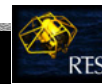


RES

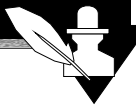


Introduction & Real-Time Languages

Uwe R. Zimmer – The Australian National University



Real-Time & Embedded Systems



References for this chapter

[Berry99] 

Gérard Berry
The Esterel v5 Language Primer
(Version 5.21 release 2.0)

Technical report: centre de Mathématiques
Appliquées, Ecole des Mines and INRIA

[Bollella01] 


Greg Bollella, Ben Brosgol, Steve Furr, David
Hardin, Peter Dibble, James Gosling, Mark
Turnbull & Rudy Belliardi
The Real-Time Specification for Java
<http://www.rti.org>

[Burns01]

Alan Burns and Andy Wellings
Real-Time Systems and Programming Languages
Addison Wesley, third edition, 2001

[Cohen96]

Norman H. Cohen
Ada as a second language
McGraw-Hill series in computer science, 2nd
edition

[GI98] 

eds. GI-working group 4.4.2
PEARL 90, Language report, Version 2.2
Technical report GI

all references and some links are available on the course page



Real-Time & Embedded Systems



Features of a Real-Time System?

- Fast context switches?
- Small size?
- Quick responds to external interrupts?
- Multitasking?
- 'low level' programming interfaces?
- Interprocess communication tools?
- High processor utilization?



Real-Time & Embedded Systems



Features of a Real-Time System?

- ~~Fast context switches?~~ **should be fast anyway**
- ~~Small size?~~ **should be small anyway**
- ~~Quick responds to external interrupts?~~ **not 'quick', but predictable**
- ~~Multitasking?~~ **real time systems are often multitasking systems**
- ~~'low level' programming interfaces?~~ **needed in many applications**
- ~~Interprocess communication tools?~~ **Feature of any distributed system**
- ~~High processor utilization?~~ **Fault tolerance builds on redundancy!**



Features of a Real-Time System

The correctness of a real-time systems depends on:

1. the logical correctness of the results as well as
2. **the time the result was delivered**

according to the specification.

☞ All results are to be delivered **just-in-time** – not too early, not too late.

Timing constraints are specified in many different ways ...

... often as a response to 'external' events ☞ reactive systems



Typical Real-Time Systems

- System sizes vary from traffic light controllers, or heating regulators to big aircrafts (Boeing, Airbus), satellites, or high speed trains (TGV)
- High degree of concurrency
- Close connections to real-world entities (sensors, actuators)
- Often a part of a real-world device (embedded systems)
- Failures may often lead to loss of life, or environmental damages.

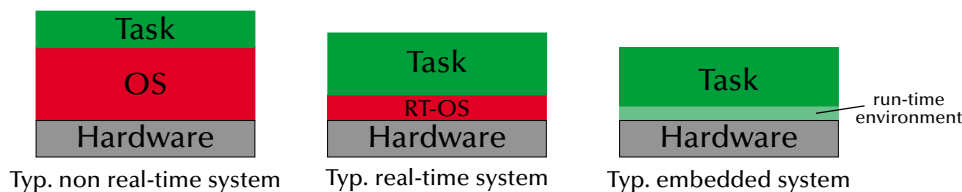
☞ **Predictability is more important than any other criterion**



