Real-Time & Embedded Systems

what is offered here?

Overview, Perspectives, Paths, Methods, and some Theory

into/for/about Real-Time & Embedded Systems
who could be interested in this?

anybody who …

… would like to see immediate real-world involvement in his/her work

… would like to learn how to create predictability and fault-tolerant complex systems

… would like to know more about the usage of 95% of all µprocessors
who are these people? – introduction

The course will be given by

Uwe R. Zimmer

and

Pat Bernardi
how will this all be done?

Lectures:

- 2 lectures à 1.5h per week … all the nice stuff and theory
  Monday & Tuesday, 9:00-10:30; in MCC T4, PSYC G8 resp.

Laboratories:

- 2 hours per week … all the rough stuff and practice
  Monday 11:00-13:00 or Tuesday 13:00-15:00 –in CSIT N114, N113 resp.
  laboratory-enrolment: https://cs.anu.edu.au/streams/

Resources:

- introduced in the lectures and collected on the course page:
  … as well as schedules, slides, sources, etc. pp. … keep an eye on this page!

Assessment:

- exam at the end of the course (70%) plus laboratories performance (30%)
Topics in this course

1. Introduction & real-time languages
2. Physical coupling
3. Interfaces
4. Time & embodiment
5. Asynchronism
6. Synchronisation
7. Scheduling
8. Resource control
9. Reliability & fault-tolerance
Central textbook

[Burns01]
Alan Burns and Andy Wellings
*Real-Time Systems and Programming Languages*
Addison Wesley, third edition, 2001

Supporting literature

[Ari90]
M. Ben-Ari
*Principles of Concurrent and Distributed Programming*
Prentice Hall, 1990

[Cohen96]
Norman H. Cohen
*Ada as a second language*
McGraw-Hill series in computer science, 2nd edition

[Ada95RM] (on-line version available)
Ada Working Group
*Ada 95 Reference Manual – Language and Standard Libraries*

many more references and links are available on the course page
# Table of Contents

1. Introduction & Real-Time Languages  
   1.1. Features (and non-features) of a real-time system  
   1.2. Components of a real-time system  
   1.3. Real-time languages criteria  
   1.4. Examples of actual real-time languages:  
      - Ada95, Esterel, Pearl, Real-time JAVA, POSIX

2. Physical coupling  
   2.1. Physical phenomena  
   2.2. Measuring temperature  
       - Thermoelements, thermocouples, thermoresistors, thermistors, noise temperature measurement and others  
   2.3. Measuring range and relative speed  
       - Triangulation, time of flight, intensity, Doppler methods, interferometry

3. Converters & Interfaces  
   3.1. Analogue signal chain in a digital system  
       - Sampling, aliasing, Nyquist’s criterion, oversampling  
       - Quantization (LSB, rms noise voltage, SNR, ENOB) – Missing codes, DNL, INL  
   3.2. A/D Converters: flash, pipelined-flash, SAR, Σ-Δ, n-th order Σ-Δ  
   3.3. Examples:  
      - Fast and simple A/D converter example

4. Time & Space  
   4.1. What is time? / What is embodiment?  
   4.2. Interfacing with time  
       - Formulating local time-dependent constraints – Access time, delay processes, detect timeouts (in different languages)  
   4.3. Specifying timing requirements  
       - Formulating global timing-constraints – Understanding time-scope parameters (and expressing them in different languages)  
   4.4. Satisfying timing requirements  
       - Real-time logic and complex systems approach

5. Asynchronism  
   5.1. Interrupts / Signals  
       - Device / system / language / operating-system level interrupt control  
       - Characteristics of interrupts and signals  
   5.2. Exceptions  
       - Exception classes / granularity / parametrization / propagation – Resumption and termination, specific language issues

6. Synchronization  
   6.1. Shared memory based synchronization  
   6.2. Message based synchronization  
   6.3. Resource control

7. Scheduling  
   7.1. Basic real-time scheduling  
       - Fixed Priority Scheduling (FPS) with Rate Monotonic (RMOPO) Deadline Monotonic Priority Ordering (DMPO)  
       - Earliest Deadline First (EDF)  
   7.2. Real-world extensions  
       - Aperiodic, sporadic, soft real-time tasks – Deadlines shorter than period – Cooporative and deferred pre-emption scheduling

8. Reliability  
   9.1. Terminology  
   9.2. Faults  
   9.3. Redundancy  
   9.4. Reduce & Formalise  
     - Ada95 Ravenscar profile  
     - Real-time Logic

9. Language support  
   - Ada95, POSIX static, off-line analysis mostly – RT-Java on-line, dynamic scheduling
Introduction & Real-Time Languages

Uwe R. Zimmer – The Australian National University
**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.rtj.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
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?
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
generated by the system 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
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 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
... introducing:

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

```haskell
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;
```
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;
Information hiding (private parts)

... introducing:

- private assignments and comparisons are allowed
- limited private 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;
    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;
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

```ada
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;
```
Object oriented programming I

... introducing:

- tagged types • the Ada-way to say that this type can be extended
- derivation of tagged types
- method overwriting
- usage of parent entities
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;
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;
Object oriented programming II

... introducing:

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

            (...)
(...)

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, Queue);
    Enqueue (ShuffleItem, Storage);
  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

```ada
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); -- 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
A protected queue specification

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;
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;
A multitasking protected queue test program

```ada
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;

(...)```

© 2009 Uwe R. Zimmer, The Australian National University
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;
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
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;
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 SendQueue (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 - 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)
Esterel

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.
Esterel

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.
Esterel

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
Real-Time & Embedded Systems

Esterel

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
Esterel

Determinism

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

\(\triangleright\) is determinism lost automatically in concurrent systems?
Esterel

a simple reactive pure-signal example

Module in Esterel:

```esterel
module A_and_B_gives_0;
  input A, B, R;
  output 0;
  loop
    [await A || await B];
    emit 0;
  each R;
end module;
```
Esterel

**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.

```plaintext
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;
```

these are exclusive

hard aborted and restarted with every ‘Metre’ signal

above block is hard aborted with every ‘Second’ signal
Immediate reactions:

by default all synchronization points:

```plaintext
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’:

```plaintext
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:

```plaintext
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:

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

the code block is now activated for a ‘final wish’, when `<signal>` occurs.
Esterel

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

1. above block is immediately aborted

2. ‘every Metre’ waits for the next ‘Metre’ ☞ 1 metre is lost
Parallel signals

‘Metre’ and ‘Second’ occur simultaneously – using ‘weak abort’:

```plaintext
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;
```

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’:

