image (Ch) & " to the queue.");

Queue.Enqueue (Ch); -- task might be blocked here!

end loop;

end Producer;

There are three of these tasks and they are all hammering the queue at full CPU speed.

ENTRANCE TO A GENERIC PACKAGE BODY

Introduction & Languages

An queue task implementation (cont.)

```

( )
or
  accept Is_Empty (Result : out Boolean) do
    Result := Is_Empty;
  end Is_Empty;
or
  accept Is_Full (Result : out Boolean) do
    Result := Is_Full;
  end Is_Full;
or terminate;
end select;
end loop;
end Queue_Task;
end Queue_Task_Generic;

```

Since task terminates if all potentially calling tasks are terminated themselves.

Introduction & Languages

Ada

Abstract types & dispatching

- Abstract tagged types & subroutines (Interfaces)
- Concrete implementation of abstract types
- Dynamic dispatching to different packages, tasks, protected types or partitions.
- Synchronous message passing.

A generic queue task test program

```

with Ada.Text_IO; use Ada.Text_IO;
procedure Queue_Task_Test_Proc_Generic is
package Queue_Task_Test_Character is
  use Queue_Task_Test_Character;
  Queue : Protected_Queue;
  task Producer is end Producer;
  task Consumer is end Consumer;
end Queue_Task_Test_Character;
( )

```

Identical to the test program for protected objects.

Introduction & Languages

Ada

Abstract types & dispatching

- Abstract tagged types & subroutines (Interfaces)
- Concrete implementation of abstract types
- Dynamic dispatching to different packages, tasks, protected types or partitions.
- Synchronous message passing.

– Advanced topic –
Proceed with caution!

A generic queue task test program (cont.)

```

task body Producer is
  subtype Lower is Character range 'a' .. 'z';
begin
  for Ch in Lower loop
    Queue.Enqueue (Ch); -- task might be blocked here!
  end loop;
end Producer;
task body Consumer is
  Item : Element;
begin
  loop
    Queue.Dequeue (Item); -- task might be blocked here!
    Put ("Received: ?"); Put (Item); Put_Line ("");
    or delay 0.001;
  end loop;
end Consumer;
end Queue_Task_Test_Proc_Generic;

```

Identical to the test program for protected objects.

Introduction & Languages

An abstract queue specification

```

generic
  type Element is private;
package Queue_Pack_Abstract is
  type Queue_Interface is synchronized interface;
  procedure Enqueue (Q : in out Queue_Interface; Item : Element) is abstract;
  procedure Dequeue (Q : in out Queue_Interface; Item : out Element) is abstract;
end Queue_Pack_Abstract;

```

Introduction & Languages

An queue task specification

```

generic
  type Element is private;
  Queue_Size : Positive := 10;
package Queue_Pack_Concrete is
  task type Queue_Concrete is
    entry Is_Empty (Result : out Boolean);
    entry Is_Full (Result : out Boolean);
  end Queue_Task;
  end Queue_Pack_Concrete;

```

While this allows for a high degree of concurrency, it does not lend itself to distributed communication directly.

Introduction & Languages

An abstract queue specification

```

Motivation:
Different, derived implementations
(potentially on different computers)
can be used to implement the
same common interface as defined here.
generic
  type Element is private;
package Queue_Pack_Abstract is
  type Queue_Interface is synchronized interface;
  procedure Enqueue (Q : in out Queue_Interface; Item : Element) is abstract;
  procedure Dequeue (Q : in out Queue_Interface; Item : out Element) is abstract;
end Queue_Pack_Abstract;

```

Introduction & Languages

```

with Queue_Pack_Abstract;
generic
  package Queue_Concrete (S : Queue_Pack_Abstract (S));
  type Index is mod S; -- Module defines size of the queue.
  use Queue_Concrete;
  type Queue_Instance is limited private;
  procedure Enqueue (Q : in out Queue_Instance; Item : Element);
  overriding entry Dequeue (Q : in out Queue_Instance; Item : out Element);
  function Is_Empty (Q : in out Queue_Instance) return Boolean;
  function Is_Full (Q : in out Queue_Instance) return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
( .. ) -- as all previous private queue declarations
end Queue_Pack_Concrete;

