

Real-Time & Embedded Systems 2019



Introduction & Languages

Uwe R. Zimmer - The Australian National University

© 2019 Uwe R. Zimmer, The Australian National University

Page 22 of 96 | Chapter 1: "Introduction & Languages", up to page 2/20

Introduction & Languages

References

| | | |
|-------------|--|---|
| [Berry2000] | Berry, Gerard <i>The Esterel v5 Language Primer</i> -Version 5.5.91 2000 pp. 1-148 | Theoretical Computer Science 1995 vol. 138 (2) pp. 243-271 |
| [Lee2009] | Lee, Edward <i>Computing Needs Time</i> Technical Report No. UCB/ECCS-2009-30-2009 | [ISO/IEC 8652:1995(E)] Society, Design Automation Standards Committee of the IEEE Computer Society Std 1076-2008 Re-vision of IEEE Std 1076-2002 |
| [NN1995] | Burns, Alan & Wellings, Andy <i>Concurrent and Real-Time Programming in Ada</i> , edition Cambridge University Press (2007) | [NN1995] Occam 2.1 reference manual 1995 pp. 1-171 |
| [NN2004] | Davies, I. & Schneider, S <i>A brief history of timed CSP</i> © 2004 Iain Davies, The Australian National University | [NN2004] The Real-Time Java Platform Walli, Stephen The POSIX family of standards of standards [Walli1995] June 2004 2004 pp. 1-26 |
| [NN2006] | [NIN1995] | [NIN2006] |

Ada Reference Manual, Language and Standard Libraries
Ada Reference Manual
ISO/IEC 8652:1995(E)
IEEE Std 1076-2008 Re-vision of IEEE Std 1076-2002
IEEE Standard VHDL Language Reference Manual
2009 pp. 1-640

page 22 of 96 | Chapter 1: "Introduction & Languages", up to page 2/20

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 responds 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’

© 2019 Uwe R. Zimmer, The Australian National University

Page 23 of 96 | Chapter 1: "Introduction & Languages", up to page 2/20

Introduction & Languages

What is a real-time system?

Features of a Real-Time System?

- Adherence to set time constraints
- Predictability
- Fault tolerance
- Accuracy
- Frequently concurrent, distributed and employing complex hardware.

Not too early – not too late

Repetable results in time and value

Robustness in the presence of foreseeable faults

Results are precise enough to drive e.g. a physical systems

© 2019 Uwe R. Zimmer, The Australian National University

Page 23 of 96 | Chapter 1: "Introduction & Languages", up to page 2/20

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 station(s) ...
- 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.

u Real-Time Systems require a specific understanding and skill set.

© 2019 Uwe R. Zimmer, The Australian National University

Page 24 of 96 | Chapter 1: "Introduction & Languages", up to page 2/20

Introduction & Languages

Simple example: Brake manager

Ways to define reliability

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

© 2019 Uwe R. Zimmer, The Australian National University

Page 25 of 96 | Chapter 1: "Introduction & Languages", up to page 2/20

Introduction & Languages

Simple example: Brake manager

Latencies:

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

Reliability:

- Provide robustness under all foreseeable and “manageable” failures.

Efficient design:

- Mass producible?

© 2019 Uwe R. Zimmer, The Australian National University

Page 26 of 96 | Chapter 1: "Introduction & Languages", up to page 2/20

Introduction & Languages

Simple example: Brake manager

- Mechanical (hydraulic) + overwrite valves
- Digitally controlled, (no mechanical connection)
- Brake lights

The controller(s):

- Single CPU
- Multiple CPUs + shared memory
- Multiple CPUs + point-to-point connections
- Multiple CPUs + communication system (e.g. a bus system)
- Redundant CPUs

Ways to implement this (hardware)

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

© 2019 Univ. K. Zinner, The Austrian National University
Page 20 of 160 | Chapter 1: Introduction & Languages | up to Page 2/20

Introduction & Languages

Simple example: Brake manager

- Mechanical (hydraulic) + overwrite valves
- Digitally controlled, (no mechanical connection)
- Brake lights

The controller(s):

- Single CPU
- Multiple CPUs + shared memory
- Multiple CPUs + point-to-point connections
- Multiple CPUs + communication system (e.g. a bus system)
- Redundant CPUs

Ways to implement this (hardware)

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

© 2019 Univ. K. Zinner, The Austrian National University
Page 20 of 160 | Chapter 1: Introduction & Languages | up to Page 2/20

Introduction & Languages

Real-Time Systems Components

© 2019 Univ. K. Zinner, The Austrian National University
Page 21 of 160 | Chapter 1: Introduction & Languages | up to Page 2/20

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**
 - Complete, unambiguous language definition.
 - Granularity, environments detecting faults as early as possible.
- Concurrency and Distribution**
 - Solid, high-level synchronization and communication primitives, automated data marshalling.
- Specific yet Scalable**
 - Mapping physical interfaces into high-level data-types and programming "in the very large".

© 2019 Univ. K. Zinner, The Austrian National University
Page 22 of 160 | Chapter 1: Introduction & Languages | up to Page 2/20

Introduction & Languages

Typical Real-Time Operating System

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

RTOS provide:

- Predictability**: User tasks
- Passivity**: OS tasks
- Small footprint**: Real-time OS
- Instrumentation**: Hardware

"Standard" OS

- Typical**: User tasks
- Real-Time OS**: OS
- Embedded system**: Hardware

© 2019 Univ. K. Zinner, The Austrian National University
Page 23 of 160 | Chapter 1: Introduction & Languages | up to Page 2/20

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) ↔ (Agent-Oriented) ↔ (Agent-Oriented)**
- Determinism: Deterministic ↔ Non-deterministic**

© 2019 Univ. K. Zinner, The Austrian National University
Page 24 of 160 | Chapter 1: Introduction & Languages | up to Page 2/20

Introduction & Languages

Real-Time Languages, Operating Systems and Libraries

What if you "cannot/want not" use a real-time language and you need to formulate some/all real-time constraints outside the programming language?

Real-time operating systems:

- Scheduling, interrupt handling, (potentially other features) migrate 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.

© 2019 Univ. K. Zinner, The Austrian National University
Page 25 of 160 | Chapter 1: Introduction & Languages | up to Page 2/20

Introduction & Languages

Real-Time Programming Languages

Some RT-Languages

- Ada (Adl2012) \rightarrow General workhorse.
- Real-Time Java (Real-Time Specification for Java 1.1) \rightarrow Soft real-time applications.
- Esterel (Esterel v7) \rightarrow An alternative for high-integrity applications.
- VHDL \rightarrow Compile real-time data flows and independent, asynchronous control paths direct to hardware.
- Timed CSP (as used and developed since 1986) \rightarrow An algebraic approach.
- PEARL (PEARL90) \rightarrow A traditional language specialized on plant modelling.
- POSIX (POSIX 1003.1b,...) \rightarrow The libraries of bare bone integers and semaphores.
- Assemblers / C \rightarrow The languages of bare bone words.

© 2019 Univ R. Zimmer, The Australian National University
Page 37 of 96 | Chapter 1: Introduction & Languages | up to page 230

Languages explicitly supporting concurrency: e.g. Ada

Ada is an **ISO standardized** (ISO/IEC 8652:2013(E)) 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 (specification and implementation), 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 real-time features, distributed programming, system-level programming, numeric, informations systems, safety and security issues.

© 2019 Univ R. Zimmer, The Australian National University
Page 37 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 37 of 96 | Chapter 1: Introduction & Languages | up to page 230

Introduction & Languages

Data structure example

Queues

Forms of implementation:

The diagram illustrates three common ways to implement a queue:

- Dequeue:** A linear list where elements are added at the end ('In') and removed from the front ('Out').
- Enqueue:** A linear list where elements are added at the front ('In') and removed from the end ('Out').
- Ring lists:** A circular linked list where both ends ('In' and 'Out') are at the same position, allowing efficient enqueue and dequeue operations.

© 2019 Univ R. Zimmer, The Australian National University
Page 40 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 40 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 40 of 96 | Chapter 1: Introduction & Languages | up to page 230

Introduction & Languages

Data structure example

Queues

Forms of implementation:

The diagram illustrates three common ways to implement a queue:

- Dequeue:** A linear list where elements are added at the end ('In') and removed from the front ('Out').
- Enqueue:** A linear list where elements are added at the front ('In') and removed from the end ('Out').
- Ring lists:** A circular linked list where both ends ('In' and 'Out') are at the same position, allowing efficient enqueue and dequeue operations.

© 2019 Univ R. Zimmer, The Australian National University
Page 40 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 40 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 40 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 40 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 40 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 40 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 40 of 96 | Chapter 1: Introduction & Languages | up to page 230

Introduction & Languages

A simple queue specification

package Queue_Pack_Simple is

```
  QueueSize : constant Positive := 10;
  type Element is new Positive range 1..1000..40_000;
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    IsEmpty : 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 IsEmpty (Queue : Queue_Type) return Boolean;
  function IsFull (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;
```

© 2019 Univ R. Zimmer, The Australian National University
Page 44 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 44 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 44 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 44 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 44 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 44 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 44 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 44 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 44 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 44 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 44 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 44 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 44 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 45 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 45 of 96 | Chapter 1: Introduction & Languages | up to page 230

Page 45 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 45 of 96 | Chapter 1: Introduction & Languages | up to page 230

© 2019 Univ R. Zimmer, The Australian National University
Page 45 of 96 | Chapter 1: Introduction & Languages | up to page 230

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;
    Queue.Free := False;
end Enqueue;
procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    Queue.Top = Queue.Free;
    end Dequeue;
    function Is_Empty : 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;
```

© 2019 Law & Zonneveld, The Australian National University

Page 56 of 96 | Introduction & Languages | up to Page 200

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;
begin
    Enqueue (2000, Queue);
    Dequeue (Item, Queue);
    Dequeue (Item, Queue);
end Queue_Test_Simple;
```

© 2019 Law & Zonneveld, The Australian National University

Page 57 of 96 | Introduction & Languages | up to Page 200

Page 57 of 96 | Introduction & Languages | up to Page 200

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;
begin
    Enqueue (2000, Queue);
    Dequeue (Item, Queue);
    Dequeue (Item, Queue);
end Queue_Test_Simple;
```

© 2019 Law & Zonneveld, The Australian National University

Page 58 of 96 | Introduction & Languages | up to Page 200

... anything on this slide
still 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.Elements (Queue.Free) := Queue.Free + 1;
    Queue.Free := False;
end Enqueue;
procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    Queue.Top = Queue.Free;
    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;
```

© 2019 Law & Zonneveld, The Australian National University

Page 57 of 96 | Introduction & Languages | up to Page 200

Introduction & Languages

A simple queue implementation

```
with Queue_Pack_Simple; use Queue_Pack_Simple;
procedure Queue_Test_Simple is
    Queue : Queue_Type;
    Item : Element;
begin
    Enqueue (2000, Queue);
    Dequeue (Item, Queue);
    Dequeue (Item, Queue);
end Queue_Test_Simple;
```

© 2019 Law & Zonneveld, The Australian National University

Page 57 of 96 | Introduction & Languages | up to Page 200

Page 57 of 96 | Introduction & Languages | up to Page 200

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;
begin
    Enqueue (2000, Queue);
    Dequeue (Item, Queue);
    Dequeue (Item, Queue);
end Queue_Test_Simple;
```

© 2019 Law & Zonneveld, The Australian National University

Page 58 of 96 | Introduction & Languages | up to Page 200

... anything on this slide
still 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.Elements (Queue.Free) := Queue.Free + 1;
    Queue.Free := False;
end Enqueue;
procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    Queue.Top = Queue.Free;
    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;
```

© 2019 Law & Zonneveld, The Australian National University

Page 57 of 96 | Introduction & Languages | up to Page 200

Introduction & Languages

A simple queue implementation

```
with Queue_Pack_Simple; use Queue_Pack_Simple;
procedure Queue_Test_Simple is
    Queue : Queue_Type;
    Item : Element;
begin
    Enqueue (2000, Queue);
    Dequeue (Item, Queue);
    Dequeue (Item, Queue);
end Queue_Test_Simple;
```

© 2019 Law & Zonneveld, The Australian National University

Page 57 of 96 | Introduction & Languages | up to Page 200

Page 57 of 96 | Introduction & Languages | up to Page 200

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;
begin
    Enqueue (2000, Queue);
    Dequeue (Item, Queue);
    Dequeue (Item, Queue);
end Queue_Test_Simple;
```

© 2019 Law & Zonneveld, The Australian National University

Page 58 of 96 | Introduction & Languages | up to Page 200

... anything on this slide
still not perfectly clear

Introduction & Languages

Ada Exceptions

... introducing:

- Exception handling
- Enumeration types
- Type attributed operators

• Type attributed operators