```esterel
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;
```

1. above block is immediately aborted
2. ‘every immediate’ takes the current ‘Metre’ into account

these are no longer exclusive
Esterel

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
Esterel

Causality: counter examples

- non-reactive programming:
  ```
  module non-reactive;
  output O;
  present O else emit O end;
  end module;
  ```

- non-deterministic programming:
  ```
  module non-deterministic;
  output O;
  present O then emit O 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;
  ```
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.
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 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

*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.
PEARL

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
PEARL – Example

MODULE;
SYSTEM;
  Alert: Hard_Int(7);
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

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 Java

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 & Embedded Systems

Real-Time Java

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)
Real-Time Java

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.
POSIX

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!)
### POSIX – some of the real-time relevant standards

<table>
<thead>
<tr>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1003.1 12/01</strong></td>
</tr>
<tr>
<td><strong>OS Definition</strong></td>
</tr>
<tr>
<td>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, …</td>
</tr>
</tbody>
</table>

| **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

<table>
<thead>
<tr>
<th></th>
<th>POSIX 1003.1 (Base POSIX)</th>
<th>POSIX 1003.1b (Real-time extensions)</th>
<th>POSIX 1003.1c (Threads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solaris</td>
<td>Full support</td>
<td>Full support</td>
<td>Full support</td>
</tr>
<tr>
<td>IRIX</td>
<td>Conformant</td>
<td>Full support</td>
<td>Full support</td>
</tr>
<tr>
<td>LynxOS</td>
<td>Conformant</td>
<td>Full support</td>
<td>Conformant (Version 3.1)</td>
</tr>
<tr>
<td>QNX Neutrino</td>
<td>Full support</td>
<td>Partial support (no memory locking)</td>
<td>Full support</td>
</tr>
<tr>
<td>Linux</td>
<td>Full support</td>
<td>Partial support (no timers, no message queues)</td>
<td>Full support</td>
</tr>
<tr>
<td>VxWorks</td>
<td>Partial support (different process model)</td>
<td>Partial support (different process model)</td>
<td>Supported through third party product</td>
</tr>
</tbody>
</table>
**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)
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 – 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 */
}
## Real-Time Languages

### Suitable for which real-time systems?

<table>
<thead>
<tr>
<th></th>
<th>Ada</th>
<th>Esterel</th>
<th>Pearl</th>
<th>RT-Java</th>
<th>Posix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Predictability</strong></td>
<td>***</td>
<td>***</td>
<td>**</td>
<td>--- (if OOP)</td>
<td>/</td>
</tr>
<tr>
<td></td>
<td>(specific run-time env.)</td>
<td>(if logical time holds)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Real-Time</strong></td>
<td>**</td>
<td>logic time</td>
<td>**</td>
<td>***</td>
<td>**</td>
</tr>
<tr>
<td><strong>Concurrency</strong></td>
<td>***</td>
<td>**</td>
<td>*</td>
<td>**</td>
<td>***</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Distribution</strong></td>
<td>**</td>
<td>/</td>
<td>*</td>
<td>***</td>
<td>**</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(dist. Pearl)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Error detection</strong></td>
<td>**</td>
<td>***</td>
<td>*</td>
<td>*</td>
<td>--- (C)</td>
</tr>
<tr>
<td></td>
<td>(strong typing)</td>
<td>(verification)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Large systems</strong></td>
<td>***</td>
<td>*</td>
<td>---</td>
<td>***</td>
<td>/</td>
</tr>
</tbody>
</table>

© 2009 Uwe R. Zimmer, The Australian National University
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
Physical coupling

Uwe R. Zimmer – The Australian National University
References for this chapter

[Burns01]
Alan Burns and Andy Wellings
*Real-Time Systems and Programming Languages*
Addison Wesley, third edition, 2001

[Peacock97]
G.R. Peacock
*Standards for temperature sensors*
web guide: [www.temperatures.com/standds.html](http://www.temperatures.com/standds.html)

[Edler01]
F. Edler, M. Kühne, E. Tegeler
*Noise temperature measurements for the determination of the thermodynamic temperature of the melting point of Palladium*

all references and some links are available on the course page
Real-Time Systems Components: Physical coupling

- Microcontroller (µC)
- Interface
- A/D
- Sensor
- Actuator
- Discrete
- Analogue
- Physical Phenomena
Physical coupling

First step to embed a system into the real world:

*Transform all kinds of physical phenomena into analogue voltages*

- e.g. speed, pressure, brightness, loudness, colour, force, humidity, distance, salinity, density, radioactivity, spectrograms, reflectivity, acceleration, deformation, ..., ..., ..., or: temperature
Physical coupling

Measuring temperature

Some observable effects of temperature changes:

- Mean square noise voltage changes
- Volume changes (gas, liquids, metals)
- Thermovoltage
- Changes in conductors and semiconductors
- State changes: into solid, liquid, or gaseous
Physical coupling: Measuring temperature (thermoelements)

Thermoelements

\[ E_{th} = K \cdot \text{grad}(T) \]

with \( K \): Seebeck coefficient (depends on material)

\[ U_{th} = \int_{0}^{L} E_{th} \, dl = \int_{0}^{L} K \cdot \text{grad}(T) \, dl = K(T_o - T_L) \]
Physical coupling: Measuring temperature (thermoelements)

Temperature measurement

- Take two wires (A & B) with different Seebeck coefficients $K_A$ & $K_B$
- Connect them on one side
- Place the connected side in one temperature zone, and the open ends in another
- Measure the voltage difference $U_{th}$
Physical coupling: Measuring temperature (thermoelements)

Temperature measurement

Measure the voltage difference $U_{th}$

$$U_{th} = K_A(T_1 - T_2) + K_B(T_2 - T_1) = (K_A - K_B)(T_1 - T_2) \text{ with } T_1 \text{ known}$$
### Physical coupling: Measuring temperature (thermoelements)

#### Temperature measurement

![Diagram of thermoelectric coupling]

some standard combinations: (typical shape: ![Thermocouple](image.png))

<table>
<thead>
<tr>
<th>short name</th>
<th>Material</th>
<th>$T_{max}$</th>
<th>$U_{th}$ with 0° to $T_{max}$</th>
<th>$K_A - K_B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T)</td>
<td>Cu-Constantan</td>
<td>400°C</td>
<td>21.000 mV</td>
<td>42.5×10^{-6}</td>
</tr>
<tr>
<td>(J)</td>
<td>Fe-Constantan</td>
<td>700°C</td>
<td>39.720 mV</td>
<td>53.7×10^{-6}</td>
</tr>
<tr>
<td>(K)</td>
<td>NiCr-Ni</td>
<td>1000°C</td>
<td>41.310 mV</td>
<td>41.1×10^{-6}</td>
</tr>
<tr>
<td>(S)</td>
<td>PtRh-Pt</td>
<td>1300°C</td>
<td>13.138 mV</td>
<td>6.43×10^{-6}</td>
</tr>
</tbody>
</table>
Physical coupling: Measuring temperature (thermoelements)

Thermocouples

Pro:
- accepts high temperatures
- small
- relatively cheap

Contra:
- requires stable amplifier
- temperature differences only
- cables to the sensor need to have the same Seebeck coefficient
Physical coupling: Measuring temperature (thermoelements)

Applications of standard thermocouples

- (TYPE N) Nicrosil-Nisil thermocouples are suitable for use in oxidizing inert or dry reducing atmospheres. Must be protected from sulphurous atmospheres. Very accurate at high temperatures. Virtually the same emf (electromotive force) and range as Type K.

- (TYPE J) Iron-Constantan thermocouples are suitable for use in vacuum, oxidizing, reducing or inert atmospheres. Suitable for measuring temperatures up to 760°C for largest wire size.

- (TYPE K) Chromel-Alumel thermocouples are suitable for continuous use in oxidizing or inert atmospheres up to 1260°C for largest wire size. Because their oxidation resistance characteristics are better than those of other base metal thermocouples, they find widest use at temperatures above 538°C.