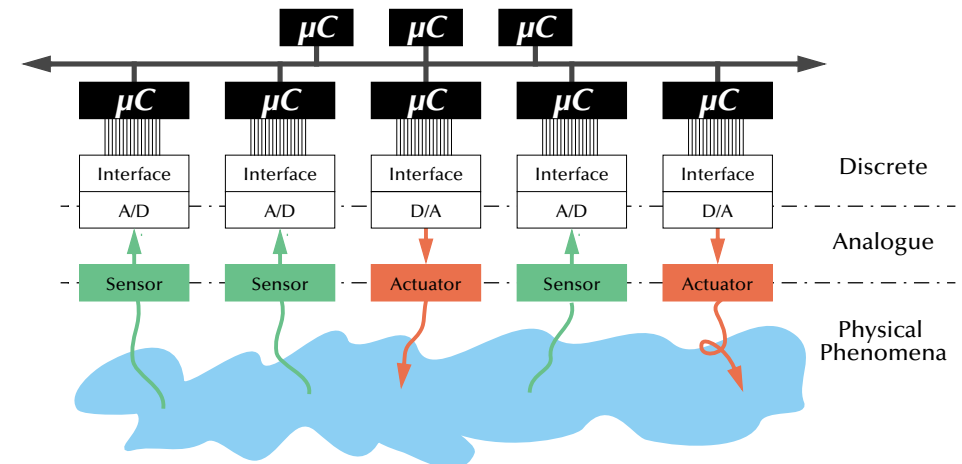
Typical Real-Time Operating Systems

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

☞ RT-OSs have the smallest possible impact on the timing behaviour

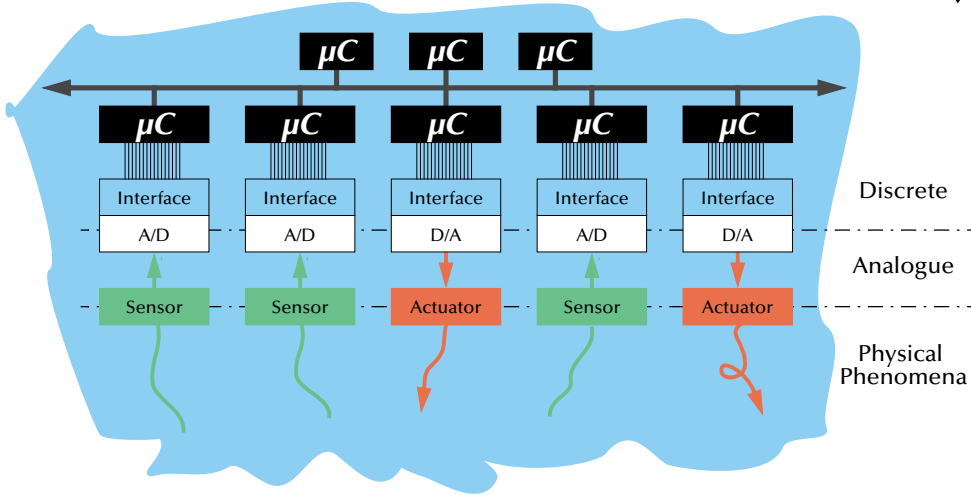


Real-Time Systems Components





Real-Time Systems ↔ Embedded Systems



what is this course about?

Parts of a Real-Time System

1. A set of physical processes
2. **Physical sensors** transforming all values into analogue voltages
3. Physical actuators delivering forces, lights, temperatures, etc. pp.
4. **D/A and A/D convertors** synchronize and discretize all voltages
5. **Interfaces** and buses feeding the measurements into a computer
6. One or many **controllers** (computers)
7. **Algorithms, Languages & Real-Time Software**



Programming styles

- **Imperative** (sequential) ↔ Ada, RT-JAVA, Eiffel, ...
- **Functional** (recursive) ↔ Lisp, OCaml, ...
- **Declarative** (logic) ↔ Prolog, ...
- **Data-flow machines** ↔ Lustre, Signal, ...
- (hierarchical) **Finite state machines** ↔ synchronous languages: Esterel, syncEifel, synERJY, ...

Real-Time styles

Imperative ↔ Functional ↔ Declarative ↔ Data-flow ↔ Finite state machines
 Static ↔ Dynamic
 Modular ↔ Concurrent ↔ Distributed
Synchronous ↔ **Continuous time**
Control oriented ↔ **Data oriented**



Programming styles

What makes a language suitable for real-time systems?

- **Predictability** ↔ no operations which will lead to unforeseeable timing behaviours (e.g. garbage collection)
- **Real-time!!** ↔ support for temporal scopes
- **Concurrency** ↔ support for tasking/threading
- **Distribution** ↔ support for message passing or rpc
- **Reliability** ↔ detect errors at compile-time or in the run-time environment
- **Large systems** ↔ scalable, modular, or object-oriented + separate compilation



Programming styles

Languages considered in this course

- Ada95 (used for assignments ↻ introduced first)
 - Esterel
 - Pearl
 - Real-time JAVA
 - POSIX
- ... others in places



Ada95

Ada95 is a **standardized** (ISO/IEC 8652:1995(E)) 'general purpose' language with **core** language primitives for

- strong typing, separate compilation (specification and implementation), object-orientation,
- concurrency, monitors, rpcs, timeouts, scheduling, priority ceiling locks
- strong run-time environments

... and **standardized** language-**annexes** for

- additional real-time features, distributed programming, system-level programming, numeric, informations systems, safety and security issues.