```

Introduction & Languages

A concrete queue specification

```

with Queue_Pack_Abstract;
generic
  package Queue_Concrete (S : Queue_Pack_Abstract (S));
  type Index is mod S; -- Module defines size of the queue.
  use Queue_Concrete;
  type Queue_Instance is limited private;
  procedure Enqueue (Q : in out Queue_Instance; Item : Element);
  overriding entry Dequeue (Q : in out Queue_Instance; Item : out Element);
  function Is_Empty (Q : in out Queue_Instance) return Boolean;
  function Is_Full (Q : in out Queue_Instance) return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
( .. ) -- as all previous private queue declarations
end Queue_Pack_Concrete;

```

Introduction & Languages

```

with Queue_Pack_Abstract;
generic
  package Queue_Concrete (S : Queue_Pack_Abstract (S));
  type Index is mod S; -- Module defines size of the queue.
  use Queue_Concrete;
  type Queue_Instance is limited private;
  procedure Enqueue (Q : in out Queue_Instance; Item : Element);
  overriding entry Dequeue (Q : in out Queue_Instance; Item : out Element);
  function Is_Empty (Q : in out Queue_Instance) return Boolean;
  function Is_Full (Q : in out Queue_Instance) return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
( .. ) -- as all previous private queue declarations
end Queue_Pack_Concrete;

```

Introduction & Languages

A concrete queue specification

```

with Queue_Pack_Abstract;
generic
  package Queue_Concrete (S : Queue_Pack_Abstract (S));
  type Index is mod S; -- Module defines size of the queue.
  use Queue_Concrete;
  type Queue_Instance is limited private;
  procedure Enqueue (Q : in out Queue_Instance; Item : Element);
  overriding entry Dequeue (Q : in out Queue_Instance; Item : out Element);
  function Is_Empty (Q : in out Queue_Instance) return Boolean;
  function Is_Full (Q : in out Queue_Instance) return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
( .. ) -- as all previous private queue declarations
end Queue_Pack_Concrete;

```

Introduction & Languages

```

with Queue_Pack_Abstract;
generic
  package Queue_Concrete (S : Queue_Pack_Abstract (S));
  type Index is mod S; -- Module defines size of the queue.
  use Queue_Concrete;
  type Queue_Instance is limited private;
  procedure Enqueue (Q : in out Queue_Instance; Item : Element);
  overriding entry Dequeue (Q : in out Queue_Instance; Item : out Element);
  function Is_Empty (Q : in out Queue_Instance) return Boolean;
  function Is_Full (Q : in out Queue_Instance) return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
( .. ) -- as all previous private queue declarations
end Queue_Pack_Concrete;

```

Introduction & Languages

A concrete queue specification

```

with Queue_Pack_Abstract;
generic
  package Queue_Concrete (S : Queue_Pack_Abstract (S));
  type Index is mod S; -- Module defines size of the queue.
  use Queue_Concrete;
  type Queue_Instance is limited private;
  procedure Enqueue (Q : in out Queue_Instance; Item : Element);
  overriding entry Dequeue (Q : in out Queue_Instance; Item : out Element);
  function Is_Empty (Q : in out Queue_Instance) return Boolean;
  function Is_Full (Q : in out Queue_Instance) return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
( .. ) -- as all previous private queue declarations
end Queue_Pack_Concrete;

```

Introduction & Languages

```

with Queue_Pack_Abstract;
generic
  package Queue_Concrete (S : Queue_Pack_Abstract (S));
  type Index is mod S; -- Module defines size of the queue.
  use Queue_Concrete;
  type Queue_Instance is limited private;
  procedure Enqueue (Q : in out Queue_Instance; Item : Element);
  overriding entry Dequeue (Q : in out Queue_Instance; Item : out Element);
  function Is_Empty (Q : in out Queue_Instance) return Boolean;
  function Is_Full (Q : in out Queue_Instance) return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
( .. ) -- as all previous private queue declarations
end Queue_Pack_Concrete;

```

Other (non-overriding) methods can be added.