- (TYPE E) Chromel-Constantan thermocouples are suitable for use up to 781°C in oxidizing or inert atmospheres for largest gauge wires. Type E thermocouples develop the highest emf per degree of all commonly used thermocouples.
**Physical coupling: Measuring temperature (thermoelements)**

**Applications of standard thermocouples (cont.)**

- (TYPE T) Copper-Constantan thermocouples are suitable for subzero temperatures with an upper temperature limit of 371°C and can be used in vacuum, oxidizing, reducing or inert atmospheres.

- (TYPE R) Platinum 13% Rhodium - Platinum thermocouples are suitable for continuous use in oxidizing or inert atmospheres at temperatures up to 1482°C.

- (TYPE S) Platinum 10% Rhodium - Platinum thermocouples are suitable for continuous use in oxidizing or inert atmospheres at temperatures up to 1482°C.

- (TYPE B) Platinum 30% Rhodium - Platinum 6% Rhodium thermocouples are suitable for continuous use in oxidizing or inert atmospheres and short-term use in vacuum atmospheres at temperatures up to 1705°C.

- (TYPE W) Tungsten - Rhenium Alloy thermocouples are used to measure temperatures up to 2760°C. These thermocouples have inherently poor oxidation resistance and should be used in vacuum, hydrogen or inert atmospheres.
Physical coupling: Measuring temperature (thermoelements)

Linearity of standard thermocouples
Physical coupling: Measuring temperature (thermoelements)

Three standard forms

- Grounded (G) junction
- Ungrounded (U) junction
- Exposed (E) junction

source: Alltemp Sensors
Physical coupling: Measuring temperature (resistors)

**Thermoresistors:** \(Pt_{100}\) sensor

- e.g. in thinfilm technology:
- also common as a Platinum wire around a glass or ceramics tube

source: sensor-nite
Physical coupling: Measuring temperature (resistors)

Some characteristics of the $Pt_{100}$ sensor:

- general: $R_T = R_0[1 + A(T - T_0) + B(T - T_0)^2 + C(T - T_0)^3 + \ldots]$ 
- Platinum: $A \approx 3.27 \times 10^{-3} \frac{1}{K} \ (550^\circ C) \ .. \ 4.2 \times 10^{-3} \frac{1}{K} \ (-150^\circ C)$
- $Pt_{100}$: $0^\circ C : R_T = 100 \Omega \pm 0.1 \Omega \approx \pm 0.26 \ K$
- correction required to compensate for non-linearity
- range approx. -200$^\circ$C .. +650$^\circ$C
- speed: 0.1s in flowing water .. multiple seconds in still air
Physical coupling: Measuring temperature (resistors)

Application of the $\text{Pt}_{100}$ sensor:

The problem of heating the sensor itself

- $\text{Pt}_{100}$ in a TO18 enclosure:

\[
R_{th} = \frac{T_{\text{Pt}_{100}} - T_E}{P_V} = 480^\circ\text{C/W in non-moving air}
\]

- limiting the sensor error to 0.5°C around 0°C ($R_T = 100\Omega$)

\[
R_{th} = \frac{T_{\text{Pt}_{100}} - T_E}{P_V} \Rightarrow P_V = \frac{\Delta T}{R_{th}} = \frac{U^2}{R} \Rightarrow U_{max} = \sqrt{\frac{\Delta T \cdot R_T}{R_{th}}} \approx 0.323\text{ V}
\]
Physical coupling: Measuring temperature (resistors)

More things to consider for the $Pt_{100}$ sensor:

- compensate for the non-linearity of sensor itself
- choose an adequate (bridge) circuit, e.g. a ‘four wire’ setup:
- limit the cable length
- keep the cables on the same temperature
- limit the influence of the sensor to the environment
Physical coupling: Measuring temperature (resistors)

In relation to thermocouples:

Pro:

• higher accuracy (if all side-effects have been taken care of)
• non-linearity less worse than with thermocouples
• long term stability
• measures absolute temperatures

Contra:

• smaller temperature range -200°C .. +650°C (some thermocouples exceed +2000°C)
• more expensive
• less robust (can be compensated by probe construction)
• usually bigger
Physical coupling: Measuring temperature (thermistors)

Temperature sensitive semi-conductors (Thermistors):

**NTC**

![Graph with B](image)

\[ R_{25°C} \]

\[ (ΔR)_{25°C} \]

\[ = \frac{(ΔR)_{25°C}}{T} \]

**PTC**

![Graph with B ± ΔB](image)

\[ R_0 \]

\[ 1000000 \]

\[ 100000 \]

\[ 1000 \]

\[ 100 \]

\[ 10 \]

\[ 0 \]

\[ 25 \]

\[ 50 \]

\[ 100 \]

\[ 150 \]

\[ 200 \]

source: AVX
Physical coupling: Measuring temperature (thermistors)

In relation to thermocouples and $Pt_{100}$:

Pro:

• very high accuracy (with special models, if extensive compensation is applied)
• usually cheaper than $Pt_{100}$
• high long term stability (with special models)
• large changes with temperature

Contra:

• even smaller temperature range -40°C .. +350°C
• terrible non-linearities
• standard components are very inaccurate and instable
• larger than thermocouples and $Pt_{100}$
Physical coupling: Measuring temperature (noise)

Noise temperature measurement

based on Nyquist formula: $\overline{U^2} = 4kT R \Delta f$

combined thermocouple-noise sensor

(source: Physikalisch-Technische Bundesanstalt, Berlin)

with $k$: Boltzmann constant, $T$: thermodynamic temperature, $R$: electric resistance, and $\Delta f$: the measurement bandwidth
Physical coupling: Measuring temperature (noise)

Noise temperature measurement

Pro:
• wide range
• high accuracy
• long term stability

Contra:
• expensive
• sophisticated amplifier setups

an actual device: range: 1..2500°K, accuracy: ±0.1% (over the full range)
Physical coupling: Measuring temperature

More ways to measure temperature:

• Spreading resistors
• Piezos and other temperature sensitive quarz elements
• Temperature controlled current sources (e.g. AD590)
• Mercury filled thermometers
• ...
Physical coupling

Basic conclusions

☞ we just scratched the surface of conversion methods for ONE physical value (temperature).

☞ converting physical phenomena into analogue voltages seems to be a complex matter … in fact a whole industry is dedicated to this field exclusively.

☞ always ask for the full sensor specifications (and read them).

☞ never assume that the output is a linear translation of a physical value.

Physical coupling is not the only loss afflicted stage of conversion, but it is usually the most complex one.
Physical coupling

Range and relative speed measurements

- **Triangulation** (optical)
- **Time of flight** (optical, acoustical, electro-magnetic)
  - Phase correlation (optical, acoustical, electro-magnetic)
- **Intensity** (optical, acoustical)
- **Doppler methods** (acoustical, electro-magnetic)
- **Interferometry** (optical, electro-magnetic)
Physical coupling

Range measurements – Triangulation

- non-linear, very poor resolution, if $d \gg b$
- highly focused light-beam required $\Rightarrow$ laser $\Rightarrow$ safety considerations
- distances are measured along the optical axis only.

\[ d = b \cdot \tan \alpha \]
Physical coupling

Range measurements – Triangulation

\[ d = b \cdot \tan \alpha \]

• projected point might be hidden \( \Rightarrow \) no measurement

\( \Rightarrow \) method is frequently used for measuring liquid levels.

© 2009 Uwe R. Zimmer, The Australian National University
Physical coupling

Range measurements – Time of flight - Phase correlation

Method: measure the time of flight between the outgoing signal and the received reflected signal.

☞ in case of light, this method requires high resolution timers (> 1GHz)

• Method is perfectly linear.

• The achieved resolution depends on the precision of the signal’s rising edge and the resolution of employed timers.

• Signals can be formed and volume measurements are possible.

☞ in order to increase the resolution, the outgoing signals are often modulated and the phase shifts between outgoing and reflected signals are detected.
Physical coupling

Range measurements: Ultrasound & Infrared

In the special case of one ultrasound transducer for sending and receiving (like above), there is a short time delay before the transducer is ready to receive a signal (oscillations need to die away first).
Physical coupling

Range measurements: Laser

a common laser range finder (SICK):
- range: max. 80\,m
- angular resolution: 0.25°
- response time: max. 53\,ms