Ada95

A crash course

... refreshing:

- specification and implementation (body) parts, basic types
- exceptions
- information hiding in specifications ('private')
- generic programming
- class-wide programming ('tagged types')
- monitors and synchronisation ('protected', 'entries', 'selects', 'accepts')
- abstract types and dispatching



Ada95

Basics

... introducing:

- specification and implementation (body) parts
- constants
- some basic types (integer specifics)
- some type attributes
- parameter specification



A simple queue specification

```

package Queue_Pack_Simple is
  QueueSize : constant Positive := 10;
  type Element is new Positive range 1_000..40_000;
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Elements : List;
  end record;
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
end Queue_Pack_Simple;

```



A simple queue implementation

```

package body Queue_Pack_Simple is
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free - 1;
  end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top - 1;
  end Dequeue;
end Queue_Pack_Simple;

```



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); -- will produce an unpredictable result!
end Queue_Test_Simple;

```



Ada95

Exceptions

... introducing:

- exception handling
- enumeration types
- functional type attributes



A queue *specification* with proper exceptions

```

package Queue_Pack_Exceptions is
  QueueSize : constant Integer := 10;
  type Element is (Up, Down, Spin, Turn);
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is record
    Top, Free : Marker      := Marker'First;
    State      : Queue_State := Empty;
    Elements   : List;
  end record;
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  Queueoverflow, Queueunderflow : exception;
end Queue_Pack_Exceptions;

```



A queue *implementations* with proper exceptions

```

package body Queue_Pack_Exceptions is
  procedure Enqueue (Item: in 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'Pred (Queue.Free);
    Queue.State := Filled;
  end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Marker'Pred (Queue.Top);
    if Queue.Top = Queue.Free then Queue.State := Empty; end if;
  end Dequeue;
end Queue_Pack_Exceptions;

```



A queue *test program* with proper exceptions

```

with Queue_Pack_Exceptions; use Queue_Pack_Exceptions;
with Ada.Text_IO;          use Ada.Text_IO;
procedure Queue_Test_Exceptions is
  Queue : Queue_Type;
  Item : Element;
begin
  Enqueue (Turn, 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_Exceptions;

```



Ada95

Information hiding (private parts)

... introducing:

- private \Leftarrow assignments and comparisons are allowed
- limited private \Leftarrow entity cannot be assigned or compared



A queue *specification* with proper information hiding

```

package Queue_Pack_Private is
  QueueSize : constant Integer := 10;
  type Element is new Positive range 1..1000;
  type Queue_Type is limited private;

  procedure Enqueue (Item: in Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

  Queueoverflow, Queueunderflow : exception;

private
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    State : Queue_State := Empty;
    Elements : List;
  end record;
end Queue_Pack_Private;

```



A queue *implementation* with proper information hiding

```

package body Queue_Pack_Private is
  procedure Enqueue (Item: in 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 := Queue.Free + 1;
    Queue.State := Filled;
  end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    if Queue.Top = Queue.Free then Queue.State := Empty; end if;
  end Dequeue;
end Queue_Pack_Private;

```



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 not be limited type
  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_Private;

```



Ada95

Generic packages

... 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: in Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

  Queueoverflow, Queueunderflow : exception;

private
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    State : Queue_State := Empty;
    Elements : List;
  end record;
end Queue_Pack_Generic;

```



A generic queue *implementation*

```

package body Queue_Pack_Generic is
  procedure Enqueue (Item: in 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 := Queue.Free - 1;
    Queue.State := Filled;
  end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    if Queue.Top = Queue.Free then Queue.State := Empty; end if;
  end Dequeue;
end Queue_Pack_Generic;

```



A generic queue *test program*

```

with Queue_Pack_Generic;
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;

  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;

```



Ada95

Object oriented programming I

... introducing:

- tagged types ^{ESR} the Ada-way to say that this type can be extended
- derivation of tagged types
- method overwriting
- usage of parent entities



An open queue base class *specification*

```

package Queue_Pack_Object_Base is
  QueueSize : constant Integer := 10;
  type Element is new Positive range 1..1000;
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is tagged record
    Top, Free : Marker      := Marker'First;
    State     : Queue_State := Empty;
    Elements  : List;
  end record;

  procedure Enqueue (Item: in Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

  Queueoverflow, Queueunderflow : exception;

end Queue_Pack_Object_Base;

```



An open queue base class *implementation*

```

package body Queue_Pack_Object_Base is
  procedure Enqueue (Item: in 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 := Queue.Free - 1;
    Queue.State := Filled;
  end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    if Queue.Top = Queue.Free then Queue.State := Empty; end if;
  end Dequeue;

end Queue_Pack_Object_Base;

```



A derived open queue class *specification*

```

with Queue_Pack_Object_Base; use Queue_Pack_Object_Base;
package Queue_Pack_Object is
  type Ext_Queue_Type is new Queue_Type with record
    Reader      : Marker      := Marker'First;
    Reader_State : Queue_State := Empty;
  end record;

  procedure Enqueue (Item: in Element; Queue: in out Ext_Queue_Type);
  procedure Read_Queue (Item: out Element; Queue: in out Ext_Queue_Type);