```
package Queue_Pack_Exceptions is
  QueueSize : constant Positive := 10;
  type Element is (Up, Down, Spin, Turn);
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    IsEmpty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue : in out Queue_Type);
  function IsEmpty (Queue : Queue_Type) return Boolean is (Queue.Is.Empty);
  function IsFull (Queue : Queue_Type) return Boolean is (not Queue.Is.Empty and then Queue.Top = Queue.Free);
  Queue_overflow, Queue_underflow : exception;
end Queue_Pack_Exceptions;
```

© 2019 Lutz Kettner: The Austrian National University

Page 65 of 96 | Introduction & Languages | up to Page 230

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 mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    IsEmpty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue : in out Queue_Type);
  function IsEmpty (Queue : Queue_Type) return Boolean is (Queue.Is.Empty);
  function IsFull (Queue : Queue_Type) return Boolean is (not Queue.Is.Empty and then Queue.Top = Queue.Free);
  Queue_overflow, Queue_underflow : exception;
end Queue_Pack_Exceptions;
```

© 2019 Lutz Kettner: The Austrian National University

Page 65 of 96 | Introduction & Languages | up to Page 230

Introduction & Languages

A queue specification 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);
    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;
    Queue.Elements (Queue.Top) := Item;
    Item := Marker'Succ (Queue.Top);
    Queue.Top := Queue.Top - 1;
    Queue.Is.Empty := Queue.Top = Queue.Free;
    end Dequeue;
end Queue_Pack_Exceptions;
```

© 2019 Lutz Kettner: The Austrian National University

Page 65 of 96 | Introduction & Languages | up to Page 230

Enumeration types are first-class types and can be used e.g. as array indices.

The representation values can be controlled and do not need to be continuous (e.g. for purposes like interfacing with hardware).

...

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

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

</

Introduction & Languages

queue specification with proper information hiding

```

private 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 IsEmpty (Queue : Queue_Type) return Boolean;
    function IsFull (Queue : Queue_Type) return Boolean;
    QueueOverflow, QueueUnderflow : exception;
private
    type Marker is mod QueueSize;
    type List is array (Marker) of Element;
    type queue_Type is record
        Top : Free : Marker := Marker'First;
        Elements : List;
        end record;
end record;
end Queue.Pack.Private;

```

Page 10 / 10 | Information & Languages - up to Chap 10

A queue implementation with proper information hiding

```

package body Queue_Pack_Private is
  procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
    begin
      if Queue.State = Filled and Queue.Top = Queue.Free then
        raise QueueOverflow;
      end if;
      Queue.Elements (Queue.Free) := Item;
      Queue.Free := Marker'Value (Queue.Free);
      Queue.Is_Empty := False;
      end Enqueue;

  procedure Dequeue (Value : out Element; Queue : in out Queue_Type) is
    begin
      if Queue.State = Empty then
        raise QueueUnderflow;
      end if;
      Value := Queue.Elements (Queue.Top);
      Queue.Top := Marker'Value (Queue.Top);
      Queue.Is_Empty := True;
      end Dequeue;

  function Is_Empty (Queue : Queue_Type) return Boolean is
    begin
      if Queue.State = Empty then
        return True;
      else
        return False;
      end if;
    end Is_Empty;
end Queue_Pack_Private;
  
```

Copyright © 2007, 2010 by David J. Barnes, The Australian National University

```

with Queue_Pack_Private; use Queue_Pack_Private;
with Ada.Text_IO      ; use Ada.Text_IO;
procedure Queue_Test_Private is
   Queue, Queue_Copy : Queue_Type;
   Item              : Element_Type;
begin
   Queue_Copy := Queue;
   -- compiler-error: "left hand of assignment must not be limited type"
   Enqueue(Item => 1, Queue => Queue); -- 
   Dequeue(Item, Queue); -- would produce a 'Queue underFlow'
   Dequeue(Item, Queue); -- would produce a 'Queue underFlow'
exception
   when Queueunderflow => Put("Queue underFlow");
   when Queueoverflow  => Put("Queue overflow");
end Queue_Test_Private;

```

Parameters can be passed by value or by reference
(Named parameters do not follow the definition)

Diagram of Ada's Queue package from the Adafruit National University

A queue **implementation** with proper *information hiding*

卷之三

```

procedure Queue_Pack_Private is
begin
  if Queue_State = Filled and Queue.Top = Queue.Free then
    raise QueueOverflow;
  end if;
  Queue.Elements(Queue.Free) := Item;
  Queue.Free := Marker and queue_free;
  Queue.IsEmpty := False;
end Queue_Pack_Private;

procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
begin
  if Queue.IsEmpty then
    item := queueunderflow;
    Queueunderflow := queueunderflow;
  else
    Item := Queue.Elements(Queue.Top);
    Queue.Top := marker_Pred(Queue.Top);
    Queue.IsEmpty := Queue.Top = Queue.Free;
  end if;
end Dequeue;

function IsEmpty (Queue : Queue_Type) return Boolean is
begin
  return Queue.Top = Queue.Free;
end IsEmpty;

```

Page 15 of 94 | [Download](#)

© 2014 K. Zaman, The Australian National University

Introduction & Languages

A queue test program with proper information hiding

```
with Queue_Pack_Private; use Queue_Pack_Private;
with Ada.Text_IO      ; use Ada.Text_IO;
procedure Queue_Test_Private is
    Queue_Copy : Queue_Type;
    Item      : Element;
begin
    Queue_Copy := Queue;
    -- compiler error: "left hand of assignment must not be limited type"
    Enqueue(Item => 1, Queue => Queue);
    Dequeue(Item, Queue);
    Dequeue(Item, Queue); -- would produce a "Queue underflow"
exception
    when QueueUnderflow => Put("Queue underflow");
    when QueueOverflow  => Put("Queue overflow");
end Queue_Test_Private;
```

Page 86 (of 96) | Introduction & Languages | Copyright © 2010, Irine R. Zlotnick, The Australian National University

```

with Queue_Pack_Private; use Queue_Pack_Private;
with Ada.Text_IO           ; use Ada.Text_IO;
procedure Queue_Test_Private is
    Queue, Queue.Copy : Queue_Type;
    Item              : Element;
begin
    Queue.Copy := Queue;
    -- compiler-error: "left hand of assignment must not be limited type"
    Enqueue (Item => 1, Queue => Queue);
    Dequeue (Item, Queue); -- would produce a "Queue underflow";
    Dequeue (Item, Queue); -- would produce a "Queue overflow";
exception
    when QueueUnderflow  => Put ("Queue underflow");
    when QueueOverflow   => Put ("Queue overflow");
end Queue_Test_Private;

```

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

Based on the work of Bertrand Lemoine, The Australian National University

A queue **implementation** with proper **information hiding**

卷之三

```

partial body Queue::Pack_Private is
procedure Enqueue (Item : Element; Queue : in out Queue_Type);
begin
  if Queue.State = Filled and Queue.Top = Queue.Free then
    raise QueueOverflow;
  end if;
  Elements (Queue.Free) := Item;
  Queue.Free := Marker'Last;
  Queue.Is_Empty := False;
  end Enqueue;
procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
begin
  if Queue.State = Empy then
    raise QueueUnderflow;
  end if;
  Item := Queue.Elements (Queue.Top);
  Queue.Top := Marker'Pred (Queue.Top);
  Queue.Top := Queue.Top;
  end Dequeue;
function Is_Empty (Queue : Queue_Type) return Boolean is (Queue
  function Is_Full (Queue : Queue_Type) return Boolean is (Queue
    not Queue.Is_Empty and then Queue.Top = Queue.Free);
end Queue_Pack_Private;

```

Copyright © 2010 by Pearson Education, Inc.

Introduction & Languages

A queue test program with proper information hiding

```
with Queue_Pack_Private; use Queue_Pack_Private;  
with Ada.Text_Io ; use Ada.Text_Io;  
procedure Queue_Test_Private is  
    Queue : Queue_Copy : Queue_Type;  
    Item : Element; begin  
    Queue_Copy := Queue; -- compiler-error: "left hand of assignment must note be limited type"  
    Enqueue (Item => 1, Queue => Queue);  
    Dequeue (Item, Queue);  
    Dequeue (Item, Queue); -- would produce a "Queue underflow"  
    when Queueunderflow => Put ("Queue underflow");  
    when Queueoverflow => Put ("Queue overflow");  
    end Queue_Test_Private;
```

page 10 of 100 | Introduction & Languages

diau 300 - Modellierung & Formalisierung
© 2011 Hans-Peter Ziegert, The Australian National University

Introduction & Languages

A contracting queue specification

```

package Queue_Pack_Contract is
  Queue_Size : constant Positive := 10;
  type Element is new Positive range 1 .. 1000;
  type Queue_Type is private;
  procedure Enqueue (Item : Element; Q : in out Queue_Type) with
    Pre => not Is_Full (Q),
    Post => not Is_Empty (Q) and then Length (Q) = Length (Q'Old) + 1
      and then Lookahead (Q, Length (Q)) = Item;
  procedure Dequeue (Item : Element; Q : in out Queue_Type) with
    Pre => not Is_Empty (Q),
    Post => not Is_Full (Q) and then Length (Q) = Length (Q'Old) - 1
      and then (for all ix in 1 .. Length (Q) - 1)
        Lookahead (Q, ix) = Lookahead (Q'Old, ix);
  function Is_Empty (Q : Queue_Type) return Boolean;
  function Is_Full (Q : Queue_Type) return Boolean;
  function Length (Q : Queue_Type) return Natural;
  function Lookahead (Q : Queue_Type; Depth : Positive) return Element;
  
```

© 2019 K. Ziviani, The Australian National University
Page 91 of 94 | Introduction & Languages | up to page 230

Introduction & Languages

A contracting queue specification

```

package Queue_Pack_Contract is
  Queue_Size : constant Positive := 10;
  type Element is new Positive range 1 .. 1000;
  type Queue_Type is private;
  procedure Enqueue (Item : Element; Q : in out Queue_Type) with
    Original
    Pre values
    can still be
    referred to.
    Post => not Is_Full (Q) and then Length (Q) = Length (Q'Old) + 1
      and then Lookahead (Q, Length (Q)) = Item;
  procedure Dequeue (Item : out Element; Q : in out Queue_Type) with
    Pre => not Is_Empty (Q),
    Post => not Is_Full (Q) and then Length (Q) = Length (Q'Old) - 1
      and then (for all ix in 1 .. Length (Q) - 1)
        Lookahead (Q, ix) = Lookahead (Q'Old, ix);
  function Is_Empty (Q : Queue_Type) return Boolean;
  function Is_Full (Q : Queue_Type) return Boolean;
  function Length (Q : Queue_Type) return Natural;
  function Lookahead (Q : Queue_Type; Depth : Positive) return Element;
  
```

© 2019 K. Ziviani, The Australian National University
Page 91 of 94 | Introduction & Languages | up to page 230

Introduction & Languages

A contracting queue specification (cont.)

```

private
  type Marker is mod Queue_Size;
  type List is array (Marker) of Element;
  type Queue_Type is record
    type List is array (Marker) of Element;
    type Queue_Type is record
      type Queue_Type is record
        Top : Free : Marker := Marker_First;
        Is_Free : Boolean := True;
        Elements : List; -- will be initialized to invalids
      end record with Type_Invariant
      => (not Queue_Type.Is_Free or else Queue_Type.Top = Queue_Type.Free)
        and then (for all ix in 1 .. Length (Queue_Type))
          Lookahead (Queue_Type, ix)'Valid;
    end record with Type_Invariant
    => (not Queue_Type.Top = Queue_Type.Free)
      and then (for all ix in 1 .. Length (Queue_Type))
        Lookahead (Queue_Type, ix)'Valid;
  end record with Type_Invariant
  => (not Queue_Type.Top = Queue_Type.Free)
    and then Natural is
      (if Is_Full (Q) then Queue_Size else Natural (Q'Free - Q.Top));
    function Lookahead (Q : Queue_Type; Depth : Positive) return Element is
      (Q.Elements (Q.Top + Marker (Depth - 1)));
  end Queue_Pack_Contract;
  
```

© 2019 K. Ziviani, The Australian National University
Page 92 of 94 | Introduction & Languages | up to page 230

Introduction & Languages

A contracting queue specification (cont.)

```

private
  type Marker is mod Queue_Size;
  type List is array (Marker) of Element;
  type Queue_Type is record
    type Queue_Type is record
      type Queue_Type is record
        Top : Free : Marker := Marker_First;
        Is_Free : Boolean := True;
        Elements : List; -- will be initialized to invalids
      end record with Type_Invariant
      => (not Queue_Type.Is_Free or else Queue_Type.Top = Queue_Type.Free)
        and then (for all ix in 1 .. Length (Queue_Type))
          Lookahead (Queue_Type, ix)'Valid;
    end record with Type_Invariant
    => (not Queue_Type.Top = Queue_Type.Free)
      and then Natural is
        (if Is_Full (Q) then Queue_Size else Natural (Q'Free - Q.Top));
      function Lookahead (Q : Queue_Type; Depth : Positive) return Element is
        (Q.Elements (Q.Top + Marker (Depth - 1)));
    end Queue_Pack_Contract;
  end Queue_Pack_Contract;
  
```

© 2019 K. Ziviani, The Australian National University
Page 93 of 94 | Introduction & Languages | up to page 230

Introduction & Languages

A contracting queue specification (cont.)