- resolution: 10\,mm
- accuracy: typ. 5-10\,mm
Physical coupling

Minimal reflectance versus maximal range

![Graph showing minimal reflectance versus maximal range for different devices and conditions.](image)
Physical coupling

*Speed measurements: Doppler current profilers*

Physical effect: Doppler shift frequency $f_d = -2f_s \frac{V}{c}$

with source frequency $f_s$, relative velocity $V$ and signal speed $c$.

A common current profiler (SonTek):

- ping signal: 250 kHz - 3 MHz
- range: up to 160 m; velocity: ±10 m/s
- resolution: 0.15 - 2 m; 1 mm/s
- accuracy: ±1%
- blanking zone: 0.2 - 2.0 m
Summary

Physical coupling

• Physical phenomena

• Measuring temperature
  • thermoelements, thermocouples
  • thermoresistors
  • thermistors
  • noise temperature measurement) and others

• Measuring range and relative speed
  • triangulation
  • time of flight
  • intensity
  • Doppler methods
  • interferometry

• Examples: time-of flight ultrasound & laser, Doppler current profiler
Converters & Interfaces

Uwe R. Zimmer – The Australian National University
References for this chapter

[Burns01]
Alan Burns and Andy Wellings
Real-Time Systems and Programming Languages
Addison Wesley, third edition, 2001

[Kester93]
Walt Kester, James Bryant, Joe Buxton
High Resolution Signal Conditioning ADCs
System Applications Guide, Analog Devices, 1993, Section 3

[M_MPC565/566_00]
Motorola
MPC565/MPC566 microcontroller
Reference manual, 2000

[M_TPU_96]
Motorola
TPU – Time Processor Unit
Reference manual, 1996

[NS_ADC08200_02]
National Semiconductor
ADC08200 – 8-Bit, 20 MSPS to 200 MSPS, 1.05 mW/MSPS A/D Converter
Datasheet, 2002

[NS_LM12L458_99]
National Semiconductor
LM12L458 – 12-Bit + Sign Data Acquisition System with Self-Calibration
Datasheet, 1999

all references and links are available on the course page
Real-Time Systems Components: A/D, D/A & Interfaces

µC
Interface
A/D
Sensor
µC
Interface
A/D
µC
Interface
D/A
Actuator
µC
Interface
A/D
Sensor
µC
Interface
D/A
Actuator

Discrete
Analogue
Physical Phenomena
Signal chain
Sampling data

Sample data with a frequency $f_s$
Sampling data

Sample data with a frequency $f_s$

Interpolation suggests an original signal like
Sample data with a higher frequency $f_s$

suggests an original signal like

the phenomenon of the observed signal at a lower $f_s$ is called "aliasing"
Nyquist’s criterion

An analog signal with a bandwidth of $f_a$ must be sampled at

$$f_s > 2f_a$$

in order to avoid loss of information.
Nyquist’s criteria

An analog signal with a bandwidth of \( f_a \) must be sampled at

\[
fs > 2fa
\]

\( fs \) theoretical sufficient: | practical case: | (oversampling)
A resolution of $N$ bit gives $2^N$ possible discrete output levels:

- smallest distinguishable value (least significant bit or LSB):
  
  $$q = \frac{1}{2^N}$$

- ratio $1/q$ expressed in $\text{db}$:
  
  $$10\log(2^{2N}) = N \cdot 20\log 2 \approx N \cdot 6.02\text{db}$$
Quantization

The mean square error over one step and the rms noise voltage is:

$$E^2 = \frac{1}{q} \int_{-q/2}^{q/2} E^2 dE = \frac{q^2}{12}$$

rms noise voltage:

$$\sqrt{\frac{q}{12}}$$
Quantization

The signal $S$ with respect to the rms error is thus:

$$S = S \cdot 2^N \sqrt{12}$$

or as the “Signal to Noise Ratio”:

$$SNR[db] = 10 \log \left( \frac{S^2}{q^2/12} \right)$$
Quantization

Assuming an ideal input signal:

\[ F(t) = A \sin \omega t ; \quad q = \frac{2A}{2^N} \]

then

\[ SNR[db] = 10 \log \left( \frac{A^2}{2} / \frac{q^2}{12} \right); \quad \bar{A} = \frac{A}{\sqrt{2}} \]
Assuming an ideal input signal:

\[ F(t) = A \sin \omega t \; ; \; q = \frac{2A}{2^N} \]

then

\[ SNR[db] = 10 \log \left( \frac{A^2}{2} / \frac{q^2}{12} \right) \]

\[ SNR[db] = 10 \log \left( \frac{A^2}{2} / \frac{A^2}{3 \times 2^{2N}} \right) \]

\[ SNR[db] = 10 \log \left( \frac{3 \times 2^{2N}}{2} \right) \]
Quantization

Assuming an ideal input signal:

\[ F(t) = A \sin \omega t \; ; \; q = \frac{2A}{2^N} \]

then (cont.)

\[ SNR[db] = 10 \log \left( \frac{3 \cdot 2^N}{2} \right) \]

\[ SNR[db] = 20N \log 2 + 10 \log \frac{3}{2} \]

\[ \approx N \cdot 6.02 + 1.76 \]
Quantization

Determining the effective number of bits (ENOB):

\[
SNR_{\text{ideal}}[\text{db}] = 20N\log 2 + 10\log \frac{3}{2}
\]

\[
ENOB = \frac{SNR_{\text{actual}} - 10\log (3/2)}{20\log 2}
\]

\[
ENOB \approx \frac{SNR_{\text{actual}} - 1.76}{6.02}
\]

(ENOB = N for an ideal A/D converter)
Quantization

Actual A/D converters also have:

- **Missing codes**
  (reduce SNR by $20 \log_2 2$ or $6.02 \text{db}$ for each missing code)

- **Differential non-linearity (DNL):**
  differences between successive code-widths

- **Integral non-linearity (INL):**
  maximal difference between code-centres and the ideal line.
A/D, D/A & Interfaces

A/D converter types

Some central criteria for A/D converters:

- Throughput (max. sampling frequency)
- Resolution (ENOB)
- Latency
- Accuracy

☞ Trade-off between: Throughput and Resolution | Accuracy
Flash converter

- $2^N - 1$ comparators are directly connected to the input signal $A_{IN}$

- Accuracy depends on the accuracy of the reference voltages only

- Conversion in one cycle

but

- a straight 16-bit ADC flash converter would require:
  65535 comparators and references
Pipelined flash converter

- divides the conversion in multiple stages with an $m$ bit ADC and DAC each.
- digital conversion results are accumulated ($D$)
- Analog residues are forwarded ($A$)

$\Rightarrow$ with $\rho m$-bit pipeline stages the converter achieves a resolution of:

$$n = \rho m \text{ bits}$$
Successive approximation shift register (SAR)

- Compare A successively with (MSB to LSB):
  \[ \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots, \frac{1}{2^N} \] of full scale

- Reduce the input in each step according to already converted bits (and shift the register).

- Conversion takes \( N \) steps
  
  ✤ Only one comparator required
\[ \Sigma - \Delta \ A/D \text{ Converters} \]

- The output of a 1-bit DAC (\( \pm U_{\text{ref}} \)) is subtracted from the input and integrated.
- The output of the integrator is compared against signal ground.
- According to the comparator the output of the 1-bit DAC changes.

☞ This is done at a very high rate!  
The density of ‘1’s in the bit-stream represents the signal level of A.
2nd order $\Sigma$-$\Delta$ A/D Converters

- Converter is inherently linear.
- Due to the high ‘oversampling’ rates, relatively slow.
- High resolutions (typically 16-24 bits).

- Higher order S-D converters achieve an even better signal to noise ratio.
Real-Time & Embedded Systems

A/D, D/A & Interfaces

Σ-Δ A/D Converters

Relation between resolution (ENOB, SNR) and oversampling in higher order Σ-Δ Converters (qualities):
## Comparison between these four A/D converters

<table>
<thead>
<tr>
<th></th>
<th>Flash</th>
<th>Pipelined Flash</th>
<th>SAR</th>
<th>Σ-Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Throughput</strong></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td><strong>Resolution</strong></td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><strong>Latency</strong></td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td><strong>Accuracy</strong></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

1 = best
A/D, D/A & Interfaces

ADC08200
(National Semiconductor)

Pipelined Flash converter with 2 stages and 2 pipelines
ADC08200 – basic characteristics

- Resolution: 8 bit
- Sampling frequency: 10 - 200 (230) MHz
- Differential non-linearity (DNL): ±0.4 LSB (typical), ±0.95 LSB (max.)
- ENOB: 7.5 (at 4 MHz), 7.3 (at 50 MHz), 7.0 (at 100 MHz)
- No missing codes
- Power consumption: 1.05 mW/MSPS, 1 mW (power down)
- Latency: 6 cycles (pipeline delay)
- Aperture (sampling delay): 2.6 ns, with 2 ps rms jitter
ADC08200 – timing
ADC08200 – integral non-linearity
A/D, D/A & Interfaces

LM12L458

(National Semiconductor)
A/D, D/A & Interfaces

LM12L458
12-Bit + sign, 8 channel, A/D converter, controller and interface

Controller features:

- Programmable acquisition times and conversion rates