Introduction & Languages

A concrete queue specification

```

with Queue_Pack_Abstract;
generic
  package Queue_Concrete (S : Queue_Pack_Abstract (S));
  type Index is mod S; -- Module defines size of the queue.
  use Queue_Concrete;
  type Queue_Instance is limited private;
  procedure Enqueue (Q : in out Queue_Instance; Item : Element);
  overriding entry Dequeue (Q : in out Queue_Instance; Item : out Element);
  function Is_Empty (Q : in out Queue_Instance) return Boolean;
  function Is_Full (Q : in out Queue_Instance) return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
( .. ) -- as all previous private queue declarations
end Queue_Pack_Concrete;

```

Introduction & Languages

```

with Queue_Pack_Abstract;
generic
  package Queue_Concrete (S : Queue_Pack_Abstract (S));
  type Index is mod S; -- Module defines size of the queue.
  use Queue_Concrete;
  type Queue_Instance is limited private;
  procedure Enqueue (Q : in out Queue_Instance; Item : Element);
  overriding entry Dequeue (Q : in out Queue_Instance; Item : out Element);
  function Is_Empty (Q : in out Queue_Instance) return Boolean;
  function Is_Full (Q : in out Queue_Instance) return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
( .. ) -- as all previous private queue declarations
end Queue_Pack_Concrete;

```

...this does not require an implementation package (as all procedures are abstract) ... anything on this slide still not perfectly clear?

Introduction & Languages

A concrete queue specification

```

with Queue_Pack_Abstract;
generic
  package Queue_Concrete (S : Queue_Pack_Abstract (S));
  type Index is mod S; -- Module defines size of the queue.
  use Queue_Concrete;
  type Queue_Instance is limited private;
  procedure Enqueue (Q : in out Queue_Instance; Item : Element);
  overriding entry Dequeue (Q : in out Queue_Instance; Item : out Element);
  function Is_Empty (Q : in out Queue_Instance) return Boolean;
  function Is_Full (Q : in out Queue_Instance) return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
( .. ) -- as all previous private queue declarations
end Queue_Pack_Concrete;

```

Introduction & Languages

```

with Queue_Pack_Abstract;
generic
  package Queue_Concrete (S : Queue_Pack_Abstract (S));
  type Index is mod S; -- Module defines size of the queue.
  use Queue_Concrete;
  type Queue_Instance is limited private;
  procedure Enqueue (Q : in out Queue_Instance; Item : Element);
  overriding entry Dequeue (Q : in out Queue_Instance; Item : out Element);
  function Is_Empty (Q : in out Queue_Instance) return Boolean;
  function Is_Full (Q : in out Queue_Instance) return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
( .. ) -- as all previous private queue declarations
end Queue_Pack_Concrete;

```

A synchronous implementation of Queue_Interface with concrete implementations.

Introduction & Languages

A concrete queue specification

```

with Queue_Pack_Abstract;
generic
  package Queue_Concrete (S : Queue_Pack_Abstract (S));
  type Index is mod S; -- Module defines size of the queue.
  use Queue_Concrete;
  type Queue_Instance is limited private;
  procedure Enqueue (Q : in out Queue_Instance; Item : Element);
  overriding entry Dequeue (Q : in out Queue_Instance; Item : out Element);
  function Is_Empty (Q : in out Queue_Instance) return Boolean;
  function Is_Full (Q : in out Queue_Instance) return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
( .. ) -- as all previous private queue declarations
end Queue_Pack_Concrete;

```

Introduction & Languages

```

with Queue_Pack_Abstract;
generic
  package Queue_Concrete (S : Queue_Pack_Abstract (S));
  type Index is mod S; -- Module defines size of the queue.
  use Queue_Concrete;
  type Queue_Instance is limited private;
  procedure Enqueue (Q : in out Queue_Instance; Item : Element);
  overriding entry Dequeue (Q : in out Queue_Instance; Item : out Element);
  function Is_Empty (Q : in out Queue_Instance) return Boolean;
  function Is_Full (Q : in out Queue_Instance) return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
( .. ) -- as all previous private queue declarations
end Queue_Pack_Concrete;

```

A generic package which can be used as a parameter.