```

private
  type Marker is mod Queue_Size;
  type List is array (Marker) of Element;
  type Queue_Type is record
    type Queue_Type is record
      type Queue_Type is record
        Top : Free : Marker := Marker_First;
        Is_Free : Boolean := True;
        Elements : List; -- will be initialized to invalids
      end record with Type_Invariant
      => (not Queue_Type.Is_Free or else Queue_Type.Top = Queue_Type.Free)
        and then Natural is
        (if Is_Full (Q) then Queue_Size else Natural (Q'Free - Q.Top));
      function Lookahead (Q : Queue_Type; Depth : Positive) return Element is
        (Q.Elements (Q.Top + Marker (Depth - 1)));
    end Queue_Pack_Contract;
    
```

© 2019 K. Ziviani, The Australian National University
Page 94 of 94 | Introduction & Languages | up to page 230

Introduction & Languages

A contracting queue implementation

```

package body Queue_Pack_Contract is
  procedure Enqueue (Item : Element; Q : in out Queue_Type) is
  begin
    Q.Free := Item;
    Q := Q'Free + 1;
    if Q'Free = Q.Top then
      Q.Top := Q'Free;
    end Enqueue;
  end Queue_Pack_Contract;
  
```

No checks in the implementation part, as all required conditions have been guaranteed via the specifications.

© 2019 K. Ziviani, The Australian National University
Page 95 of 94 | Introduction & Languages | up to page 230

Introduction & Languages

A contracting queue test program

```

with Ada.Text_IO;
use Ada.Text_IO;
with Exceptions;
use Exceptions;
with Queue_Pack_Contract;
with System.Assertions;
procedure Queue_Test_Contract is
  Item : Element;
  Queue : Queue_Type;
begin
  Enqueue (Item => 1, Q => Queue);
  Enqueue (Item => 2, Q => Queue);
  Dequeue (Item, Queue); Put (Element'Image (Item));
  Dequeue (Item, Queue); Put (Element'Image (Item));
  Put (Element'Image (Item));
  Put (Queue is empty on exit: ''); Put (Boolean'Image (Is_Empty (Queue)));
  exception
    when Exception_Id : Assert_Failure => Show_Exception (Exception_Id);
  end Queue_Test_Contract;
  
```

Violated Pre-condition will raise an assert failure exception.

© 2019 K. Ziviani, The Australian National University
Page 96 of 94 | Introduction & Languages | up to page 230

Introduction & Languages

A contracting queue specification (cont.)

```

private
  type Marker is mod Queue_Size;
  type List is array (Marker) of Element;
  type Queue_Type is record
    type Queue_Type is record
      type Queue_Type is record
        Top : Free : Marker := Marker_First;
        Is_Free : Boolean := True;
        Elements : List; -- will be initialized to invalids
      end record with Type_Invariant
      => (not Queue_Type.Is_Free or else Queue_Type.Top = Queue_Type.Free)
        and then (for all ix in 1 .. Length (Queue_Type))
          Lookahead (Queue_Type, ix)'Valid;
    end record with Type_Invariant
    => (not Queue_Type.Top = Queue_Type.Free)
      and then Natural is
        (if Is_Full (Q) then Queue_Size else Natural (Q'Free - Q.Top));
      function Lookahead (Q : Queue_Type; Depth : Positive) return Element is
        (Q.Elements (Q.Top + Marker (Depth - 1)));
    end Queue_Pack_Contract;
  end Queue_Pack_Contract;
  
```

© 2019 K. Ziviani, The Australian National University
Page 97 of 94 | Introduction & Languages | up to page 230

Introduction & Languages

A contracting queue specification

```

package Queue_Pack_Contract is
  Queue_Size : constant Positive := 10;
  type Element is new Positive range 1 .. 1000;
  type Queue_Type is private;
  procedure Enqueue (Item : Element; Q : in out Queue_Type) with
    Pre => not Is_Full (Q),
    Post => not Is_Empty (Q) and then Length (Q) = Length (Q'Old) + 1
      and then Lookahead (Q, Length (Q)) = Item;
  procedure Dequeue (Item : out Element; Q : in out Queue_Type) with
    Pre => not Is_Empty (Q),
    Post => not Is_Full (Q) and then Length (Q) = Length (Q'Old) - 1
      and then (for all ix in 1 .. Length (Q) - 1)
        Lookahead (Q, ix) = Lookahead (Q'Old, ix);
  function Is_Empty (Q : Queue_Type) return Boolean;
  function Is_Full (Q : Queue_Type) return Boolean;
  function Length (Q : Queue_Type) return Natural;
  function Lookahead (Q : Queue_Type; Depth : Positive) return Element;
  
```

© 2019 K. Ziviani, The Australian National University
Page 98 of 94 | Introduction & Languages | up to page 230

Introduction & Languages

A contracting queue test program

```

with Ada.Text_IO;
use Ada.Text_IO;
with Exceptions;
use Exceptions;
with Queue_Pack_Contract;
with System.Assertions;
procedure Queue_Test_Contract is
  Item : Element;
begin
  Enqueue (Item => 1, Q => Queue);
  Enqueue (Item => 2, Q => Queue);
  Dequeue (Item, Queue); Put (Element'Image (Item));
  Dequeue (Item, Queue); Put (Element'Image (Item));
  Put (Element'Image (Item));
  Put (Queue is empty on exit: ''); Put (Boolean'Image (Is_Empty (Queue)));
  exception
    when Exception_Id : Assert_Failure => Show_Exception (Exception_Id);
  end Queue_Test_Contract;
  
```

Violated Pre-condition will raise an assert failure exception.

© 2019 K. Ziviani, The Australian National University
Page 99 of 94 | Introduction & Languages | up to page 230

Introduction & Languages

contracting queue test program

```

with Ada.Text_IO; use Ada.Text_IO;
with Exceptions; use Exceptions;
with Queue_Pack_Contract; use Queue_Pack_Contract;
with System.Assertions; use System.Assertions;

procedure Queue_Test_Contract is
    Queue : Queue_Type;
    Item   : Element;
begin
    Enqueue (Item => 1, Q => Queue);
    Enqueue (Item => 2, Q => Queue);
    Dequeue (Item, Queue); Put (Element'Image (Item));
    Dequeue (Item, Queue); Put (Element'Image (Item));
    Dequeue (Item, Queue); Put (Element'Image (Item));
    Put ("Queue is empty on exit: "); Put (Bool'Image (Is_Empty (Queue)));
    when Exception_Id : Assert_Failure => Show_Exception (Exception_Id);
end Queue_Test_Contract;

```

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

jag@csail.mit.edu (jag) 2008-12-23

© 2009 Lawrie C. Brown, The皇后大学, Waterloo, Ontario, Canada

103

Ada

| Post | => | not Is.Empty (Q) and then Length (Q) = length (Q' Old) + 1 and then Lookahead (Q, Length (Q)) = item | specifications |
|------|----|---|--|
| | | | ... introducing: |
| | | | <ul style="list-style-type: none"> • Specification of generic packages • Instantiation of generic packages |
| | | | <pre>procedure Dequeue (Item : out Element; Q : in out Queue_Type) with Pre => not Is.Empty (Q), -- could also be =>"True" according to specifications Post => not Is.Full (Q) and then Length (Q) = length (Q' Old) - 1 and then (for all ix in 1 .. Length (Q)) => Lookahead (Q, ix) = Lookahead (Q' Old, ix + 1);</pre> |
| | | | <p>(<i>c</i>)</p> <pre>type Queue_Type is record Top : Free := Marker_First; Elements : List; end record; type Type_Invariant is (not Queue_Type'Empty or else Queue_Type.Top = Queue_Type.Free)</pre> |
| | | | <p>Those contracts can be used to fully specify operations and types. Specifications should be complete, consistent and canonical, while using as little implementation details as possible.</p> |
| | | | <p style="text-align: right;">© 2015 Mark Weiser, Zentrum für Informatik, Universität Karlsruhe (TH)</p> |

103

A generic queue specification

```

generic
    type Element is private;
    package Queue_Pack_Generic is
        QueueSize: constant Integer := 10;
        type Queue 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;
        QueueOverflow, QueueUnderflow : exception;
    private
        type Marker is mod QueueSize;
        type List is array (Marker) of Element;
        type Queue is record
            Top, Free : Marker := Marker'First;
            IsEmpty : Boolean := True;
            Elements : List;
        end record;
    end Queue_Pack_Generic;

```

106

A generic queue specification

卷之三

Introduction & Languages

Ada

... introducing

- Specification of generic packages
- Instantiation of generic packages

A generic queue specification

```

generic
    type Element is private;
    package Queue_Pack_Generic is
        QueueSize: constant Integer := 10;
        type Queue_Type is limited private;
        procedure Enqueue (Item : out Element; Queue : Queue_Type);
        procedure Dequeue (Item : out Element; Queue : Queue_Type) return Boolean;
        function IsEmpty (Queue : Queue_Type) return Boolean;
        function IsFull (Queue : Queue_Type) return Boolean;
        QueueOverflowFlow : exception;
    private
        type Marker is mod QueueSize;
        type List is array (Marker) of Element;
        type Queue_Type is record
            Top : Free := Marker; First := True;
            IsEmpty : Boolean := True;
            Elements : List;
        end record;
    end Queue_Pack_Generic;

generic
    type Element is private;
    package Queue_Pack is
        QueueSize: constant Integer := 10;
        type Queue_Type is limited private;
        procedure Enqueue (Item : out Element; Queue : Queue_Type);
        procedure Dequeue (Item : out Element; Queue : Queue_Type) return Boolean;
        function IsEmpty (Queue : Queue_Type) return Boolean;
        function IsFull (Queue : Queue_Type) return Boolean;
        QueueOverflowFlow : exception;
    private
        type Marker is mod QueueSize;
        type List is array (Marker) of Element;
        type Queue_Type is record
            Top : Free := Marker; First := True;
            IsEmpty : Boolean := True;
            Elements : List;
        end record;
    end Queue_Pack;

```

A brief tour of Ada's Translation & Implementation Guide
© 2013 The Board of Trustees of The Australian National University

A generic queue test program

```

with Queue_Pack_Generic; -- cannot apply 'use' clause here
with Ada.Text_IO; use Ada.TextIO;
procedure Queue_Test_Generic is
    package Queue_Pack_Generic is
        new Queue_Pack_Positive; -- 'use' clause can be applied to instantiated package
    end Queue_Pack_Positive;
    Queue : Queue_Type;
    Item : Positive;
begin
    Enqueue (Item => 1, Queue => Queue);
    Dequeue (Item, Queue);
    Dequeue (Item, Queue); -- will produce a "Queue underflow"
exception
    when QueueUnderflow => Put ("queue underflow");
    when QueueOverflow => Put ("queue overflow");
end Queue_Test_Generic;

```

THE JOURNAL OF CLIMATE

Introduction & Languages

A generic queue test program

```

with Queue_Pack_Generic; -- cannot apply 'use' clause here
with Ada.Text_IO ; use Ada.Text_IO;
procedure Queue_Test_Generic is
  package Queue_Pack_Positive is
    new Queue_Pack_Generic (Element => Positive);
  use Queue_Pack_Positive; -- 'use' clause can be applied to instantiated package
  Queue : Queue_Type;
  Item : Positive;
begin
  Enqueue (Item => 1, Queue => Queue);
  Dequeue (Item, Queue); -- will produce a "Queue underflow"
  Dequeue (Item, Queue); -- will produce a "Queue overflow"
exception
  when Queueunderflow => Put ("Queue underflow");
  when Queueoverflow => Put ("Queue overflow");
end Queue_Test_Generic;

```

© 2019 Iain R.躡mer, The Australian National University [\[Page 1/20\]](#)

A generic protected queue specification

```

generic
  type Element is private;
  package Queue_Pack_Generic is
    constant Queue_Size : Integer := 10;
    type Queue_Type is limited private;
    procedure Enqueue (Item : in out Queue_Type; Element : 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;
  private
  QueueOverflow, QueueUnderflow : exception;
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    IsEmpty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Generic;

```

© 2019 Iain R.躡mer, The Australian National University [\[Page 2/20\]](#)

A generic protected queue specification

```

generic
  type Element is private;
  type Index is mod <>; -- 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 : Element);
      entry Dequeue (Item : out Element);
      procedure Empty_Queue;
      function Is_Empty return Boolean;
      function Is_Full return Boolean;
    private
      Queue : Queue_Type;
    end Protected_Queue;
  private
  type List is array (Index) of Element;
  type Queue_Type is record
    Top, Free : Index := Index'First;
    IsEmpty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Protected_Generic;

```

© 2019 Iain R.躡mer, The Australian National University [\[Page 3/20\]](#)

A generic protected queue specification