- 32-word conversion FIFO
- Self-calibration and diagnostic mode
- 8- or 16-bit wide data bus microprocessor or DSP

Typ. applications:

- Data Logging
- Process Control
## LM12L458 – accessible registers

<table>
<thead>
<tr>
<th>A4</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>Purpose</th>
<th>Type</th>
<th>D15</th>
<th>D14</th>
<th>D13</th>
<th>D12</th>
<th>D11</th>
<th>D10</th>
<th>D9</th>
<th>D8</th>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Instruction RAM (RAM Pointer = 00)</td>
<td>R/W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Instruction RAM (RAM Pointer = 00)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Instruction RAM (RAM Pointer = 01)</td>
<td>R/W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Instruction RAM (RAM Pointer = 01)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Instruction RAM (RAM Pointer = 10)</td>
<td>R/W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Instruction RAM (RAM Pointer = 10)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Configuration Register</td>
<td>R/W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Configuration Register</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Interrupt Enable Register</td>
<td>R/W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Interrupt Enable Register</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Interrupt Status Register</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Interrupt Status Register</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Timer Register</td>
<td>R/W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Timer Register</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Conversion FIFO</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Conversion FIFO</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Limit Status Register</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Limit Status Register</td>
</tr>
</tbody>
</table>

- **Purpose**
  - Instruction RAM
  - Configuration Register
  - Interrupt Enable Register
  - Interrupt Status Register
  - Timer Register
  - Conversion FIFO
  - Limit Status Register

- **Type**
  - R/W

- **Access**
  - Don’t Care
  - >/≤ Sign

- **Function**
  - Acquisition Time
  - Watch-dog
  - Timer
  - Test = 0
  - Number of Conversions in Conversion FIFO
  - Sequencer Address to Generate INT1
  - Actual Number of Conversion Results in Conversion FIFO
  - Address of Sequencer Instruction being Executed
  - Timer Preset High Byte
  - Timer Preset Low Byte
  - Sign

- **Values**
  - Limit #1
  - Limit #2

- **Other**
  - VIN
  - Timer Sync
  - VIN+Pause Loop
  - Test RAM I/O
  - Auto Chan Stand- Full Auto- Reset Start
  - Interrupt Enable Number of Conversions Sequencer INT7 Don’t INT5 INT4 INT3 INT2 INT1 INT0
  - Auto- Zero
  - Full CAL
  - Stand-by
  - Chan Mask
  - Auto Zero
  - Sel
  - INT5 INT4 INT3 INT2 INT1 INT0
every entry in the **instruction RAM** consists of:

- **Loop** (1bit): indicates the last instruction and branches to the first one.
- **Pause** (1bit): halts the sequencer before this instruction.
- $V_{IN+}, V_{IN-}$ (2*3bit): select the input channels (000 selects ground in $V_{IN-}$)
- **Sync** (1bit): wait for an external sync. signal before this instruction.
- **Timer** (1bit): wait for a preset 16-bit counter delay before this instruction.

| A4 A3 A2 A1 | Purpose | Type | D15 D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0 |
|------------|---------|------|------------------------|-----------------|------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 0 0    | Instruction RAM (RAM Pointer = 00) | R/W  | Acquisition Time     | Watch-dog 8/T2 | Timer Sync | $V_{IN-}$ | $V_{IN+}$ | Pause | Loop |
| 0 0 0 0    | Instruction RAM (RAM Pointer = 01) | R/W  | Don't Care            | $>/\leq$ Sign | Limit #1     |                 |                 |                 |
| 0 0 0 0    | Instruction RAM (RAM Pointer = 10) | R/W  | Don't Care            | $>/\leq$ Sign | Limit #2     |                 |                 |                 |

© 2009 Uwe R. Zimmer, The Australian National University
every entry in the **instruction RAM** consists of (cont.):

- **8/12** (1 bit): selects the resolution (8 bit + sign or 12 bit + sign).

- **Watchdog** (1 bit): activates comparisons with two programmed limits.

- **Acquisition time** \((D)\) (4 bit): the converter takes \(9 + 2D\) cycles (12 bit mode) or \(2 + 2D\) cycles (8 bit mode) to sample to input. Depends on the input resistance: \(D = 0.45 \cdot R_S[k\Omega] \cdot f_{CLK}[MHz]\) for 12 bit conversions.

- **Limits** (including sign and comparator): used for Watchdog operation.
type ChannelPlus is (Ch0, Ch1, Ch2, Ch3, Ch4, Ch5, Ch6, Ch7);
type ChannelMinus is (Gnd, Ch1, Ch2, Ch3, Ch4, Ch5, Ch6, Ch7);
type Resolutions is (TwelveBit, EightBit);
type Acquisition_D is new Integer range 0..15; -- 9+2D (12bit), 2+2D (8bit)

for ChannelPlus use
(Ch0 => 0, Ch1 => 1, Ch2 => 2, Ch3 => 3,
 Ch4 => 4, Ch5 => 5, Ch6 => 6, Ch7 => 7);

for ChannelMinus use
(Gnd => 0, Ch1 => 1, Ch2 => 2, Ch3 => 3,
 Ch4 => 4, Ch5 => 5, Ch6 => 6, Ch7 => 7);

for Resolutions use
(TwelveBit => 0, EightBit => 1);

type Instruction is record
   EndOfLoop, Pause, Sync, Timer, Watchdog : Boolean;
   Vplus                                   : ChannelPlus;
   Vminus                                  : ChannelMinus;
   Resolution                              : Resolutions;
   AcquisitionTime                         : Acquisition_D;
end record;

<table>
<thead>
<tr>
<th>A4</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>Purpose</th>
<th>Type</th>
<th>D15</th>
<th>D14</th>
<th>D13</th>
<th>D12</th>
<th>D11</th>
<th>D10</th>
<th>D9</th>
<th>D8</th>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Instruction RAM</td>
<td>R/W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(RAM Pointer = 00)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Acquisition Time</td>
<td>Watchdog</td>
<td>8/12</td>
<td>Timer</td>
<td>Sync</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Watchdog</td>
<td>V_{IN^-}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Pause</td>
<td>V_{IN^+}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Loop</td>
<td>Pause</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LM12L458 – instruction RAM
Units_Per_Word : constant Integer := Word_Size / Storage_Unit;

for Instruction use record

   EndOfLoop at 0*Units_Per_Word range 0..0;
   Pause at 0*Units_Per_Word range 1..1;
   Vplus at 0*Units_Per_Word range 2..4;
   Vminus at 0*Units_Per_Word range 5..7;
   Sync at 0*Units_Per_Word range 8..8;
   Timer at 0*Units_Per_Word range 9..9;
   Resolution at 0*Units_Per_Word range 10..10;
   Watchdog at 0*Units_Per_Word range 11..11;
   AcquisitionTime at 0*Units_Per_Word range 12..15;

end record;
Real-Time & Embedded Systems

LM12L458 – instruction RAM

<table>
<thead>
<tr>
<th>A4</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>Purpose</th>
<th>Type</th>
<th>D15</th>
<th>D14</th>
<th>D13</th>
<th>D12</th>
<th>D11</th>
<th>D10</th>
<th>D9</th>
<th>D8</th>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Instruction RAM</td>
<td>R/W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>to</td>
<td></td>
<td></td>
<td>(RAM Pointer = 00)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for Instruction’Size use 16; -- Bits
for Instruction’Alignment use 2; -- Storage_Units (Bytes)
for Instruction’Bit_Order use High_Order_First;

type Instructions is array (0..7) of Instruction;
pragma Pack (Instructions);

ADC_Instructions : Instructions;
for ADC_Instructions’Address use To_Address (16#0000132D#);
ADC_Instructions (0) := (EndOfLoop => False,
Pause => False,
Vplus => Ch0,
Vminus => Gnd,
Sync => True,
Timer => False,
Resolution => EightBit,
Watchdog => False,
AquisitionTime => 10);

ADC_Instructions (1) := (EndOfLoop => True, -- last instruction
Pause => False,
Vplus => Ch1,
Vminus => Ch2,
Sync => False,
Timer => False,
Resolution => TwelveBit,
Watchdog => False,
AquisitionTime => 0);
**Data structures in ‘C’:**