Introduction & Languages

A concrete queue specification

```

with Queue_Pack_Abstract;
generic
  package Queue_Concrete (S : Queue_Pack_Abstract (S));
  type Index is mod S; -- Module defines size of the queue.
  use Queue_Concrete;
  type Queue_Instance is limited private;
  procedure Enqueue (Q : in out Queue_Instance; Item : Element);
  overriding entry Dequeue (Q : in out Queue_Instance; Item : out Element);
  function Is_Empty (Q : in out Queue_Instance) return Boolean;
  function Is_Full (Q : in out Queue_Instance) return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
( .. ) -- as all previous private queue declarations
end Queue_Pack_Concrete;

```

Introduction & Languages

```

with Queue_Pack_Abstract;
generic
  package Queue_Concrete (S : Queue_Pack_Abstract (S));
  type Index is mod S; -- Module defines size of the queue.
  use Queue_Concrete;
  type Queue_Instance is limited private;
  procedure Enqueue (Q : in out Queue_Instance; Item : Element);
  overriding entry Dequeue (Q : in out Queue_Instance; Item : out Element);
  function Is_Empty (Q : in out Queue_Instance) return Boolean;
  function Is_Full (Q : in out Queue_Instance) return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
( .. ) -- as all previous private queue declarations
end Queue_Pack_Concrete;

```

...anything on this slide still not perfectly clear?

Introduction & Languages

A concrete queue specification

```

with Queue_Pack_Abstract;
generic
  package Queue_Concrete (S : Queue_Pack_Abstract (S));
  type Index is mod S; -- Module defines size of the queue.
  use Queue_Concrete;
  type Queue_Instance is limited private;
  procedure Enqueue (Q : in out Queue_Instance; Item : Element);
  overriding entry Dequeue (Q : in out Queue_Instance; Item : out Element);
  function Is_Empty (Q : in out Queue_Instance) return Boolean;
  function Is_Full (Q : in out Queue_Instance) return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
( .. ) -- as all previous private queue declarations
end Queue_Pack_Concrete;

```


Introduction & Languages

Esterel

Control-oriented → Dataflow-oriented

- **Dataflow-oriented:**
Continuous data-streams, functional processing (DSPs, filters, ...)
↳ Typically high bandwidth
- **Control-oriented:**
Discrete signals controlling data-streams and processes
↳ Typically low bandwidth

Introduction & Languages

Esterel

Esterel: Strong synchrony or "zero delay" assumption

- In logical terms:
 ↳ All operations are instantaneous!
- In physical terms:
 ↳ There is no observable delay between stimuli and reaction!
- In computer science terms:
 ↳ All operations are finished before the next input sampling.
 i.e. ↳ The base frequency is high enough such that the sampling can be considered instantaneous with respect to the physical system.

Introduction & Languages

Esterel

Weak aborts

- ↳ By default a code block is aborted immediately, when <signal>-occurs

```

when <signal> {
  [statements];
}
    
```

 Not sometimes a finalization semantic
 ↳ activate the code block for one last time, when <signal>-occurs!
 is more useful and expressed in Esterel as:

```

weak abort
[statements];
when <signal> {
  where the code block is now activated for a final wish, when <signal>-occurs.
    
```

Introduction & Languages

Esterel

Esterel: Control-dominated Reactive Systems

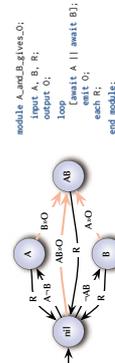
- **Real-time process control:**
↳ reaction to (sparse) stimuli in specified time-spans by emitting control signals.
- **Embedded systems / device control:**
↳ switching modes, event and emergency handling.
- **Complex systems control:**
↳ supervision, moderation and control of complex data-streams.
- **Communication protocols:**
↳ control protocol part of communication systems.
- **Human-machine interface:**
↳ switching modes, event and emergency handling.
- **Control logic (hardware):**
↳ glue logic, interconnect, pipeline control, state machines.

Introduction & Languages

Esterel

A simple, reactive, pure-signal example

Module in Esterel:



Introduction & Languages

Esterel

A simple, reactive, wrong integrator

- Specifications: a module should count the number of occurrences per second and emit this number as 'speed' once per second.
 These are no longer exclusive
- ```

var Distance := 0 ; integer in
loop
 every Metre do
 Distance := Distance + 1;
 end every;
 when Second do
 emit Speed (Distance);
 end when;
end loop;
end module;

```
- No guarantee that this block is active when 'Metre' occurs  
 Abort immediately  
 ↳ even if a 'Metre' signal occurred simultaneously

## Introduction & Languages

Esterel

### Non-causality in Synchronous Languages

- ↳ Non-reactive output:  
 module non-reactive;
 output O;
 when O
 emit O;
 end when;
end module;
- ↳ Cyclic dependencies with multiple signals:  
 module cyclic-dependency;
 output A, B;
 present A, then emit B end | present B, else emit A end |
 end module;
- ↳ All examples contain a reference to 'the future', i.e. are 'cyclic'.