```

generic
  type Element is private;
  type Index is mod <>; -- 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 : Element);
      entry Dequeue (Item : out Element);
      procedure Empty_Queue;
      function Is_Empty return Boolean;
      function Is_Full return Boolean;
    private
      Queue : Queue_Type;
    end Protected_Queue;
  private
  type List is array (Index) of Element;
  type Queue_Type is record
    Top, Free : Index := Index'First;
    IsEmpty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Protected_Generic;

```

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

© 2019 Iain R.躡mer, The Australian National University [\[Page 4/20\]](#)

Introduction & Languages

A generic queue test program

```

with Queue_Pack_Generic; -- cannot apply 'use' clause here
with Ada.Text_IO ; use Ada.Text_IO;
procedure Queue_Test_Generic is
  Instantiate generic package
  new Queue_Pack_Positive (Element => Positive);
  use Queue_Pack_Positive; -- 'use' clause can be applied to instantiated package
  Queue : Queue_Type;
  Item : Positive;
begin
  Enqueue (Item => 1, Queue => Queue);
  Dequeue (Item, Queue); -- will produce a "Queue underflow"
  Dequeue (Item, Queue); -- will produce a "Queue overflow"
exception
  when Queueunderflow => Put ("Queue underflow");
  when Queueoverflow => Put ("Queue overflow");
end Queue_Test_Generic;

```

... anything on this slide
still not perfectly clear?
© 2019 Iain R.躡mer, The Australian National University [\[Page 1/20\]](#)

A generic protected queue specification

```

generic
  type Element is private;
  type Index is mod <>; -- 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 : Element);
      entry Dequeue (Item : out Element);
      procedure Empty_Queue;
      function Is_Empty return Boolean;
      function Is_Full return Boolean;
    private
      Queue : Queue_Type;
    end Protected_Queue;
  private
  type List is array (Index) of Element;
  type Queue_Type is record
    Top, Free : Index := Index'First;
    IsEmpty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Protected_Generic;

```

... anything on this slide
still not perfectly clear?
© 2019 Iain R.躡mer, The Australian National University [\[Page 2/20\]](#)

A generic protected queue specification

```

generic
  type Element is private;
  type Index is mod <>; -- 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 : Element);
      entry Dequeue (Item : out Element);
      procedure Empty_Queue;
      function Is_Empty return Boolean;
      function Is_Full return Boolean;
    private
      Queue : Queue_Type;
    end Protected_Queue;
  private
  type List is array (Index) of Element;
  type Queue_Type is record
    Top, Free : Index := Index'First;
    IsEmpty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Protected_Generic;

```

... anything on this slide
still not perfectly clear?
© 2019 Iain R.躡mer, The Australian National University [\[Page 3/20\]](#)

A generic protected queue specification

```

generic
  type Element is private;
  type Index is mod <>; -- 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 : Element);
      entry Dequeue (Item : out Element);
      procedure Empty_Queue;
      function Is_Empty return Boolean;
      function Is_Full return Boolean;
    private
      Queue : Queue_Type;
    end Protected_Queue;
  private
  type List is array (Index) of Element;
  type Queue_Type is record
    Top, Free : Index := Index'First;
    IsEmpty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Protected_Generic;

```

... anything on this slide
still not perfectly clear?
© 2019 Iain R.躡mer, The Australian National University [\[Page 4/20\]](#)

Introduction & Languages

A generic queue specification

```

generic
  type Element is private;
  package Queue_Pack_Generic is
    constant Queue_Size : Integer := 10;
    type Queue_Type is limited private;
    procedure Enqueue (Item : in out Queue_Type; Element : Queue_Type);
    procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
    function IsEmpty (Queue : Queue_Type) return Boolean;
    function IsFull (Queue : Queue_Type) return Boolean;
  private
  QueueOverflow, QueueUnderflow : exception;
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    IsEmpty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Generic;

```

© 2019 Iain R.躡mer, The Australian National University [\[Page 1/20\]](#)

A generic protected queue specification

```

generic
  type Element is private;
  type Index is mod <>; -- 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 : Element);
      entry Dequeue (Item : out Element);
      procedure Empty_Queue;
      function IsEmpty return Boolean;
      function IsFull return Boolean;
    private
      Queue : Queue_Type;
    end Protected_Queue;
  private
  type List is array (Index) of Element;
  type Queue_Type is record
    Top, Free : Index := Index'First;
    IsEmpty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Protected_Generic;

```

Generic components of the package:
Element can be anything
while the Index need to
be a modulo type.
© 2019 Iain R.躡mer, The Australian National University [\[Page 2/20\]](#)

A generic protected queue specification

```

generic
  type Element is private;
  type Index is mod <>; -- 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 : Element);
      entry Dequeue (Item : out Element);
      procedure Empty_Queue;
      function IsEmpty return Boolean;
      function IsFull return Boolean;
    private
      Queue : Queue_Type;
    end Protected_Queue;
  private
  type List is array (Index) of Element;
  type Queue_Type is record
    Top, Free : Index := Index'First;
    IsEmpty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Protected_Generic;

```

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

© 2019 Iain R.躡mer, The Australian National University [\[Page 3/20\]](#)

generic protected queue specification

```

generic
  type Element is private;
  type Index is mod >; -- 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 : Element);
      entry Dequeue (Item : out Element);
      procedure Empty_Queue;
      function Is_Empty return Boolean;
      function Is_Full return Boolean;
    private
      Queue : Queue_Type;
      end Protected_Queue;
    end Queue_Pack_Protected_Generic;
  end Queue_Pack_Protected_Generic;

  type List is array (Index) of Element;
  type Queue_Type is record
    Top : First_Index := Index'First;
    Is_Empy : Boolean := True;
    Elements : List;
  end record;
  end Queue_Pack_Protected_Generic;
end Queue_Pack_Protected_Generic;

private
  type List is array (Index) of Element;
  type Queue_Type is record
    Top : First_Index := Index'First;
    Is_Empy : Boolean := True;
    Elements : List;
  end record;
  end Queue_Pack_Protected_Generic;
end Queue_Pack_Protected_Generic;

procedure Enqueue (Item : out Element) when not Is_Full is
begin
  Queue (Top) := Item;
  Top := Top + 1;
end Enqueue;

procedure Dequeue (Item : in Element) when not Is_Empy is
begin
  Item := Queue (Top);
  Top := Top - 1;
end Dequeue;

function Is_Full return Boolean is
begin
  return Top = Queue'Length;
end Is_Full;

function Is_Empy return Boolean is
begin
  return Top = Index'First;
end Is_Empy;

procedure Empty_Queue is
begin
  Top := Index'First;
end Empty_Queue;

function Is_Empy return Boolean is
begin
  return Top = Index'First;
end Is_Empy;

procedure Protected_Queue is
begin
  Top := Index'First;
end Protected_Queue;

```

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

Rationale:

Entries can be blocking even if the protected object itself is unlocked. Hence a separate task waiting queue is provided per entry.

A generic protected queue specification

```

generic
  type Element is private;
  type Index := mod <-- Modulo defines size of the queue.
package Queue_Pack_Protected_Generic is
  type Queue_Type is limited private;
  protected body Queue_Pack_Protected_Generic is
    entry Enqueue (Item : Element);
    entry Dequeue (Item : out Element);
    procedure Empty_Queue;
    function Is_Empty return Boolean;
    function Is_Full return Boolean;
    private
      Queue : Queue_Type;
      protected_Queue;
      type List is array (Index) of Element;
      type Queue_Type is record
        Top_Index : Index := Index'First;
        Is_Empty : Boolean := True;
        Elements : List;
      end record;
    end Queue_Pack_Protected_Generic;
  end Queue_Pack_Protected_Generic;
  package body Queue_Pack_Protected_Generic is
    protected body Protected_Queue is
      entry Enqueue (Item : Element) when not Is_Full is
        begin
          Queue.Elements (Queue.Free) := Item;
          Queue.Free := Index'Succ (Queue.Free);
          Queue.IsEmpty := False;
        end Enqueue;
      entry Dequeue (Item : out Element) when not Is_Empty is
        begin
          Item := Queue.Elements (Queue.Top);
          Queue.Top := Index'Succ (Queue.Top);
          Queue.IsEmpty := Queue.Top = Queue.Free;
        end Dequeue;
      procedure Empty_Queue is
        begin
          Queue.Top := Index'First;
          Queue.IsEmpty := True;
        end Empty_Queue;
      function Is_Full return Boolean is
        begin
          return Queue.Top = Queue.Elements'Last;
        end Is_Full;
      function Is_Empty return Boolean is
        begin
          return Queue.Top = Queue.Elements'First;
        end Is_Empty;
    end Protected_Queue;
  end Queue_Pack_Protected_Generic;
end Queue_Pack_Protected_Generic;

```

page 120 of 967 "Introduction to Computer Organization" by Dr. Rajesh Kumar

©2010 by W. B. Zimmerman, The Australian National University

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

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
        begin
            Queue.Elements (Queue_Free) := Item;
            Queue_Free := Index'succ (Queue_Free);
            Queue_IsEmpty := False;
        end Enqueue;
        entry Dequeue (Item : out Element) when not Is_Empty is
        begin
            Item := Queue.Elements (Queue_Top);
            Queue.Top := Index'first;
            Queue.IsEmpty := Queue.Top = Queue.Items;
        end Dequeue;
    procedure Empty_Queue is
        begin
            Queue.Top = Index'First;
            Queue_Free := Index'First;
            Queue_IsEmpty := True;
            end Empty_Queue;
    function Is_Empty return Boolean is
        begin
            return Queue.Top = Queue.Items;
        end IsEmpty;
    function Is_Full return Boolean is
        begin
            if not Queue.IsEmpty and then Queue.Top = Queue.Items
                then
                    return True;
                else
                    return False;
            end if;
        end IsFull;
end Queue_Pack_Protected_Generic;

```

generic protected queue implementation

A generic protected queue *implementation*

generic protected queue test program

| | | | |
|--|--|--|---|
| <pre> with Ada_Task_Identification; use Ada_Task_Identification; with Ada.Text_Io; use Ada.Text_Io; with Queue_Pack.Protected_Generic; use Ada.Text_Io; 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; </pre> | <p>If more than one instance of a specific task is to be run then a task type as opposed to a concrete task is declared.</p> | <p>Multiple instances of a task can be instantiated eg. by declaring an array of this task type.</p> | <p>Tasks are started right when such an array is created.</p> |
| <pre> with Ada_Task_Identification; use Ada_Task_Identification; with Ada.Text_Io; use Ada.Text_Io; with Queue_Pack.Protected_Generic; use Ada.Text_Io; 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; </pre> | <p>If more than one instance of a specific task is to be run then a task type as opposed to a concrete task is declared.</p> | <p>Multiple instances of a task can be instantiated eg. by declaring an array of this task type.</p> | <p>Tasks are started right when such an array is created.</p> |
| <pre> with Ada_Task_Identification; use Ada_Task_Identification; with Ada.Text_Io; use Ada.Text_Io; with Queue_Pack.Protected_Generic; use Ada.Text_Io; 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; </pre> | <p>If more than one instance of a specific task is to be run then a task type as opposed to a concrete task is declared.</p> | <p>Multiple instances of a task can be instantiated eg. by declaring an array of this task type.</p> | <p>Tasks are started right when such an array is created.</p> |
| <pre> with Ada_Task_Identification; use Ada_Task_Identification; with Ada.Text_Io; use Ada.Text_Io; with Queue_Pack.Protected_Generic; use Ada.Text_Io; 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; </pre> | <p>If more than one instance of a specific task is to be run then a task type as opposed to a concrete task is declared.</p> | <p>Multiple instances of a task can be instantiated eg. by declaring an array of this task type.</p> | <p>Tasks are started right when such an array is created.</p> |

A generic protected queue test program

```

with Ada_Task_Identification;
with Ada.Text_10;
with Queue_Pack.Protected_Generic;
with Queue_Pack.Protected_Character;
procedure Queue_Test_Protected 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 declarations spawned off all the production code.

Often there are no statements for the "main task" (here explicitly stated by a null statement).

This task is prevented from terminating though until all tasks inside its scope terminated.

© 2017 Dr. rer. oec. R. Zimmer, The Institute of National Technical University

A generic protected queue test program

```

with Ada_Task_Identification;
with Ada.Text_IO;           use Ada.Text_Identification;
with Queue_Pack.Protected_Generic; use Queue.Text_10;
with Queue_Pack.Protected_Generic; use Queue_Pack.Protected.Character;
procedure Queue_Test_Protected 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 declarations spawned off all the production code.

Often there are no statements for the "main task".
 (here explicitly stated by a null statement).

This task is prevented from terminating though until all tasks inside its scope terminated.