```c
enum ChannelPlus  {Ch0=0, Ch1, Ch2, Ch3, Ch4, Ch5, Ch6, Ch7};
enum ChannelMinus {Gnd=0, Ch1, Ch2, Ch3, Ch4, Ch5, Ch6, Ch7};
enum Resolutions  {TwelveBit=0, EightBit};

struct {
    unsigned int EndOfLoop      : 1;
    unsigned int Pause          : 1;
    ChannelPlus  Vplus          : 3;
    ChannelMinus Vminus         : 3;
    unsigned int Sync           : 1;
    unsigned int Timer          : 1;
    Resolutions  Resolution     : 1;
    unsigned int Watchdog       : 1;
    unsigned int AcquisitionTime : 4;
} Instruction;
```
Real-Time & Embedded Systems

LM12L458 – instruction RAM

| A4 | A3 | A2 | A1 | Purpose                                      | Type   | D15 | D14 | D13 | D12 | D11 | D10 | D9  | D8  | D7  | D6  | D5  | D4  | D3  | D2  | D1  | D0  |
|----|----|----|----|---------------------------------------------|--------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0  | 0  | 0  | 0  | Instruction RAM                             | R/W    |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 0  | to |     |    | (RAM Pointer = 00)                          |        |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 1  | 1  | 1  | 1  |                                              |        |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

Data structures in ‘C’:

```c
struct {
    unsigned int EndOfLoop      : 1;
    unsigned int Pause          : 1;
    ChannelPlus Vplus          : 3;
    ChannelMinus Vminus         : 3;
    unsigned int Sync           : 1;
    unsigned int Timer          : 1;
    Resolutions Resolution     : 1;
    unsigned int Watchdog       : 1;
    unsigned int AcquisitionTime : 4;
} Instruction;

Instruction    InstructionsA[8];
InstructionsA *Instructions;
Instructions = 0x0000132D;
```
Data structures in ‘C’:

```c
*Instructions (0).EndOfLoop = 0;
*Instructions (0).Pause = 0;
*Instructions (0).Vplus = Ch0;
*Instructions (0).Vminus = Gnd;
*Instructions (0).Sync = 1;
*Instructions (0).Timer = 0;
*Instructions (0).Resolution = EightBit;
*Instructions (0).Watchdog = 0;
*Instructions (0).AquisitionTime = 10;
```

If this works, you were lucky two times:

- The compiler implemented the struct-fields in the intended places and order.
- The bit ordering in your device is the way the compiler assumed it.
Macros-Assembler style programming:

In order to produce portable code in ‘C’, it is necessary to set bits manually:

```c
unsigned int setbits (unsigned int *r,
                     unsigned int n,          /* set n bits     */
                     unsigned int p,          /* at position p */
                     unsigned int x)          /* to bitstring x */
{
    unsigned int mask;
    mask = ~(~0 << n);
    *r &= ~(mask << p);
    *r |= (x & mask) << p;
    return (*r);
}
```
The **configuration register** consists of (1 bit each):

- **Start**: starts the sequencer.
- **Reset**: sets the instruction pointer to ‘000’.
- **Auto-Zero**: triggers a ‘short’ calibration (76 cycles – 1 offset sample).
- **Full-Cal**: initiates a full calibration (4944 cycles, 8 samples) interrupt.
- **Stand-by**: disconnects the external clock, preserves the registers. After powering up again (≈ 10 ms): a specific interrupt is issued.
- **Chan-Mask**: format selection for the FIFO output registers.
- **Auto-Zero\(_{ec}\)**: auto-zeros the ADC automatically in every conversion.
The **configuration register** consists of (1 bit each) (cont.):

- **I/O Sel**: sets the Sync pin to input or output mode.
- **RAM Pointer**: selects the current (16-bit) part in each 48-bit instruction.
- **Test=0**: production testing mode: leave this bit at ‘0’.
- **DIAG**: connects $V_{IN^+}$ and $V_{IN^-}$ to $V_{REF^+}$ and $V_{REF^-}$ for testing purposes.
LM12L458 – sequencer operation

IP := 0;
loop
  repeat
    if Auto_Zero or Full_Cal then Calibrate;
  until Start_bit;
  if Instr (IP).Timer then Run_Timer;
  Current_Signal := Aquisition (Instr (IP).Aquisition_Time);
  if Instr (IP).Watchdog then begin
    if Instr (IP).Sync then Wait_for_external_sync;
    Compare (Current, Instr (IP).Limit_1);
    if Instr (IP).Sync then Wait_for_external_sync;
    Compare (Current, Instr (IP).Limit_2);
  end;
  else
    if Instr (IP).Sync then Wait_for_external_sync;
    Convert_and_store_in_FIFO (Current_Signal);
  end;
  if Instr (IP).Loop then IP := 0;
  else IP := IP + 1;
end loop;
The conversion FIFO is accessible via one address only:

- Every read on this address will delete this result from the internal memory and shift the next one into the visible conversion FIFO register.
- The FIFO holds 32 conversion results max.
- **Data will be lost**, if the results are not read fast enough to prevent a buffer overrun.

☞ The controller can issue specific interrupts or initiate a DMA transfer, when a given number of results are accumulated or a certain instruction is completed.
on the CPU side: Polling, interrupt or DMA scheme
µControllers

MC68HC05

- **Clock**: max. 2.1MHz internal (4.2MHz external)
- **RAM**: 176bytes
- **ROM**: 5936bytes
- **EEPROM**: 256bytes
- **Power saving modes** (stop, wait, slow)
- **Serial**: 46-76800 baud (at 2.4576MHz) / 79-131072 baud (at 4.194394MHz)
- **Parallel I/O**: 3*8bit; Parallel in: 1*8bit
- **Timers**: 1*16bit
- **A/D**: 8 channels, 8bit
- **PWM**: 2 generators
MAIN BRCLR 6,TSR,MAIN ;Loop here till Output Compare flag set
LDA OCMP+1 ;Low byte of Output Compare register
ADD #$D4 ;Add \( \Delta t_1 = (50\text{ms} / 4\mu\text{s}) \mod 2^8 = $D4 \)
STA TEMPA ;Save till high half calculated
LDA OCMP ;High byte of Output Compare register
ADC #$30 ;Add \( \Delta t_h = (50\text{ms} / 4\mu\text{s}) \div 2^8 = $30 \) (+carry)
STA OCMP ;Update high byte of Output Compare register
LDA TEMPA ;Get low half of updated value
STA OCMP+1 ;Update low half and reset Output Compare flag
LDA TIC ;Get current TIC value
INCA ;TIC := TIC + 1
STA TIC ;Update TIC
CMP #20 ;20th TIC?, 1 second passed?
BLO NOSEC ;If not, skip next clear
CLR TIC ;Clear TIC on 20th

NOSEC EQU *

JSR TIME ;Update time-of-day & day-of-week
JSR KYPAD ;Check/service keypad
JSR A2D ;Check Temp Sensors
JSR HVAC ;Update Heat/Air Cond Outputs
JSR LCD ;Update LCD display
BRA MAIN ;End of main loop
μControllers

MPC565
µControllers MPC565

- -40° - +125°C, power dissipation: 0.8 - 1.12W
- CPU: PowerPC core (incl. FPU & BBC), 40/56MHz
- RAM: flash: 1M, static: 36K
- Time processing units: 3 (via dual-ported RAM)
- Timers: 22 channels (PWM & RTC supported)
- A/D convertors: 40 channels, 10bit, 250kHz
- Can-bus: 3 TOUCAN modules
- Serial: 2 interfaces
- Data link controller: SAE J1850 class B communications module
- Real-time embedded application development interface: NEXUS debug port (IEEE-ISTO 5001-1999)
- Packing: 352/388 ball PBGA
µControllers MPC565

Time processing unit

a special-purpose µcontroller:

- Independent µengine.
- 16 digital I/O channels with independent *match* and *capture* capabilities.
- Meant to operate these I/O channels for timing control purposes.
- Predefined µengine command set (ROM functions in control store).
- 2 16-bit time bases
### µControllers MPC565

#### Time processing unit: some example functions

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Period- / Pulse-width accumulator</td>
<td>The period/pulse-width accumulator (PPWA) algorithm accumulates a 16-bit or 24-bit sum of either the period or the pulse width of an input signal over a programmable number of periods or pulses (from 1 to 255).</td>
</tr>
<tr>
<td>Stepper motor</td>
<td>The stepper motor (SM) control algorithm provides for linear acceleration and deceleration control of a stepper motor with a programmable number of step rates of up to 14.</td>
</tr>
<tr>
<td>Position-synchronized pulse generator</td>
<td>The PSP function generates pulses of variable length at specified “angles.” Angle clock period is measured (in TCR1 clocks) using the PMA/PMM function on another channel.</td>
</tr>
<tr>
<td>Period measurement</td>
<td>This function measures the period (in TCR1 clocks) between regularly occurring input transitions and makes this period available for use by other functions or by the CPU (optional detection of misses and additional transitions).</td>
</tr>
<tr>
<td>Pulse-width modulation</td>
<td>The TPU can generate a pulse-width modulation (PWM) waveform with any duty cycle from zero to 100% (within the resolution and latency capability of the TPU).</td>
</tr>
<tr>
<td>Synchronized pulse-width modulation</td>
<td>Three different operating modes allow the function to maintain complex timing relationships between channels without CPU intervention.</td>
</tr>
<tr>
<td>Quadruple decode</td>
<td>QDEC uses two channels to decode a pair of out-of-phase signals in order to present the CPU with directional information and a position value.</td>