## Introduction & Languages

Esterel

### Esterel: Strong synchrony or "zero delay" assumption

- In logical terms:  
 ↳ All operations are instantaneous!

## Introduction & Languages

Esterel

### A simple, reactive integrator

- Specifications: a module should count the number of metres per second and emit this number as 'speed' once per second.  
 These are exclusive
- ```

input Metre, Second; relation Metre & Second;
output Speed; integer;
loop
  var Distance := 0 ; integer in
  abort
  every Metre do
    Distance := Distance + 1;
  end every;
  when Second do
    emit Speed (Distance);
  end when;
end loop;
end module;
    
```
- ↳ Metre is always active when 'Metre' occurs
 ↳ activates block one last time with 'Second'
 ↳ even if a 'Metre' signal occurred simultaneously, it will not be lost

Introduction & Languages

Esterel

A simple, reactive, simultaneous signals integrator

- Metre and 'Second' occur potentially simultaneously
 ↳ using weak abort to handle the simultaneous case:
- ```

input Metre, Second;
output Speed; integer;
loop
 var Distance := 0 ; integer in
 weak abort
 every Metre do
 Distance := Distance + 1;
 end every;
 when Second do
 emit Speed (Distance);
 end when;
end loop;
end module;

```

## Introduction & Languages

Esterel

### Causality in Synchronous Languages

- ↳ Cyclic dependencies can cause causality problems in synchronous languages (similar to potential dead-locks in asynchronous languages).
- ↳ Strict synchronous languages avoid all cyclic dependencies in signals.  
 ↳ Esterel: fully-acyclic programs are considered too restrictive, because cyclic dependencies can make programs more intuitive/simpler.  
 ↳ Cyclic programs can be reactive and deterministic.

## Introduction & Languages

Esterel

### Esterel: Strong synchrony or "zero delay" assumption

- In logical terms:  
 ↳ All operations are instantaneous!
- In physical terms:  
 ↳ There is no observable delay between stimuli and reaction!

## Introduction & Languages

Esterel

### Immediate Reactions

- ↳ by default all synchronization points:  

```

await <signal>;
[statements];
when <signal> do ... end every;
loop ... each <signal>;
wait for the next signal occurrence (using edge trigger).
... yet with an additional 'limited state':
await immediate <signal>;
abort ... when immediate <signal>;
loop ... each immediate <signal>;
end every;

```

 a currently active signal will trigger these statements ('level trigger').

## Introduction & Languages

Esterel

### A simple, reactive, simultaneous signals integrator

- Metre and 'Second' occur potentially simultaneously  
 ↳ using every, immediate, to handle the simultaneous case:
- ```

input Metre, Second;
output Speed; integer;
loop
  var Distance := 0 ; integer in
  every immediate Metre do
    Distance := Distance + 1;
  end every;
  when Second do
    emit Speed (Distance);
  end when;
end loop;
end module;
    
```
- ↳ These are not exclusive

Introduction & Languages

Esterel

Strong synchrony or "zero delay" assumption

- ↳ The system is assumed 'synchronous' if
 ↳ the total worst case computation time is smaller than the minimal time between two observable changes in the environment.
- ↳ Synchronous systems assume a logic of discrete rather than continuous time.
- Enables:
 ↳ Strong analysis and simplification tools.
 ↳ Significantly easier program verification.
 ↳ Straight forward hardware implementations.
-

Introduction & Languages

Esterel

Esterel: Strong synchrony or "zero delay" assumption

- In logical terms:
 ↳ All operations are instantaneous!

Introduction & Languages

Esterel

Esterel: Strong synchrony or "zero delay" assumption

- In logical terms:
 ↳ All operations are instantaneous!

Introduction & Languages

Esterel

Esterel: Strong synchrony or "zero delay" assumption

- In logical terms:
 ↳ All operations are instantaneous!

Introduction & Languages

Esterel

Esterel: Strong synchrony or "zero delay" assumption

- In logical terms:
 ↳ All operations are instantaneous!