© 2017 Dr. rer. oec. R. Zimmer, The Institute of National Technical University

A generic protected queue test program

```

with Ada.Task_Identification;      use Ada.Task_Identification;
with Ada.Text_IO;                 use Ada.Text_IO;
with Queue_Pack.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_Type);
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
    begin

```

A generic protected queue test program (*cont.*)

```

subType Some.Characters is Character range 'a' .. 'f'
task body Producer is
begin
  for Ch in Some.Characters loop
    Put_Line ("Task " & Image (Current_Task) & " finished");
    if Queue_Is_Empty then "EPNPY" else
      " and " &
      "If Queue_Is_Full then "FULL" else "
      " and prepares to add: " & Character
      " to the queue; ";
    end if;
    Queue.Enqueue (Ch);
    -- task might be blocked here
  end loop;
  Put_Line ("Task " & Image (Current_Task) & " ended");
end Producer;

```

```
...  
end Queue_Test_Protected_Generic;
```

A generic protected queue test program (cont.)

```

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

task body Producer is
begin
  begin
    for Ch in Some_Characters loop
      Put_Line ("Task " & Image (Current_Task) & " finds the queue to be " &
                " and " & Image (Ch);
      if Queue_Is_Full then "Full" else "not full" &
        " and prepares to add: " & Character'Image (Ch) &
        " to the queue");
```

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

```

    end loop;
  end;
end Producer;
Put_Line ("<---- Task " & Image (Current_Task) & " terminates");
end Producer;

```

generic protected queue test program (cont.)

```

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

task body Producer is
begin
  for Ch in Some_Characters loop
    Put_Line ("Task # & Image (Current_Task) & " finds the
              "if Queue is Empty then ""EMPTY"" else "not empty"
              " and & " if Queue is Full then ""FULL"" else "not FULL"
              " and prepare to add: & Character" Image
              " to the queue");
    Queue.Enqueue (Ch); -- task might be blocked here!
  end loop;
  Put_Line ("<----- Task # & Image (Current_Task) & " terminatia
            "Put Backward
            "Put Backward

```

There are 11 and they are

generic protected queue test program (cont.)

```

task body Consumer is
  Item : Character;
  Counter : Natural := 0;
begin
  Loop
    Queue.Dequeue (Item); -- task might be blocked here!
    Counter := Natural'Succ (Counter);
    Put_Line ("Task " & Image (Current_Task) &
              " received: " & Image (Item) &
              " and the queue appears to be " &
              " and " & Queue'Image);
    if Queue'Is_Empty then "EMPTY" else "not empty"
      and " and " & Queue'Image;
    end if;
    if Queue'Is_Full then "FULL" else "not full"
      and " afterwards";
    end if;
    exit when Item = Some_Characters'Last;
  end loop;
  Put_Line ("<--- Task " & Image (Current_Task) &
            " terminates and received" & Natural'Image (Count);
end Consumer;

```

A generic protected queue test program (cont.)

A generic protected queue test program (cont.)

```

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

task body Producer is
begin
  for Ch in Some_Characters loop
    Put_Line ("Task " & Image (Current_Task) & " finds the queue to be " &
              "Empty" when "EOF" else "not empty" &
              " " and " " & Image (Ch));
    if Queue_Is_Full then "FULL" else "not full" &
      " " and " " & prepares to add: " & Character Image (Ch) &
      " to the queue";
    Queue_Enqueue (Ch); -- task might be blocked here!
  end loop;
  Put_Line ("<----- Task " & Image (Current_Task) & " terminates.");
end Producer;

```

anything on this slide

A generic protected queue test program (cont.)

```

task body Consumer is
  Item : Character;
  Counter : Natural := 0;

  begin
    loop
      Queue.Dequeue (Item); -- task might be blocked here!
      Counter := Natural.Succ (Counter);
      Put_Line ("Task " & Image (Current_Task) &
                " received " & Character.Image (Item) &
                " and the queue appears to be " &
                (if Queue_Is_Empty then "EMPTY" else "not empty") &
                " and " &
                (if Queue_Is_Full then "FULL" else "not full") &
                " afterwords");
      exit when Item = Some_Characters'Last;
    end loop;
    Put_Line (<---- Task " & Image (Current_Task) &
              " terminates and received" & Natural.Image (Counter) &
              " items");
  end Consumer;

```

... anything on this slide

A generic protected queue test program (output)

```

Task_producers(1) finds the queue to be EMPTY and not full and prepares to add: 'a' to the queue.
Task_producers(1) finds the queue to be not empty and not full and prepares to add: 'a' to the queue.
Task_producers(2) finds the queue to be not empty and full and prepares to add: 'b' to the queue.
Task_consumers(1) finds the queue to be not empty and full and prepares to add: 'c' to the queue.
Task_producers(1) finds the queue to be EMPTY and not full and prepares to add: 'd' to the queue.
Task_consumers(1) received: 'a' and the queue appears to be EMPTY and not full afterwards.
Task_producers(3) tries the queue to be EMPTY and not full and prepares to add: 'a' to the queue.
Task_producers(1) tries the queue to be EMPTY and not full and prepares to add: 'c' to the queue.
Task_producers(2) tries the queue to be EMPTY and not full and prepares to add: 'd' to the queue.
Task_consumers(1) received: 'd' and the queue appears to be EMPTY and not full afterwards.
Task_producers(2) tries the queue to be EMPTY and not full and prepares to add: 'b' to the queue.
Task_consumers(2) received: 'd' and the queue appears to be EMPTY and not full afterwards.
Task_consumers(3) received: 'b' and the queue appears to be EMPTY and not full afterwards.
Task_producers(1) terminates.
Task_consumers(2) receives: 'f' and the queue appears to be not empty and not full afterwards.
Task_consumers(3) receives: 'f' and the queue appears to be not empty and not full afterwards.
Task_consumers(3) finds the queue to be not empty and full and prepares to add: 'f' to the queue.
Task_producers(3) terminates.
Task_consumers(1) receives: 'd' and the queue appears to be not empty and full afterwards.
Task_consumers(2) receives: 'e' and the queue appears to be not empty and full afterwards.
Task_consumers(3) receives: 'e' and the queue appears to be not empty and not full afterwards.
Task_producers(2) receives: 'f' and the queue appears to be EMPTY and not full afterwards.
Task_consumers(3) receives: 'f' and the queue appears to be not empty and not full afterwards.
Task_consumers(3) receives: 'f' and the queue appears to be not empty and full afterwards.
Task_producers(3) receives: 'f' and the queue appears to be EMPTY and not full afterwards.
Task_consumers(1) receives: 'f' and the queue appears to be not empty and full afterwards.
Task_consumers(2) receives: 'f' and the queue appears to be not empty and full afterwards.
Task_consumers(3) receives: 'f' and the queue appears to be not empty and not full afterwards.
Task_producers(3) terminates and received 12 items.
Task_consumers(3) terminates and received 5 items.

```

© 2019 University of Zaragoza, The Australian National University

Page 139 of 96 | Introduction & Languages | up to page 230

© 2019 University of Zaragoza, The Australian National University

140

Introduction & Languages

Ada

Service tasks (Message passing)

... introducing:

- Select statements.
- Rendezvous (synchronous message passing).
- Automatic termination.

```

package body Queue_Pack_Task_Generic is
  task body Queue_Task is
    subtype Marker is array (Marker_Range) of Element;
    type Queue_Type is record
      List : array (Marker_Range) of Element;
      Top : free : Marker := Marker_First;
      Is_Empty : Boolean := True;
      Elements : List;
      end record;
      Queue : Queue_Type;
      function Is_Empty return Boolean is
        (Queue_Is_Empty);
      function Is_Full return Boolean is
        (not Queue_Is_Empty and then Queue.Top = Queue_Free);
      end;

```

© 2019 University of Zaragoza, The Australian National University

Page 139 of 96 | Introduction & Languages | up to page 230

141

Introduction & Languages

An queue task specification

Page 139 of 96 | Introduction & Languages | up to page 230

141

Introduction & Languages

An queue task specification (cont.)

Page 139 of 96 | Introduction & Languages | up to page 230

142

Introduction & Languages

An queue task implementation

Page 139 of 96 | Introduction & Languages | up to page 230

142

Introduction & Languages

An queue task implementation (cont.)

Page 139 of 96 | Introduction & Languages | up to page 230

143

Introduction & Languages

An queue task implementation (cont.)

Page 139 of 96 | Introduction & Languages | up to page 230

143

Introduction & Languages

An queue task implementation

Page 139 of 96 | Introduction & Languages | up to page 230

144

Introduction & Languages

An queue task implementation (cont.)

Page 139 of 96 | Introduction & Languages | up to page 230

144

Introduction & Languages

An queue task implementation

Page 139 of 96 | Introduction & Languages | up to page 230

145

Introduction & Languages

Data structures and functions

Page 139 of 96 | Introduction & Languages | up to page 230

145

Introduction & Languages

Data structures and functions are declared local to the task.

Page 139 of 96 | Introduction & Languages | up to page 230

146

Introduction & Languages

Data structures and functions are declared local to the task.

Page 139 of 96 | Introduction & Languages | up to page 230

146

Introduction & Languages

An queue task implementation (cont.)

```
(...)
begin
  begin
    loop
      select
        when not Is_Full =>
          accept Enqueue (Item : Element) do
            Queue.Elements (Queue_Free) := Item;
            end Enqueue;
            Queue_Free := (Queue_Free + 1) mod Queue_Size;
            Queue_Is_Empty := False;
        or
        when not Is_Empty =>
          accept Dequeue (Item : out Element) do
            Item := Queue.Elements (Queue_Top);
            end Dequeue;
            Queue_Top := (Queue_Top + 1) mod Queue_Size;
            Queue_Is_Empty := Queue.Top = Queue_Free;
        (...)
```

© 2019 Dr. Karl Zimmer, The Australian National University
page 145 of 161 | Introduction & Languages | up to page 230

A generic queue task test program

```
with Ada.Text_Io; use Ada.Text_Io;
with Queue_Pack_Task.Generic;
procedure Queue_Test_Protected_Generic (Element => character, Queue_Size => 12);
package Queue_Pack_Task_Character is
  new Queue_Pack_Task_Generic (Queue : Protected_Queue;
  Producer is end Producer;
  Consumer is end Consumer);
  (...)
```

Identical to the test program for protected objects.

© 2019 Dr. Karl Zimmer, The Australian National University
page 146 of 161 | Introduction & Languages | up to page 230

Introduction & Languages

Ada

Abstract types & dispatching

... introducing:

- 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!

© 2019 Dr. Karl Zimmer, The Australian National University
page 147 of 161 | Introduction & Languages | up to page 230

Introduction & Languages

An queue task implementation (cont.)

```
(...)
begin
  begin
    loop
      select
        when not Is_Full =>
          accept Enqueue (Item : Element) do
            Queue.Elements (Queue_Free) := Item;
            end Enqueue;
            Queue_Free := (Queue_Free + 1) mod Queue_Size;
            Queue_Is_Empty := False;
        or
        when not Is_Empty =>
          accept Dequeue (Item : out Element) do
            Item := Queue.Elements (Queue_Top);
            end Dequeue;
            Queue_Top := (Queue.Top + 1) mod Queue.Size;
            Queue_Is_Empty := Queue.Top = Queue_Free;
        (...)
```

Client tasks are released (and continue concurrent operations). Service task completes the operation on its own.

© 2019 Dr. Karl Zimmer, The Australian National University
page 146 of 161 | Introduction & Languages | up to page 230

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
    select
      Queue.Dequeue (Item); -- task might be blocked here!
      Put ('Received:'); Put (Item); Put_Line ("!");
    exit; -- main task loop
    delay 0.001;
  end select;
  end loop;
begin
  null;
end Queue_Test_Protected_Generic;
```

These two calls are 'hammering' the queue task concurrently and at full CPU speed.

Identical to the test program for protected objects.

© 2019 Dr. Karl Zimmer, The Australian National University
page 147 of 161 | Introduction & Languages | up to page 230

Introduction & Languages

An abstract queue specification

```
generic
  type Element is private;
  type Queue_Pack_Task_Generic is
    task type Queue_Task is
      entry Enqueue (Item : Element);
      entry Dequeue (Item : out Element);
      entry Is_Full (Result : out Boolean);
    end Queue_Task;
    end Queue_Pack_Task_Generic;
```

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

© 2019 Dr. Karl Zimmer, The Australian National University
page 147 of 161 | Introduction & Languages | up to page 230

Introduction & Languages

An queue task implementation (cont.)