end Queue_Pack_Object;

```



A derived open queue class *implementation*

```

package body Queue_Pack_Object is
  procedure Enqueue (Item: in Element; Queue: in out Ext_Queue_Type) is
  begin
    Enqueue (Item, Queue_Type (Queue));
    Queue.Reader_State := Filled;
  end Enqueue;

  procedure Read_Queue (Item: out Element; Queue: in out Ext_Queue_Type) is
  begin
    if Queue.Reader_State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Reader);
    Queue.Reader := Queue.Reader + 1;
    if Queue.Reader = Queue.Free then Queue.Reader_State := Empty; end if;
  end Read_Queue;

end Queue_Pack_Object;

```



An open class test program

```

with Queue_Pack_Object_Base; use Queue_Pack_Object_Base;
with Queue_Pack_Object;      use Queue_Pack_Object;
with Ada.Text_IO;            use Ada.Text_IO;

procedure Queue_Test_Object is
  Queue : Ext_Queue_Type;
  Item : Element;
begin
  Enqueue (Item => 1, Queue => Queue);
  Read_Queue (Item, Queue);
  Enqueue (Item => 5, Queue => Queue);
  Dequeue (Item, 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_Object;

```



Ada95

Object oriented programming II

... introducing:

- private tagged types
- objects which are protected against their children also



An encapsulated queue base class *specification*

```

package Queue_Pack_Object_Base_Private is
  QueueSize : constant Integer := 10;
  type Element is new Positive range 1..1000;
  type Queue_Type is tagged limited private;

  procedure Enqueue (Item: in Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

  Queueoverflow, Queueunderflow : exception;

private
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is tagged limited record
    Top, Free : Marker := Marker'First;
    State : Queue_State := Empty;
    Elements : List;
  end record;
end Queue_Pack_Object_Base_Private;

```



An encapsulated queue base class *implementation*

```

package body Queue_Pack_Object_Base_Private is
  procedure Enqueue (Item: in 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 := Queue.Free - 1;
    Queue.State := Filled;
  end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top - 1;
    if Queue.Top = Queue.Free then Queue.State := Empty; end if;
  end Dequeue;
end Queue_Pack_Object_Base_Private;

```



A derived encapsulated queue class *specification*

```
with Queue_Pack_Object_Base_Private; use Queue_Pack_Object_Base_Private;
package Queue_Pack_Object_Private is
  type Ext_Queue_Type is new Queue_Type with private;
  subtype Depth_Type is Positive range 1..QueueSize;
  procedure Look_Ahead (Item: out Element;
    Depth: in Depth_Type; Queue: in out Ext_Queue_Type);
private
  type Ext_Queue_Type is new Queue_Type with null record;
end Queue_Pack_Object_Private;
```



A derived encapsulated queue class *implementation*

```
package body Queue_Pack_Object_Private is
  procedure Look_Ahead (Item: out Element;
    Depth: in Depth_Type; Queue: in out Ext_Queue_Type) is
    Storage : Queue_Type;
    ShuffleItem : Element;
  begin
    for I in 1..Depth - 1 loop
      Dequeue (ShuffleItem, Queue);
      Enqueue (ShuffleItem, Storage);
    end loop;
    Dequeue (Item, Queue);
    Enqueue (Item, Storage);
  (...)
end;
```



```
(...)
Read_The_Rest:
  begin
    for I in 1..QueueSize - Depth loop
      Dequeue (ShuffleItem, Queue);
      Enqueue (ShuffleItem, Storage);
    end loop;
  exception
    when Queueunderflow => null; -- read the rest is done
  end Read_The_Rest;
Restore_The_Queue:
  begin
    for I in 1..QueueSize loop
      Dequeue (ShuffleItem, Storage);
      Enqueue (ShuffleItem, Queue);
    end loop;
  exception
    when Queueunderflow => null; -- restore is done
  end Restore_The_Queue;
end Look_Ahead;
end Queue_Pack_Object_Private;
```



An encapsulated class test program

```
with Queue_Pack_Object_Base_Private; use Queue_Pack_Object_Base_Private;
with Queue_Pack_Object_Private; use Queue_Pack_Object_Private;
with Ada.Text_IO; use Ada.Text_IO;
procedure Queue_Test_Object_Private is
  Queue : Ext_Queue_Type;
  Item : Element;
begin
  Enqueue (Item => 1, Queue => Queue);
  Enqueue (Item => 1, Queue => Queue);
  Look_Ahead (Item => Item, Depth => 2, Queue => Queue);
  Enqueue (Item => 5, Queue => Queue);
  Dequeue (Item, Queue);
  Dequeue (Item, Queue);
  Dequeue (Item, 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_Object_Private;
```

... introducing:

- protected types
- tasks (definition, instantiation and termination)
- task synchronisation
- entry guards
- entry calls
- accept and selected accept statements