Introduction & Languages

Esterel

Esterel: Strong synchrony or "zero delay" assumption

- In logical terms:
 ↳ All operations are instantaneous!
- In physical terms:
 ↳ There is no observable delay between stimuli and reaction!
- In computer science terms:
 ↳ All operations are finished before the next input sampling.
 i.e. ↳ The base frequency is high enough such that the sampling can be considered instantaneous with respect to the physical system.

Introduction & Languages

Esterel

Esterel: Strong synchrony or "zero delay" assumption

- In logical terms:
 ↳ All operations are instantaneous!
- In physical terms:
 ↳ There is no observable delay between stimuli and reaction!
- In computer science terms:
 ↳ All operations are finished before the next input sampling.
 i.e. ↳ The base frequency is high enough such that the sampling can be considered instantaneous with respect to the physical system.

Introduction & Languages

Esterel

Esterel: Strong synchrony or "zero delay" assumption

- In logical terms:
 ↳ All operations are instantaneous!
- In physical terms:
 ↳ There is no observable delay between stimuli and reaction!
- In computer science terms:
 ↳ All operations are finished before the next input sampling.
 i.e. ↳ The base frequency is high enough such that the sampling can be considered instantaneous with respect to the physical system.

Introduction & Languages

Esterel

Esterel: Strong synchrony or "zero delay" assumption

- In logical terms:
 ↳ All operations are instantaneous!
- In physical terms:
 ↳ There is no observable delay between stimuli and reaction!
- In computer science terms:
 ↳ All operations are finished before the next input sampling.
 i.e. ↳ The base frequency is high enough such that the sampling can be considered instantaneous with respect to the physical system.

Introduction & Languages

Assembler level programming

Macro-Assemblers

- ☞ Closest to hardware.
- ☞ Predictable results (as predictable as the underlying hardware).
- ☞ Small footprint.
- ☞ As sequential or concurrent as the underlying hardware.
- No abstraction or support for large systems.
- Basic types are defined by the deployed processor (similar to C).
- Hard to read.
- ☞ Used mostly in very small applications or short code sequences.

© 2011 Frank E. Barto, Inc. All rights reserved. ISBN: 978-1-449-99999-9, pp. 149-225

Introduction & Languages

Cross-over between assembler and higher level languages

C

Combines the main disadvantages of assemblers and higher programming languages:

- ✦ Does not offer the abstractions of a higher programming language.
- ✦ Cannot take direct advantage of hardware-specific features.

Yet:

- ☞ Extremely popular.
- ☞ Simple and fast parameter passing (without compiler optimizations).
- ☞ Small footprint (zero runtime environment).
- ☞ Simple to write compilers (basically, a macro-assembler).
- ☞ Available for virtually any processor.

© 2011 Frank E. Barto, Inc. All rights reserved. ISBN: 978-1-449-99999-9, pp. 149-225

Introduction & Languages

Real-Time Programming Languages

Languages mentioned so far

- Ada (ADA2012) ☞ General purpose.
- Real-Time Java (Real-Time Specification for Java 1.1) ☞ (Very) soft real-time applications.
- Esterel (Esterel V7) ☞ An alternative for high-integrity applications.
- VHDL ☞ Compile real-time data flows and independent, asynchronous control paths into hardware.
- Timed CSP (as used and developed since 1986) ☞ An algebraic approach.
- PEARL (PEARL-90) ☞ A traditional language, specialized on plan modelling.
- POSIX (POSIX.1b, ...) ☞ The libraries of bare bone integers and semaphores.
- Assemblers / C ☞ The languages of bare bone words.

© 2011 Frank E. Barto, Inc. All rights reserved. ISBN: 978-1-449-99999-9, pp. 149-225

Introduction & Languages

Summary

Introduction & Real-Time Languages

- **Features** (and non-features) of a real-time system
 - Features, definitions, scenarios, and characteristics.
- **Components** of a real-time system
 - Converters, interfaces, sensors, actuators, communication systems, controllers, ...
- **Software layers** of a real-time system
 - Algorithms, operating systems, protocols, languages, concurrent and distributed systems.
- **Real-time languages criteria**
 - Mostly high-integrity, predictable languages with means for explicit time scopes.
- **Examples of actual real-time languages**

© 2011 Frank E. Barto, Inc. All rights reserved. ISBN: 978-1-449-99999-9, pp. 149-225