```
(...)
begin
  begin
    loop
      select
        when not Is_Full =>
          accept Enqueue (Item : Element) do
            Queue.Elements (Queue_Free) := Item;
            end Enqueue;
            Queue_Free := (Queue_Free + 1) mod Queue_Size;
            Queue_Is_Empty := False;
        or
        when not Is_Empty =>
          accept Dequeue (Item : out Element) do
            Item := Queue.Elements (Queue.Top);
            end Dequeue;
            Queue.Top := (Queue.Top + 1) mod Queue.Size;
            Queue_Is_Empty := Queue.Top = Queue_Free;
        (...)
```

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

© 2019 Dr. Karl Zimmer, The Australian National University
page 147 of 161 | Introduction & Languages | up to page 230

An queue task specification

```
generic
  type Element is private;
  type Queue_Pack_Task_Generic is
    task type Queue_Task is
      entry Enqueue (Item : Element);
      entry Dequeue (Item : out Element);
      entry Is_Full (Result : out Boolean);
    end Queue_Task;
    end Queue_Pack_Task_Generic;
```

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

© 2019 Dr. Karl Zimmer, The Australian National University
page 147 of 161 | Introduction & Languages | up to page 230

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);
    procedure Dequeue (Q : in out Queue_Interface; item : out Element);
    end Queue_Pack_Abstract;
```

© 2019 Dr. Karl Zimmer, The Australian National University
page 147 of 161 | Introduction & Languages | up to page 230

concrete queue implementation

```

backstage body Queue.Pack.Concrete is
protected body Protected_Concrete is
begin
  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;
  entry Dequeue (Item : out Element) when not Is_Full is
    begin
      Item := Queue.Elements (Queue.Top); Queue.Top := Index'succ (Queue.Top);
      Queue.Is_Full := Queue.Top = Queue.Elements;
      Queue.Elements := Queue.Top - 1;
      end Dequeue;
    procedure Clear
    begin
      Queue.Top := Index'first; Queue.Free := Index'First; Queue.Is_Full := True;
      Queue.Is_Empty := True;
    end Clear;
    function Is_Full return Boolean is
      begin
        return Queue.Top = Queue.Elements;
      end Is_Full;
    function Is_Empty return Boolean is
      begin
        return Queue.Top = Queue.Elements;
      end Is_Empty;
    function First return Element is
      begin
        return Queue.Elements;
      end First;
    end Protected_Concrete;
end Queue.Pack.Concrete;

```

A dispatching test program

```

with Ada.Text_IO;      use Ada.Text_IO;
with Queue_Pack.Abstract;
with Queue_Pack.Concrete;
procedure Queue_Test.Dispatching is
  package Queue_Pack_Abstract(Character) is
    new Queue_Pack_Abstract(Character);
  use Queue_Pack_Abstract.Character;
  type Queue_Size is mod 3;
  package Queue_Pack_Concrete is
    new Queue_Pack_Concrete(Queue_Pack_Abstract.Character);
    use Queue_Pack.Character;
    type Queue_Class is access all Queue_Interface_Class;
    task Queue_Holder; -- could be on an individual partition
    task Queue_User; -- could be on an individual partition
    entry Send_Queue (Remote_Queue : Queue_Class);
    end Queue_User;
  end Queue_Test.Dispatching;
begin
  null;
end Queue_Test.Dispatching;

```

A dispatching test program

```

with Ada.Text_IO;
use Ada.Text_IO;

with Queue_Pack_Abstract;
with Queue_Pack_Concrete;

procedure Queue_Test_Dispatching is
  package Queue_Pack_Abstract_Character is
    new Queue_Pack_Abstract(Character);
  use Queue_Pack_Abstract_Character;
  type Queue_Size is mod 3;
  package Queue_Pack_Concrete is
    new Queue_Pack_Concrete(Queue_Pack_Abstract_Character, Queue_Size);
  use Queue_Pack_Concrete;
  use Queue_Pack_Character;
  type Queue_Class is access all Queue_Interface_Class;
  task Queue_Holder; -- could be on an individual partition / separate computer
  task Queue_User is -- could be on an individual partition / separate computer
    entry Send_Queue (Remote_Queue : Queue_Class);
    end Queue_User;
  (...)

begin
  null;
end Queue_Test_Dispatching;

```

ISO/IEC 9080-1:2014(E) | Chapter 1: Introduction & Languages | 01 / 102 / 2013

dispatching test program

```
with Ada.Text_Io;           use Ada.Text_Io;
with Queue_Pack_Abstract; 
with Queue_Pack_Concrete;
procedure Queue_Test.Dispatching is
  package Queue_Pack_Abstract_Character is
    new Queue_Pack_Abstract (Character);
  use Queue_Pack_Abstract_Character;
  type Queue_Size is mod 3;
  package Queue_Pack_Concrete is
    new Queue_Pack_Concrete (Queue_Pack_Abstract_Character, Queue_Size);
  use Queue_Pack_Concrete;
  type Queue_Class is access all Queue_Interface'Class;
  task Queue_Holder; -- could be on an individual partition / separate computer
  task Queue_User; -- could be on an individual partition / separate computer
  entry Send_Queue (Remote_Queue : Queue_Class);
end Queue_User;
```

A dispatching test program

```

with Ada.Text_IO;
use Ada.Text_IO;

with Queue_Pack.Abstract;
with Queue_Pack.Concrete;

procedure Queue_Tests.Dispatching is
  new Queue_Pack_Abstract.Character;
use Queue_Pack_Abstract.Character;

type Queue_Size is mod 3;
package Queue_Pack_Concrete is
  new Queue_Pack_Concrete(Queue_Pack_Abstract.Character, Queue_Size);
use Queue_Pack_Concrete;
use Queue_Pack_Character;

type Queue_Class is access all Queue_Interface.Class;
task Queue_Holder; -- could be on an individual partition / separate computer
task Queue_User is -- could be on an individual partition / separate computer
entry Send_Queue (Remote_Queue : Queue_Class);
end Queue_User;
end Queue_Tests;

```

A dispatching test program (cont.)

```

task body Queue_Holder is
    Local_Queue : constant Queue_Class := new Protected_Queue;
    Item : character;
begin
    Queue_User.Send_Queue (Local_Queue);
    Local_Queue.all.Dequeue (Item); Put_Line ("Local dequeue (Holder) : " & Character'Image (Item));
    end Queue_Holder;

task body Queue_User is
    Local_Queue : constant Queue_Class := new Protected_Queue;
    Item : character;
begin
    Local_Queue.all.Dequeue (Item); accept Send_Queue (Remote_Queue : Queue_Class) do
        Remote_Queue.all.Enqueue ("r"); -- potentially a remote procedure call!
        Local_Queue.all.Enqueue ("l");
    end Send_Queue;
    Local_Queue.all.Dequeue (Item); Put_Line ("Local dequeue (User) : " & Character'Image (Item));
end Queue_User;

```

A dispatching test program (cont.)

```

task body QueueHolder is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : Character;
begin
  Handing over the Holder's queue
  -- via synchronous message passing.
  Queue_User.Send_Queue (Local_Queue);
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (Holder): " & Character'Image (Item));
end Queue_Holder;

task body QueueUser is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : Character;
begin
  accept Send_Queue (Remote_Queue : Queue_Class) do
    Remote_Queue.all.Enqueue ("r"); -- potentially a remote procedure call!
    Local_Queue.all.Enqueue ("l");
  end Send_Queue;
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (User) : " & Character'Image (Item));
end Queue_User;

```

A dispatching test program (cont.)

```

task body QueueHolder is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : Character;
begin
  Handing over the Holder's queue
  -- via synchronous message passing.
  Queue_User.Send_Queue (Local_Queue);
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (Holder): " & Character'Image (Item));
end Queue_Holder;

task body QueueUser is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : Character;
begin
  accept Send_Queue (Remote_Queue : Queue_Class) do
    Remote_Queue.all.Enqueue ("r"); -- potentially a remote procedure call!
    Local_Queue.all.Enqueue ("l");
  end Send_Queue;
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (User) : " & Character'Image (Item));
end Queue_User;

```

A dispatching test program (cont.)

A *dispatching test program* (*cont.*)

```

task body Queue.Holder is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : character;
begin
  Queue_User.Send_Queue (Local_Queue);
  Local_Queue.all.Dequeue (Item);
  Put_Line ("local dequeue (Holder) : " & Character'Image (Item));
end Queue_Holder;

task body Queue_Listener is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : character;
begin
  accept Send_Queue (Remote_Queue : Queue_Class) do
    Remote_Queue.all.Enqueue ('r') -- potentially a remote
    Local_Queue.all.Enqueue ('l');
  end Send_Queue;
  Local_Queue.all.Dequeue (Item);
  Put_Line ("local dequeue (User) : " & Character'Image (Item));
end Queue_Listener;

```

A dispatching test program (cont.)

A dispatching test program (cont.)

```

task body Queue_Holder is
    Local_Queue : constant Queue_Class := new Protected.Queue;
    Item : Character;
begin
    Queue_User.Send_Queue (Local_Queue);
    Local_Queue.all.Dequeue (Item);
    Put_Line ("Local dequeu (holder) : " & Character'Image (Item));
    end Queue_Holder;

task body Queue_User is
    Local_Queue : constant Queue_Class := new Protected.Queue;
    Item : Character;
begin
    accept Send_Queue (Remote_Queue : Queue_Class) do
        Remote_Queue.all.Enqueue ('*'); -- potentially a remote procedure call!
        Local_Queue.all.Enqueue ("1");
    end Send_Queue;
    Local_Queue.all.Dequeue (Item);
    Put_Line ("Local dequeu (User) : " & Character'Image (Item));
end Queue_User;

```

Page 72 of 191 | Chapter 8 & Language Reference | Up to Page 72

A dispatching test program (cont.)

```

task body Queue_Holder is
local_Queue : constant Queue_Class := new Protected_Queue;
Item : Character;
begin
  Queue_User.Send_Queue (Local_Queue);
  Local_Queue.all.Dequeue (Item);
  Put_Line ("local dequeue (Holder): " & Character'Image (Item));
end Queue_Holder;

task body Queue_User is
local_Queue : constant Queue_Class := new Prot
Item : Character;
begin
  accept Send_Queue (Remote_Queue : Queue_Class) do
    Remote_Queue.all.Enqueue ('-');
    Local_Queue.all.Enqueue ('1');
    end Send_Queue;
  Local_Queue.all.Dequeue (User) :> Character'Image (Item);
end Queue_User;

```

Page 173 of 363 | Information & Languages - up to Now | 23

Introduction & Languages

Ada

Coordinating concurrent reader tasks

... introducing:

- Entry families
- Entry attributes

A dispatching test program (cont.)

```

task body Queue_Holder is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item       : character;
begin
  Queue_User.Send_Queue (Local_Queue);
  Local_Queue.all.Dequeue (Item);
  Put_Line ("local dequeue (Holder): " & Character'Image (Item));
end Queue_Holder;
end task body Queue_Holder;

task body Queue_User is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item       : character;
begin
  accept Send_Queue (Remote_Queue : Queue_Class) do
    Remote_Queue.all.Enqueue ('r'); -- potentially a remote p
    Local_Queue.all.Enqueue ('l');
  end Send_Queue;
  Local_Queue.all.Dequeue (Item);
  Put_Line ("local dequeue (User) : " & Character'Image (Item));
end task body Queue_User;

```

Copyright © 2019, The Institution of Naval Architects
Page 75 of 86

```

generic
    type Element is private;
    type Queue_Enum is (<>);
    Queue_Size : Positive := 18;
    package Queues_Pack_Protected_Generic is
        type Queue_Type is limited private;
        protected type Protected_Queue is
            entry Enqueue (Queue_Enum Item : Element);
            entry Dequeue (Queue_Enum Item : out Element);
            function Is_Empty () : Queue_Enum return Boolean;
            function Is_Full return Boolean;
    private
        Queue : Queue_Type;
    end Protected_Queue;
  (...)
```

| | |
|--|---|
| <pre> task body Queue::Holder is Local_Queue : constant Queue_Class := new Protected_Queue; Item : character; begin Queue_User.Send(Queue'(Local_Queue); Local_Queue.all.Dequeue(Item); Put_Line("Local dequeue (Holder) : " & Character'Image (Item)); end Queue_Holder; </pre> | Reading |
| <pre> task body Queue::User is Local_Queue : constant Queue_Class := new Protected_Queue; Item : character; begin accept Send_Queue (Remote_Queue : Queue_Class) do Remote_Queue.all.Enqueue ('r'); -- potentially a remote procedure Local_Queue.all.Enqueue ('l'); end Send_Queue; Local_Queue.all.Dequeue(Item); Put_Line("Local dequeue (User) : " & Character'Image (Item)); end Queue_User; </pre> | Reading |
| <p style="text-align: right;">Page 73 of 100 Introduction to Parallel Programming</p> | |

A protected, generic queues specification

```

generic
    type Element is private;
    type Queue<T> is (→);
    Queue<T>.Positive := 10;
    package Queues.Pack.Protected_Generic is
        type Queue<T>.Type is limited private;
        protected type Protected_Queue is
            entry Enqueue (Queue<T>un) (Item : Element);
            entry Dequeue (Queue<T>un) (Item : out Element);
        end;
        function IsEmpty (Q : Queue<T>un) return Boolean;
        function Is_Full return Boolean;
    end;
private
    Queue : Queue<T>;
    end Protected_Queue;
  end
  