```
package body Queue_Pack_Protected is
  protected body Protected_Queue is
    entry Enqueue (Item: in Element) when
      Queue.State = Empty or Queue.Top /= Queue.Free is
    begin
      Queue.Elements (Queue.Free) := Item;
      Queue.Free := Queue.Free - 1;
      Queue.State := Filled;
    end Enqueue;

    entry Dequeue (Item: out Element) when
      Queue.State = Filled is
    begin
      Item := Queue.Elements (Queue.Top);
      Queue.Top := Queue.Top - 1;
      if Queue.Top = Queue.Free then Queue.State := Empty; end if;
    end Dequeue;

  end Protected_Queue;
end Queue_Pack_Protected;
```

```
Package Queue_Pack_Protected is
  QueueSize : constant Integer := 10;
  subtype Element is Character;
  type Queue_Type is limited private;

  Protected type Protected_Queue is
    entry Enqueue (Item: in Element);
    entry Dequeue (Item: out Element);

  private
    Queue : Queue_Type;

  end Protected_Queue;

private
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    State : Queue_State := Empty;
    Elements : List;
  end record;
end Queue_Pack_Protected;
```

```
with Queue_Pack_Protected; use Queue_Pack_Protected;
with Ada.Text_IO; use Ada.Text_IO;

procedure Queue_Test_Protected is
  Queue : Protected_Queue;

  task Producer is entry shutdown; end Producer;
  task Consumer is end Consumer;

  task body Producer is
    Item : Element;
    Got_It : Boolean;
  begin
    loop
      select
        accept shutdown; exit; -- main task loop
      else
        Get_Immediate (Item, Got_It);
        if Got_It then
          Queue.Enqueue (Item); -- task might be blocked here!
        else
          delay 0.1; --sec.
        end if;
      end select;
    end loop;
  end Producer;

  (...)
end Queue_Test_Protected;
```



A multitasking protected queue test program (cont.)

```

(...)
task body Consumer is
  Item : Element;
begin
  loop
    Queue.Dequeue (Item); -- task might be blocked here!
    Put ("Received: "); Put (Item); Put_Line ("!");
    if Item = 'q' then
      Put_Line ("Shutting down producer"); Producer.Shutdown;
      Put_Line ("Shutting down consumer"); exit; -- main task loop
    end if;
  end loop;
end Consumer;

begin
  null;
end Queue_Test_Protected;

```



Ada95

Abstract types & dispatching

... introducing:

- abstract tagged types
- abstract subroutines
- concrete implementation of abstract types
- dispatching to different packages, tasks, and partitions according to concrete types



An abstract queue specification

```

package Queue_Pack_Abstract is
  subtype Element is Character;
  type Queue_Type is abstract tagged limited private;

  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
    abstract;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
    abstract;

private
  type Queue_Type is abstract tagged limited null record;
end Queue_Pack_Abstract;

```



A concrete queue specification

```

with Queue_Pack_Abstract; use Queue_Pack_Abstract;
package Queue_Pack_Concrete is
  QueueSize : constant Integer := 10;
  type Real_Queue is new Queue_Type with private;

  procedure Enqueue (Item: in Element; Queue: in out Real_Queue);
  procedure Dequeue (Item: out Element; Queue: in out Real_Queue);

  Queueoverflow, Queueunderflow : exception;

private
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Real_Queue is new Queue_Type with record
    Top, Free : Marker := Marker'First;
    State : Queue_State := Empty;
    Elements : List;
  end record;
end Queue_Pack_Concrete;

```



A concrete queue implementation

```

package body Queue_Pack_Concrete is
  procedure Enqueue (Item: in Element; Queue: in out Real_Queue) is
  begin
    if Queue.State = Filled and Queue.Top = Queue.Free then
      raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free - 1;
    Queue.State := Filled;
  end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Real_Queue) is
  begin
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    if Queue.Top = Queue.Free then Queue.State := Empty; end if;
  end Dequeue;
end Queue_Pack_Concrete;

```



A multitasking dispatching test program

```

with Queue_Pack_Abstract; use Queue_Pack_Abstract;
with Queue_Pack_Concrete; use Queue_Pack_Concrete;

procedure Queue_Test_Dispatching is
  type Queue_Class is access all Queue_Type'class;

  task Queue_Holder is -- could be on an individual partition
    entry Queue_Filled;
  end Queue_Holder;

  task Queue_User is -- could be on an individual partition
    entry Send_Queue (Remote_Queue: in Queue_Class);
  end Queue_User;