```

page 177 of 390 Introduction & Lab Session 1

© 2019 Werner Zommer The Australian National University

A protected, generic queues implementation

```

package body Queues.Pack.Protected_Generic is

protected body Protected_Queue is

entry Enqueue (Item : Element) when not Is_Full is
begin
    Queue.Elements (Queue_Free := (Element => Item, Reads => None_Read));
    Queue_Free := (Queue_Free + 1) mod Queue.Size;
    end Enqueue;

entry Dequeue (for Q in Queue_Error) (Item : out Element)
when not Is_Empty (Q) and then (Error_Count = 0 or else Is_Full) is
begin
    Item := Queue.Elements (Queue_Readers (Q)).Item;
    Queue.Elements (Queue_Readers (Q)).Reads (Q) := Queue.Size;
    Queue_Readers (Q) := (Queue_Readers (Q) + 1) mod Queue.Size;
    end Dequeue;

function Is_Full return Boolean;
begin
    if Queue_Free = 0 then
        return True;
    else
        return False;
    end if;
end Dequeue;

function Is_Empty (Q : Queue_Error) return Boolean;
begin
    if Queue_Free = Queue.Size then
        return True;
    else
        return False;
    end if;
end Is_Full;

function Is_Full return Boolean is
begin
    if Queue_Free = 0 then
        return True;
    else
        return False;
    end if;
end Is_Full;

function Is_Empty (Queue_Error : Queue_Error) return Boolean;
begin
    if Queue_Free = Queue.Size then
        return True;
    else
        return False;
    end if;
end Is_Empty;

end Protected_Queue;

end Queues.Pack.Protected_Generic;

```

A protected, generic queues test program

```

with Queues.Pack_Protected_Generic;
with Ada.Text_IO; use Ada.Text_IO;
procedure Queues_Test_Protected_Generic is
  type Sequence is (Ready, Set, Go);
  type Flight_Status is (Take_Off, Cruising, Landing);
  package Queue_Pack_Protected_Character is
    new Queues_Pack_Protected_Generics ('A'..'Z');
    use Queue_Pack_Protected_Character;
    Queue : Protected Queue;
    task type Avionics_Module is
      entry Provide_State (State : Flight_Status);
    end Avionics_Module;
    Avionics : array (Flight_Status) of Avionics_Module;
  end;
end;

```

© 2019 Uwe R. Zimmer, The Australian National University
page 184 of 916 | Translation & Linguistics [up to page 2340]

A protected, generic queues *test program* (*cont.*)

(\hookrightarrow) task body Avionics_Module is
 Local_State : Flight_Status;
 Item : Sequence;
 begin
accept Provide_State (State : Flight_Status) **do**
 Local_State := State;
 end Provide_State;
 for Order **in** Sequence **loop**
 Queue.Dequeue (Local_State) (Item);
 Put_Line (Flight_Status'Image (Local_State) &
 " says: " & Sequence'Image (Item));
 end loop;
 end Avionics_Module;

begin
 for State **in** Flight_Status **loop**
 Avionics (State).Provide_State (State);
 end loop;

for Order **in** Sequence **loop**
 Queue.Enqueue (Order);
 Put_Line (CItem added to queue: " & Sequence'Image (Order));
end loop;

end Queues.Test_Provider_Generic;

75/230, Page 27, The Australian National University

page 187 29.9.11 Introduction & Languages 1 up to page 230

A protected, generic queues implementation

```

package body Queues.Pack_Protected_Generic is
  protected body Protected_Queue is
    entry Enqueue (Item : Element) when not Is_Full is
      begin
        Queue.Elements (Queue.Free) := (Element => Item, Reads => None, Reader => None);
        Queue.Free := (Queue.Free + 1) mod Queue.Size;
        end Enqueue;
        entry Dequeue (for Q in Queue.Enqueue) (Item : out Element)
          when not Is_Empty (Q) and then (Enqueue'Count = 0 or else Is_Full) is
          begin
            Item := Queue.Elements (Queue.Readers (Q)).Element;
            Queue.Elements (Queue.Readers (Q)).Reads (Q) := True;
            Queue.Readers (Q) := Queue.Readers (Q) + 1 mod Queue.Size;
            end Dequeue;
            function Is_Empty (Q : Queue.Enqueue) return Boolean is
              (Queue.Elements (Queue.Readers (Q)).Reads (Q));
            end Is_Empty;
            function Is_Full (Q : Queue.Enqueue) return Boolean is
              (Queue.Elements (Queue.Readers (Q)).Reads (Q));
            end Is_Full;
            function Dequeue (Q : Queue.Enqueue) return Element is
              (Queue.Elements (Queue.Readers (Q)).Element);
            end Dequeue;
            function Fill_Queue (Q : Queue.Enqueue) is
              (Queue.Elements (Queue.Readers (Q)).Reads (Q));
            end Fill_Queue;
            end Protected_Queue;
            end Queues.Pack_Protected_Queues;

```

A protected, generic queues test program

```

with Queues_Pack_Protected.Generic;
with Ada.Text_Io; use Ada.Text_Io;
procedure Queues_Test_Protected.Generic is
  type Sequence is (Ready, Set, Go);
  type Flight_Status is (Take_Off, Cruising, Landing);
  package Queue_Pack_Protected.Character is
    new Queues_Pack_Protected.Generic
      (Element => Sequence, Queue_Enum => Flight_Status, Queue_Size
       =>自然数);
  use Queue_Pack_Protected.Character;
  Queue : Protected.Queue;
  task type Avionics_Module is
    entry Provide_State (State : Flight_Status);
    end Avionics_Module;
  Avionics : array (Flight_Status) of Avionics_Module;
end;

```

Page 185 of 961 | [Interactions](#)

A protected, generic queues implementation

```

package body Queues.Pack_Protected_Generic is
protected body Protected_Queue is
entry Enque (Item : Element) when not Is_Full is
begin
  Queue.Elements (Queue.Free) := (Element => Item, Reads => None_Read);
  Queue.Free := (Queue.Free + 1) mod Queue.Size;
end Enqueue;
entry Dequeue (for O in Queue_Enum) (Item : out Element)
when not Is_Empty (O) and then (Enqueue'Count = 0 or else Is_Full) is
begin
  Item := Queue.Elements (Queue.Readers (O)).Elem;
  Queue.Elements (Queue.Readers (O)).Reads (O) := True;
  Queue.Readers (O) := (Queue.Readers (O) + 1) mod Queue.Size;
end Dequeue;
function Is_Empty (Q : Queue_Enum) return Boolean is
  Queue.Elements (Queue.Readers (Q)).Reads (Q);
begin
  return Queue.Elements (Queue.Readers (Q)).Reads (Q) = All_Read;
end Protected_Queue;
end Queues.Pack_Protected_Generic;

```

The multi-reader data-structure makes full and empty detections easy.

A protected, generic `aveyes` test program (cont.)

```

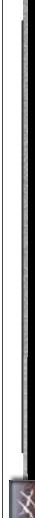
local.State : Flight._States;
begin
  Sequence;
  accept Provide_State (State : Flight._States) do
    Local.State := State;
    end Provide.State;
    for Order in Sequence loop
      Queue.Enqueue (Local.State) (Item);
      Put_Line (Flight._States.Image (Local._State) &
                " says: " & Sequence.Image (Item));
    end loop;
  end Avionics.Module;

begin
  for State in Flight._States loop
    Avionics (State).Provide.State (State);
  end loop;
  for Order in Sequence loop
    Queue.Enqueue (Order);
    Put_Line ("Item added to Queue: " & Sequence.Image (Order));
  end loop;
  end Queues.Test.Protected.Generic;

```

http://www.mcs.anl.gov/~kofman/Parallel/Parallel%20Programming%20in%20Java%20-%20Implementation%20of%20the%20Avionics%20Module.htm

| A protected, generic queues specification | <small>page 18 of 30 Threading & Concurrency 2017-2018</small> |
|--|---|
| <pre> generic type Element is private; type Queue_Enum is (<>); Queue_Size : Positive := 18; package Queues_Pack_Protected_Generic is type Queue_Type is limited private; protected type Protected_Queue is entry Enqueue (Queue_Enum) (Item : in Element); entry Dequeue (Queue_Enum) (Item : out Element); function Is_Empty (Q : Queue_Enum) return Boolean; function Is_Full return Boolean; private Queue : Queue_Type; end Protected_Queue; end; end; </pre> | <p>For an actual real-time system, functional and temporal specifications (e.g., contracts) would be added and preferably proven.</p> |



Introduction & Languages

Real-Time Java

Real-Time Specification for Java

- ☞ Standard library classes still rely on garbage collection!
 - ☞ i.e. usage of standard libraries destroys hard real-time integrity.
- ☞ RTSI is backwards compatible
 - ☞ i.e. no syntactical extensions and no additional compiler checks (integers are still wrapping around, switch-statements need breaks etc.).
- ☞ Allows for different Java-engine implementations:
 - ☞ i.e. scheduling is not mandatory.
 - ☞ in terms of semantics: e.g. "instantiations per time span" can but does not need to imply equal distance intervals.
- ☞ Concept is still based on Java-style object oriented programming
 - ☞ Inheritance anomaly in concurrent systems needs to be considered carefully.

© 2010 David Kullman, The Australian National University
Page 19 of 99 | Author: T. Märtinsson & D. Kullman | URL: http://rtg.iress.anu.edu.au/rtg/2010/



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:**
 - local, discrete device control.
- **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, interfaces, pipeline control, state machines.

- **Threads:** Priorities, scheduling, and dispatching
- **Memory:** Controlled garbage collection and physical memory access
- **Synchronization:** Ordered queues, and priority ceiling protocols
- **Asynchronism:** Generalized asynchronous event handling, asynchronous transfer of control, timers, and an operational implementation of thread termination

☞ All current real-time Java extensions keep the underlying consequence object orientation. ☞ Predictability often questionable.

☞ Some restrict the language standard, some extend it.

Page 19 of 91 | Dr. Jürgen "Jug" Grauer | Real-Time Java & Languages | Spring 2013



Introduction & Languages

Esterel

Transformational \leftrightarrow *Interactive* \leftrightarrow *Reactive*

- **Transformational** (functional) systems:
Generating outputs based on input and stop, utilizing no or only a small number of internal states.
- **Interactive** systems:
I.e. servers and other systems in long-term operation, requesting occasional inputs, and accepting service-calls, when there are resources to do so.
- **Reactive** (reflex) systems:
Systems which are reacting to external stimuli only (by generating other stimuli). Can be viewed as a predictable, functional system, which is listening to inputs continuously, while holding enough resources to ensure specified reaction times.



Introduction & Languages

Esterel

Strong synchrony or “zero delay” assumption

In logical terms:

- All operations are instantaneous!

Introduction & Languages

- Enhanced thread model (memory attributes, more precise specs).
- Enabling powerful and highly adaptive scheduling policies.
- Introducing scoped, immortal (keep the garbage collector out), and physical memory to lava (map to a physical architecture).
- Introducing timers, interrupts, and more exceptions.
- Higher resolution time model.
- Optional support for POSIX signals.

Introduction & Languages

Esterel

Control-oriented \leftrightarrow Dataflow-oriented

Introduction & Languages

Esterel

Introduction & Languages

Synchronous languages

Non-causality in Synchronous Languages

- Non-reactive output:


```

module non-deterministic;
  output O;
  present O else emit O end;
end module;
```
- Cyclic dependencies with multiple signals:


```

module cyclic_dependency;
  [ present A then emit B end || present B then emit A end ]
end module;
```
- All examples contain a reference to "the future" i.e. are "cyclic".

© 2019 Use R. Zitzen: The Australian National University page 209 of 360 | Chapter 1: Introduction & Languages up to page 210

Introduction & Languages

VHDL hardware description language (VHDL)

VHDL

- Standardized hardware description language (IEEE 1076-2008)
- Can be data-flow or control-flow oriented.
- Programming in the large (packages, generics).
- Entities, architectures (processes), and configurations are separate.
- Signals can be digital or analog.
- Strong typing (all basic Ada types plus low level types; std_logic).
- Modules can be clocked independently or not at all (combinatorial).
- Concurrency only limited by number of gates on module (use millions).
- High level synchronization primitives (protected types).

© 2019 Use R. Zitzen: The Australian National University page 211 of 360 | Chapter 1: Introduction & Languages up to page 210

Introduction & Languages

Process Algebras

Timed CSP

$$\text{P} ::= \text{Stop} \mid \text{Skip} \mid \text{Wait } t \mid t^a \rightarrow P \mid P \sqcap P' \mid P \sqcup P' \mid p_a \rightarrow P \mid p_a \sqcup P \mid p_a \sqcap P \mid P \sqsupseteq P'$$

where: P is a process, t is an event, A is a set of events, and f is a non-negative real number.

- Stop : terminal process.
- Skip : Null process.
- Wait t : Delay process.
- $a \rightarrow P$: Process P is preceded by event a .
- P, P' : Processes in sequence.
- $P \sqcap P'$: Deterministic alternative.
- $P \sqcup P'$: Non-deterministic alternative.
- $a \rightarrow P_a$: Process P is preceded by one event $a \in A$.
- $P \sqsupseteq P'$: Alternative based on time

© 2019 Use R. Zitzen: The Australian National University page 212 of 360 | Chapter 1: Introduction & Languages up to page 210

Introduction & Languages

Synchronous languages

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.
- Erased: fully acyclic programs are considered more restrictive, because cyclic dependencies can make programs more intuitive/simpler.
- Cyclic programs can be reactive and deterministic.

© 2019 Use R. Zitzen: The Australian National University page 209 of 360 | Chapter 1: Introduction & Languages up to page 210

Introduction & Languages

VHDL state machine example

```

entity enum is port (Clock, Reset : in Std_Logic;
                     A, B : out Std_Logic);
end entity;
architecture A_Simple_Moore.Machine of enum is
begin
  type States is (Start, A_Detected, B_Detected, AB_Detected);
  attribute enum_encoding of States : type is "one-hot"; -- "grey", "binary", ...
  signal Current_State, Next_State: States;
  begin
    Synchronous_Proc: process (Clock, Reset)
    begin
      if (Reset='1') then
        Current_State <= Start;
      elsif (Clock'event and Clock = '1') then
        Current_State <= Next_State after 1 ns; -- inertial delay
      end if;
    end process; -- End Synchronous_Proc
  end;
```

© 2019 Use R. Zitzen: The Australian National University page 212 of 360 | Chapter 1: Introduction & Languages up to page 210

Introduction & Languages

PEARL

Process and Experiment Automation Realtime Language

- Simple and 'readable' language for small projects.
- Supports tasking and timed activations.
- Supports interrupts, signals, semaphores, and bolt variables.
- Lacks support for programming in the large!

Is a settled standard:

- DIN 66253-1: Basic PEARL 1981
- DIN 66253-2: Full PEARL 1982 as both replaced by DIN 66253-2: PEARL 90 1996
- DIN 66253-3: PEARL for distributed systems 1989

© 2019 Use R. Zitzen: The Australian National University page 213 of 360 | Chapter 1: Introduction & Languages up to page 210

Introduction & Languages

Synchronous languages

Strong synchrony or "zero delay" assumption

Enables:

- Strong analysis and simplification tools.
- Significantly easier program verification.
- Straight forward hardware implementations.

© 2019 Use R. Zitzen: The Australian National University page 210 of 360 | Chapter 1: Introduction & Languages up to page 210

Introduction & Languages

VHDL state machine example

```

Combinatorial_Process: process (Current_State, A, B)
begin
  case Current_State is
    when Start => Out <= '0';
    when A_Detected => Out <= '0' after 1 ns;
    when B_Detected => Out <= '0' after 1 ns;
    when AB_Detected => Out <= '0' after 1 ns;
    when A then Next_State <= A_Detected;
    when B then Next_State <= B_Detected;
    when AB then Next_State <= AB_Detected;
    end if;
  end if;
  Next_State <= Start;
end process;
```

© 2019 Use R. Zitzen: The Australian National University page 211 of 360 | Chapter 1: Introduction & Languages up to page 210

Introduction & Languages

PEARL

Process and Experiment Automation Realtime Language

- Simple and 'readable' language for small projects.
- Supports tasking and timed activations.
- Supports interrupts, signals, semaphores, and bolt variables.
- Lacks support for programming in the large!

Is a settled standard:

- DIN 66253-1: Basic PEARL 1981
- DIN 66253-2: Full PEARL 1982 as both replaced by DIN 66253-2: PEARL 90 1996
- DIN 66253-3: PEARL for distributed systems 1989

© 2019 Use R. Zitzen: The Australian National University page 214 of 360 | Chapter 1: Introduction & Languages up to page 210

Introduction & Languages

PEARL

MODULE:

```

SYSTEM: Alert: Hard_Int (7):
SPECIFY Alert INTERRUPT;
SPECIFY Help TASK GLOBAL;
SPECIFY Pushed BIT GLOBAL;
DECLARE Swich BIT INITIAL 0;

Init : TASK MAIN;
WHEN Alert ACTIVATE Recovery;
ENABLE Alert;
Swich := Pushed;

Recovery: TASK PRIORITY 5;
DISABLE Alert;
IF Swich = 1 THEN ACTIVATE Help; FIN;
AFTER 30 MIN ALL 5 MIN DURING 1 HRS ACTIVATE Help;
END;

```

MODULE:

```

MODEND;

```

© 2019 Univ. K. Zinner, The Austrian National University

Page 217 of 360 | Chapter 1: Introduction & Languages | up to page 210

Introduction & Languages

POSIX - some of the relevant standards...

| | | |
|---------|----------------------------------|--|
| 1003.1 | OS Definition | single process, multi process, job control, signals, user groups, file system, file attributes, file device management, file locking, device I/O, device-specific control, system database, pipes, I/F, O... |
| 1003.1b | Real-time Extensions | real-time signals, priority scheduling, timers, asynchronous I/O, prioritized I/O, synchronized I/O, sync, mapped files, memory protection, memory passing, semaphores, ... |
| 1003.1c | Threads | multiple threads within a process, includes support for thread control, thread attributes, priority scheduling, mutexes, priority inheritance, mutex, priority ceiling and condition variables |
| 1003.1d | Additional Real-time Extensions | new process, create semantics (spawn), sporadic server scheduling, execution time monitoring, processes and threads, O (advise) information, timeouts on blocking functions, server control, and interrupt control |
| 1003.1j | Distributed Real-time Extensions | load memory, nanosleep improvements, barrier synchronization, reader/writer locks, spin locks, and persistent notification for message queues |
| 1003.21 | Distributed Real-time | buffer management, send control blocks, asynchronous and synchronous operations, bounded blocking, message priorities, message labels, and implementation protocols |

© 2019 Univ. K. Zinner, The Austrian National University

Page 220 of 360 | Chapter 1: Introduction & Languages | up to page 210

Introduction & Languages

POSIX - other languages

POSIX is a 'C' standard...

... but bindings to other languages are also (suggested) POSIX Standards.

- Ada:** 1003.5*, 1003.24
(some PAR approved only, some withdrawn, some (partly) implemented)
- Fortran:** 1003.9 (6/92)
- Fortran90:** 1003.19 (withdrawn)
- ... and there are POSIX standards for task-specific POSIX profiles, e.g.:
- Super computing: 1003.10 (6/05)
- Realtime: 1003.13, 1003.13b - profiles 51-54; combinations of the above RT-relevant POSIX standards. ⇒ RT-Linux
- Embedded Systems: 1003.13a (PAR approved only)

© 2019 Univ. K. Zinner, The Austrian National University

Page 221 of 360 | Chapter 1: Introduction & Languages | up to page 210

Introduction & Languages

PEarl

Process and Experiment Automation Realtime Language

PROBLEM:

```

SYSTEM: Alert: Hard_Int (7):

```

- Established standard.
- Compilers available for all major OSs (and some RT-OSs).
- as well as for a number of single-board systems (one free compiler for academic users).
- Used for educational purposes mostly.
- A configuration part ("SYSTEM") allows for hardware migration.
- Synchronization primitives on the level of semaphores.
- Designed for small scale engineering applications.

Currently maintained by a German special-interest community and one company (EPF).

© 2019 Univ. K. Zinner, The Austrian National University

Page 218 of 360 | Chapter 1: Introduction & Languages | up to page 210

Introduction & Languages

POSIX - 1003.1b/c

Frequently employed POSIX features include:

- Threads:** a common interface to threading - differences to 'classical UNIX' processes'
- Timers:** delivery is accomplished using POSIX signals
- Priority scheduling:** fixed priority, 32 priority levels
- Real-time signals:** signals with multiple levels of priority
- Semaphore:** named semaphore
- Memory queues:** message passing using named queues
- Shared memory:** memory regions shared between multiple processes
- Memory locking:** no virtual memory swapping of physical memory pages

© 2019 Univ. K. Zinner, The Austrian National University

Page 219 of 360 | Chapter 1: Introduction & Languages | up to page 210

Introduction & Languages

POSIX - example: setting a timer (cont.)

```

void timer_create(int num_secs, int num_nsecs)
{
    struct sigaction sa;
    sa.sa_handler = timer_intr;
    sa.sa_flags = SA_INTERRUPT | SA_NORM;
    if (sigaction(SIGRTMIN, &sa, NULL) < 0)
        perror("sigaction");
    if (timer_create(&timer_h, 0, &num_secs,
                    &num_nsecs) < 0)
        perror("timer_create");
    if (timer_settime(timer_h, 0, &tmr_setting, NULL) < 0)
        perror("timer_settime");
    if (timer_gettime(timer_h, &tmr_setting) < 0)
        perror("timer_gettime");
}

```

© 2019 Univ. K. Zinner, The Austrian National University

Page 220 of 360 | Chapter 1: Introduction & Languages | up to page 210

Introduction & Languages

POSIX

Portable Operating System Interface for Unix

- IEEE/ANSI Std 1003.1 and following.
- Library Interface (API)
- [C language calling conventions – types exit mostly in terms of (open) lists of pointers and integers with overloaded meanings].
- More than 30 different POSIX standards (and growing / changing).
- ⇒ a system is '100% POSIX compliant' if it implements parts of one of them!
- IF a system is '100% POSIX compliant', if it implements one of them!

© 2019 Univ. K. Zinner, The Austrian National University

Page 219 of 360 | Chapter 1: Introduction & Languages | up to page 210

Introduction & Languages

POSIX - support by different operating systems

| | POSIX 1003.1 (Base POSIX) | POSIX 1003.1b (Real-time extensions) | POSIX 1003.1c (Threads) |
|----------|---|--|-------------------------------------|
| Solaris | Full support | Full support | Full support |
| IRIX | Conformant | Full support | Full support |
| lynxOS | Conformant | Full support | Conformant |
| QNX | Full support | Partial support (no memory locking) | Full support |
| Neutrino | Full support | Partial support (no memory locking) | Full support |
| Linux | Full support | Partial support (no memory locking, no message queues) | Full support |
| VxWorks | Partial support (different process model) | Partial support (different process model) | Support through third party add-ons |

© 2019 Univ. K. Zinner, The Austrian National University

Page 220 of 360 | Chapter 1: Introduction & Languages | up to page 210

Introduction & Languages

POSIX - example: setting a timer

```

/* set the initial expiration and frequency of timer */
time_Setting.it_value.tv_sec = 1;
time_Setting.it_value.tv_nsec = 0;
time_Setting.it_interval.tv_sec = 0;
time_Setting.it_interval.tv_nsec = num_secs;
if ((timer_Setting_it_interval.tv_sec = num_secs,
    timer_Setting_it_interval.tv_nsec = num_nsecs,
    timer_Setting_it_interval.tv_nsec < 0))
    perror("timer_Setting_it_interval");
/* wait for signals */
sigemptyset(&sa.sigmask);
while (1) {
    if (sigsuspend(&sa.sigmask) == -1)
        /* routine that is called when timer expires */
        void timer_intr(sig, extra, void *scrut);
    if (timer_Setting_it_interval.tv_sec > 0)
        /* create timer, which uses the REALTIME_CLOCK */
        /* If (timer_Setting_it_interval.tv_nsec < 0)
            perror("timer_Setting_it_interval");
        if (timer_Setting_it_interval.tv_nsec < 0)
            perror("timer_Setting_it_interval");
    */
}

```

© 2019 Univ. K. Zinner, The Austrian National University

Page 221 of 360 | Chapter 1: Introduction & Languages | up to page 210

Introduction & Languages

POSIX - example: setting a timer (cont.)

```

/* set the initial expiration and frequency of timer */
tmr_setting.it_value.tv_sec = 1;
tmr_setting.it_value.tv_nsec = 0;
tmr_setting.it_interval.tv_sec = num_secs;
tmr_setting.it_interval.tv_nsec = num_secs;
if ( timer_settime(timer_id, &tmr_setting, NULL) < 0 )
    perror("setting timer");
/* wait for signals */
while (1) {
    /* Compare to peak: AFTER 30 MIN ALL 5 MIN DURING 1 HRS ACTIVATE Help;
    sleep(5);
}
/* routine that is called when timer expires */
void timer_int(int sig, siginfo_t *extra, void *craf)
{
    /* perform periodic processing and then exit */
}

```

© 2015 Ian R. Zelonek, The Australian National University

Page 227 of 363 | Chapter 1: Introduction & Languages | Up to page 228

229

Introduction & Languages

Real-Time Programming Languages

Languages mentioned so far

- Ada (Ada2012) General workhorse.
- Real-Time Java (RealTime Specification for Java 1.1) (Very) soft real-time applications.
- Esterel 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 plant modelling.
- POSIX (POSIX 1003.1b, ...) The libraries of bare bone integers and semaphores.
- Assemblers / C The languages of bare bone words.

© 2015 Ian R. Zelonek, The Australian National University

Page 228 of 363 | Chapter 1: Introduction & Languages | Up to page 229

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.

© 2015 Ian R. Zelonek, The Australian National University

Page 228 of 363 | Chapter 1: Introduction & Languages | Up to page 229

230

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**

© 2015 Ian R. Zelonek, The Australian National University

Page 229 of 363 | Chapter 1: Introduction & Languages | Up to page 230

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.

© 2015 Ian R. Zelonek, The Australian National University

Page 229 of 363 | Chapter 1: Introduction & Languages | Up to page 230

231