  (...)

```



```

task body Queue_Holder is
  Local_Queue : Queue_Class;
  Item : Element;
begin
  Local_Queue := new Real_Queue; -- could be a different implementation!
  Queue_User.Send_Queue (Local_Queue);
  accept Queue_Filled do
    Dequeue (Item, Local_Queue.all); -- Item will be 'r'
  end Queue_Filled;
end Queue_Holder;

task body Queue_User is
  Local_Queue : Queue_Class;
  Item : Element;
begin
  Local_Queue := new Real_Queue; -- could be a different implementation!
  accept Send_Queue (Remote_Queue: in Queue_Class) do
    Enqueue ('r', Remote_Queue.all); -- potentially a rpc!
    Enqueue ('l', Local_Queue.all);
  end Send_Queue;
  Queue_Holder.Queue_Filled;
  Dequeue (Item, Local_Queue.all); -- Item will be 'l'
end Queue_User;

begin null; end Queue_Test_Dispatching;

```



Ada95

Ada95 language status

- Established language standard with free and commercial compilers available for all major OSs.
- Stand-alone runtime environments for embedded systems (some are only available commercially).
- Special (yet non-standard) extensions (i.e. language reductions and proof systems) for extreme small footprint embedded systems or high integrity real-time environments available in Ravenscar profile systems.
- has been used and is in use in numberless large scale projects (e.g. in the international space station, and in some spectacular crashes: e.g. Ariane 5)



Transformational ↔ Interactive ↔ Reactive

- **Transformational (functional) systems:**

... generating outputs based on input and stop, utilizing no or a small number of internal states.

- **Interactive systems:**

... servers and other systems in longer 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 reasonable reaction times.



Strong synchrony or 'zero delay' assumption:

- ☞ theoretical perspective: **all operations are 'instantaneous'**

... or a little bit more realistic:

- ☞ there is no *observable* delay,
i.e. **all operations are finished before the next signal occurs.**



Control- ↔ Data-handling

- **Data-handling:**

... continuous data-streams, functional processing (DSPs, 'number crunching'),

- ☞ high bandwidth

- **Control-handling:**

... discrete signals, controlling data-streams and processes,

- ☞ low bandwidth

Control-dominated reactive systems

- **Real-Time process control:**

... reaction to (sparse) stimuli in predefined time-spans

- **Embedded systems / device control:**

... local, discrete control

- **Complex systems control:**

... supervision of complex data-streams

- **Communication protocols:**

... control part of communication systems

- **Human-machine interface:**

... switching modes, event handling

- **Control logic (hardware):**

... glue logic, interfaces, pipe control

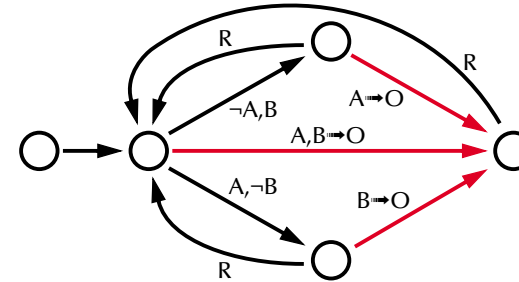
Determinism

- Control-dominated reactive systems \Rightarrow **mostly deterministic**
... e.g. real-time & embedded systems
- Interactive systems \Rightarrow **usually non-deterministic**
... e.g. operating systems, IP-servers

\Rightarrow is determinism lost automatically in concurrent systems?

a simple reactive pure-signal example

Mealy machine



Module in Esterel:

```

module A_and_B_gives_0;
input A, B, R;
output O;
loop
  [await A || await B];
  emit O;
  each R;
end module;
  
```

a simple reactive integrator example

Specification: a module should count the number of metres per second and emit this number as 'speed' once per second.

```

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

\leftarrow these are exclusive
 \leftarrow hard aborted and restarted with every 'Metre' signal
 \leftarrow above block is hard aborted with every 'Second' signal

Immediate reactions:

by default all synchronization points:

```

await <signal>;
abort ... when <signal>;
every <signal> do ... end every;
loop ... each <signal>;
  
```

wait for the **next** signal occurrence ('rising edge trigger').
... but with an additional 'immediate':

```

await immediate <signal>;
abort ... when immediate <signal>;
every immediate <signal> do ... end every;
loop ... each immediate <signal>;
  
```

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

Weak aborts:

by default a code block is aborted immediately, when a signal occurs:

```
abort
  [ <statement>; ]+
when <signal>;
```

Sometimes the semantic like

'activate the code block for one last time, when <signal> occurs' is more useful and expressed in Esterel as:

```
weak abort
  [ <statement>; ]+
when <signal>;
```

the code block is now activated for a 'final wish', when <signal> occurs.

Parallel signals

assuming 'Metre' and 'Second' occur simultaneously **this code is wrong!**

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

these are no longer exclusive

2. 'every Metre' waits for the next 'Metre' **1 metre is lost**

1. above block is immediately aborted

Parallel signals

'Metre' and 'Second' occur simultaneously – using 'weak abort':

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

these are no longer exclusive

above block is activated one last time, when 'Second' occurs. Therefore the simultaneous 'Metre' is taken into account

Parallel signals

'Metre' and 'Second' occur simultaneously – using 'every immediate':

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

these are no longer exclusive

2. 'every immediate' takes the current 'Metre' into account

1. above block is immediately aborted

Causality and synchronous languages

Causality in general terms: 'the future should not influence the past'

Technically:

Causal synchronous programs need to be

- ☛ **Reactive**
provides a well-defined output for each signal sequence
- ☛ **Deterministic**
provides exactly one output for each signal sequence

Causality and synchronous languages

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

Causality: counter examples

☛ non-reactive programming:

```
module non-reactive;
  output 0;
  present 0 else emit 0 end;
end module;
```

☛ non-deterministic programming:


```
module non-deterministic;
  output 0;
  present 0 then emit 0 end;
end module;
```

☛ cyclic dependencies with multiple signals:

```
module cyclic_dependency;
  output A, B;
  [ present A then emit B end || present B else emit A end ]
end module;
```

More on 'strong synchrony' and 'zero delay' assumptions:

Assuming a system operates in three phases:

- 
1. collect current input signals
 2. calculate responses
 3. emit new output signals and goto step 1.

☛ The system is assumed to be by '**synchronous**' or '**instantaneous**', iff the total worst case computation time is smaller than the minimal time between two observable changes in the environment.

☛ Synchronous systems assume a **logical** rather than a continuous **time**.



More on 'strong synchrony' and 'zero delay' assumptions:

In many applications these assumptions are justified and enable:

- a strong analysis and simplification theory (Boolean calculus and automata theory)
- a significantly easier program verification
- an easier hardware implementation



Esterel language status

- Created by the Esterel group at INRIA, France.
- Available freely for Linux, Solaris, and Windows NT.
- Produces C/C++ code, which need to be cross-compiled for an actual target system.
- Currently maintained by Esterel Technologies (a spin-off company).
- No standards.
- Employed in telecommunication, automotive, energy, aerospace, and defence projects by some major companies.



Process and Experiment Automation Realtime Language

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

Is a settled standard:

- DIN 66253-1: Basic PEARL 1981
- DIN 66253-2: Full PEARL 1982
- DIN 66253-3: PEARL for distributed systems 1989



```

MODULE;
SYSTEM;
    Alert: Hard_Int(?);
PROBLEM;
    SPECIFY Alert  INTERRUPT;
    SPECIFY Help  TASK GLOBAL;
    SPECIFY Pushed BIT GLOBAL;
    DECLARE Switch BIT INITIAL 0;
    Init      : TASK MAIN;
                WHEN Alert ACTIVATE Recovery;
                ENABLE Alert;
                Switch := Pushed;
    END;
    Recovery: TASK PRIORITY 9;
                DISABLE Alert;
                IF Switch = 1 THEN ACTIVATE Help; FIN;
                AFTER 30 MIN ALL 5 MIN DURING 1 HRS ACTIVATE Help;
    END;
MODEND;

```

PEARL – language status

- 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 mainly.
- Currently maintained by a German special-interest community and a small company (IEP).

Specific Java engines and classes enhance:

- **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, consequent object orientation.
- ☞ some restrict the language standard, some extend it.

*Real-Time Specification for Java 1.0**(final 11/01 – currently no release date for 1.0.1)*

- enhanced thread model (memory attributes, more precise specs)
- enabling powerful and highly adaptive scheduling policies
- introducing scoped, immortal, and physical memory to Java
- introducing timers, interrupts, and more exceptions
- higher resolution time model
- optional support for POSIX signals

*Real-Time Specification for Java 1.0**(final 11/01 – currently no release date for 1.0.1)*

- ☞ **Backward compatible:** offers the full standard Java specification, no syntactical extensions
- ☞ Allows for **different** Java-engine implementations:
 - in terms of completeness: e.g. scheduling is not mandatory
 - in terms of interpretations: e.g. 'instantiations per time-span' (RationalTime) strongly suggests but does not enforce equal distance internals

currently one reference implementation available (for TimeSys Linux)

RT-Java – Language status

- RT-Java is still a consequently object-oriented language.
- Garbage collection can be restricted or even fully suppressed (mandatory requirement for predictable systems).
- ☞ How do you program in a clean object oriented manner without garbage collection?
- ☞ Using it in hard-realtime environments implies to ‘program badly’ (in terms of strong OOP).

Many potential applications in soft- or mixed-realtime environments.

Portable Operating System Interface for Computing Environments

- IEEE/ANSI Std 1003.1 and following
- Program Interface (API) [C Language]
- more than 30 different POSIX standards
(a system is ‘POSIX compliant’, if it implements parts of just one of them!)

| | | |
|------------------|---------------------------------|---|
| 1003.1 12/01 | 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, FIFO, ... |
| 1003.1b 10/93 | Real-time Extensions | real-time signals, priority scheduling, timers, asynchronous I/O, prioritized I/O, synchronized I/O, file sync, mapped files, memory locking, memory protection, message passing, semaphore, ... |
| 1003.1c 6/95 | Threads | multiple threads within a process; includes support for: thread control, thread attributes, priority scheduling, mutexes, mutex priority inheritance, mutex priority ceiling, and condition variables |
| 1003.1d 10/99 | Additional Real-time Extensions | new process create semantics (spawn), sporadic server scheduling, execution time monitoring of processes and threads, I/O advisory information, timeouts on blocking functions, device control, and interrupt control |
| 1003.1j 1/00 | Advanced Real-time Extensions | typed 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 |

Frequently found POSIX RT-features include:

- **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

POSIX – support in some OSs

| | 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 (Version 3.1) |
| QNX Neutrino | Full support | Partial support (no memory locking) | Full support |
| Linux | Full support | Partial support (no timers, no message queues) | Full support |
| VxWorks | Partial support (different process model) | Partial support (different process model) | Supported through third party product |

POSIX – example: setting a timer

```
void timer_create(int num_secs, int num_nsecs)
{
    struct sigaction sa;
    struct sigevent sig_spec;
    sigset_t allsigs;
    struct itimerspec tmr_setting;
    timer_t timer_h;

    /* setup signal to respond to timer */
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = timer_intr;

    if (sigaction(SIGRTMIN, &sa, NULL) < 0)
        perror('sigaction');

    sig_spec.sigev_notify = SIGEV_SIGNAL;
    sig_spec.sigev_signo = SIGRTMIN;

```

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)
- 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 (6/95)
- **Realtime: 1003.13, 1003.13b** (3/98)
 - profiles 51-54: combinations of the above RT-relevant POSIX standards [⊗] RT-Linux
- **Embedded Systems: 1003.13a** (PAR approved only)

POSIX – example: setting a timer (cont.)

```
/* create timer, which uses the REALTIME clock */
if (timer_create(CLOCK_REALTIME, &sig_spec, &timer_h) < 0)
    perror('timer create');

/* 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_nsecs;
if (timer_settime(timer_h, 0, &tmr_setting, NULL) < 0)
    perror('settimer');

/* wait for signals */
sigemptyset(&allsigs);
while (1) {
    sigsuspend(&allsigs);
}

/* routine that is called when timer expires */
void timer_intr(int sig, siginfo_t *extra, void *cruft)
{
    /* perform periodic processing and then exit */
}

```

POSIX – example: setting a timer (cont.)

```

/* create timer, which uses the REALTIME clock */
if (timer_create(CLOCK_REALTIME, &sig_spec, &timer_h) < 0)
    perror('timer create');

/* 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_nsecs;
if ( timer_settime(timer_h, 0, &tmr_setting, NULL) < 0)
    perror('settimer');

/* wait for signals */
sigemptyset(&allSIGs);
while (1) {
    sigsuspend(&allSIGs);
}

/* routine that is called when timer expires */
void timer_intr(int sig, siginfo_t *extra, void *cruft)
{
    /* perform periodic processing and then exit */
}

```

remember the Pearl timers?
AFTER 30 MIN ALL 5 MIN DURING 1 HRS ACTIVATE He1p;

Real-Time Languages

Suitable for which real-time systems?

| | Ada | Esterel | Pearl | RT-Java | Posix |
|-----------------|---------------------------------|--------------------------------|----------------|--------------|---------|
| Predictability | *** (specific run-time env.) | *** (if logical time holds) | ** | --- (if OOP) | / |
| Real-Time | ** | logic time | ** | *** | ** |
| Concurrency | *** | ** | * | ** | *** |
| Distribution | ** | / | *(dist. Pearl) | *** | ** |
| Error detection | ** (strong typing) | *** (verification) | * | * | --- (C) |
| Large systems | *** | * | --- | *** | / |

Summary

Introduction & Real-Time Languages

- Features (and non-features) of a real-time system
- Components of a real-time system
- Real-time languages criteria
- Examples of actual real-time languages:
 - Ada95
 - Esterel
 - Pearl
 - Real-time JAVA
 - POSIX