</tr>
</tbody>
</table>
µControllers MPC565

Emulation mode
(or: create your own µengine)

Refer the control store of the µengine to the dual-ported RAM instead of the integrated ROM area and supply:

- up to 16 µengine commands (functions)
- in 2-8Kbyte of long-word (32-bit) organized memory
- programmed in a 32-bit µinstruction format (explained next)

the dual-ported RAM is now cut off from the CPU
(the TPU parameter RAM is not affected)
µControllers MPC565 – TPU: the µinstruction formats:

1: Execution unit and RAM

2: Execution unit, flag, and channel control

3: Conditional branch, flag, and channel control

4: Jump, flag, and RAM

5: Execution unit, immediate, and flag
### µControllers MPC565 – TPU: the µinstruction formats:

<table>
<thead>
<tr>
<th>RW</th>
<th>RAM</th>
<th>Read/Write Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1ABS</td>
<td>T1</td>
<td>A-Bus Source Control</td>
</tr>
<tr>
<td>T3ABD</td>
<td>T3</td>
<td>A-Bus Destination Control</td>
</tr>
<tr>
<td>SHF</td>
<td>AU</td>
<td>Shifter Control</td>
</tr>
<tr>
<td>SRC</td>
<td>Shift</td>
<td>Register Control</td>
</tr>
<tr>
<td>CCL</td>
<td>AU</td>
<td>Condition Code Latch Control</td>
</tr>
<tr>
<td>T1BBS</td>
<td>T1</td>
<td>B-Bus Source Control</td>
</tr>
<tr>
<td>CIN</td>
<td>AU</td>
<td>B-Bus Carry Control</td>
</tr>
<tr>
<td>BINV</td>
<td>AU</td>
<td>B-Bus Invert Control</td>
</tr>
<tr>
<td>IOM</td>
<td>RAM</td>
<td>Input/Output Mode Control</td>
</tr>
<tr>
<td>AIS</td>
<td>RAM</td>
<td>Address</td>
</tr>
<tr>
<td>DEC/END</td>
<td>SEQ</td>
<td>Decrementor / End Control</td>
</tr>
</tbody>
</table>

**Operation groups:**
- Execution unit
- Channel control
- RAM
- Sequencer
**μControllers MPC565 – TPU: the μinstruction formats:**

2: Execution unit, flag, and channel control

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>ERW</td>
<td>T1ABS</td>
<td>T3ABD</td>
<td>SHF</td>
<td>TDL</td>
<td>MRL</td>
<td>T1BBS</td>
<td>CIN</td>
<td>BINV</td>
<td>PAC</td>
<td>LSL</td>
<td>PSC</td>
</tr>
</tbody>
</table>

**Operation groups:**

- **Execution unit**
- **Channel control**
- **RAM**
- **Sequencer**

<table>
<thead>
<tr>
<th>ERW</th>
<th>RAM</th>
<th>Event Register Write Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1ABS</td>
<td>T1</td>
<td>A-Bus Source Control</td>
</tr>
<tr>
<td>T3ABD</td>
<td>T3</td>
<td>A-Bus Destination Control</td>
</tr>
<tr>
<td>SHF</td>
<td>AU</td>
<td>Shifter Control</td>
</tr>
<tr>
<td>TDL</td>
<td>CC</td>
<td>Transition Detect Latch Negation Control</td>
</tr>
<tr>
<td>MRL</td>
<td>CC</td>
<td>Match Recognition Latch Negation Control</td>
</tr>
<tr>
<td>T1BBS</td>
<td>T1</td>
<td>B-Bus Source Control</td>
</tr>
<tr>
<td>CIN</td>
<td>AU</td>
<td>B-Bus Carry Control</td>
</tr>
<tr>
<td>BINV</td>
<td>AU</td>
<td>B-Bus Invert Control</td>
</tr>
<tr>
<td>PAC</td>
<td>CC</td>
<td>Pin Action Control</td>
</tr>
<tr>
<td>LSL</td>
<td>CC</td>
<td>Link Service Latch Negation Control</td>
</tr>
<tr>
<td>PSC</td>
<td>CC</td>
<td>Pin State Control</td>
</tr>
<tr>
<td>FLC</td>
<td>CC</td>
<td>Flag Control</td>
</tr>
<tr>
<td>CIR</td>
<td>CC</td>
<td>Channel Interrupt Request</td>
</tr>
<tr>
<td>DEC/END</td>
<td>SEQ</td>
<td>Decrementor / End Control</td>
</tr>
</tbody>
</table>
μControllers MPC565 – TPU: the μinstruction formats:

3: Conditional branch, flag, and channel control

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>BCC</td>
<td>SEQ</td>
<td>BAF (8:0)</td>
<td>TBS</td>
<td>PAC</td>
<td>BCF</td>
<td>PSC</td>
<td>FLC</td>
<td>CIR</td>
</tr>
<tr>
<td>24</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **BCC**: Branch Condition Code Field
- **FLS**: μPC Flush Control
- **BAF**: Branch Address Field
- **TBS**: Time Base Select Control
- **PAC**: Pin Action Control
- **BCF**: Branch Condition Control
- **PSC**: Pin State Control
- **FLC**: Flag Control
- **CIR**: Channel Interrupt Request
- **MTSR**: Match/Transition Detect Service Request Inhibit Control

**Operation groups:**
- Execution unit
- Channel control
- RAM
- Sequencer
μControllers MPC565 – TPU: the μinstruction formats:

| RW | RAM | Read/Write Control |
| NMA | SEQ | Next μPC Address Mode Control |
| FLS | SEQ | μPC Flush Control |
| BAF | SEQ | Branch Address Field |
| FLC | CC | Flag Control |
| LSL | CC | Link Service Latch Negation Control |
| IOM | RAM | Input/Output Mode Control |
| AIS | RAM | Address |
| DEC/END | SEQ | Decrementor / End Control |

Operation groups:
- Execution unit
- Channel control
- RAM
- Sequencer
**μControllers MPC565 – TPU: the μinstruction formats:**

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>T1ABS</td>
<td>1</td>
<td>T3ABD</td>
<td>SHF</td>
<td>SRC</td>
<td>CCL</td>
<td>IMMEDIATE DATA (7:0)</td>
<td>T1BBI</td>
<td>LSL</td>
<td>EQ/GE</td>
<td>FLC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A-Bus Source Control</td>
<td></td>
<td>A-Bus Destination Control</td>
<td>Shifter Control</td>
<td>Register Control</td>
<td>Condition Code Latch Control</td>
<td>B-Bus Immediate Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1ABS</td>
<td>T3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3ABD</td>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHF</td>
<td>AU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRC</td>
<td>Shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCL</td>
<td>AU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1BBI</td>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSL</td>
<td>CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EQ/GE</td>
<td>CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLC</td>
<td>CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CIR</td>
<td>CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC/END</td>
<td>SEQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation groups:**
- Execution unit
- Channel control
- RAM
- Sequencer
### µControllers MPC565 – TPU: the µinSTRUCTION formats:

A state of a function (example):

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>R</td>
<td>W</td>
<td>T1ABS</td>
<td>T3ABD</td>
<td>SHF</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>E</td>
<td>R</td>
<td>W</td>
<td>T1ABS</td>
<td>T3ABD</td>
<td>SHF</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>T1ABS</td>
<td>T3ABD</td>
<td>SHF</td>
<td>S</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>NMA</td>
<td>F L S</td>
<td>BAF (8:0)</td>
<td>TIBBS</td>
<td>FLC</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>E</td>
<td>R</td>
<td>W</td>
<td>T1ABS</td>
<td>T3ABD</td>
<td>SHF</td>
</tr>
</tbody>
</table>

- Proceed to next µinstruction
- Decrement register
- Proceed at “0”
- Decrement register
- Call at “0”
- End of state!

Function are composed of one or multiple “states”

States are **atomic** instructions blocks!
Ready to go …

Entities to consider:

- **States**: non-interruptible µcode-blocks
- **Functions**: constructed of one or multiple states
- **Channels**: digital I/O lines with 16-bit match and capture
- **Priorities**: of channels
- **Timers**: 2 16-bit time-bases

☞ Associate functions, time-bases, channels and priorities!

…but let it run …
µControllers MPC565 – TPU: Scheduler

Time slots for states on three priorities

☞ no state will be suspended permanently
Free time slots are transferred:

- Free slots are made available in priority order
- Channels on the same level are scheduled by their channel-number
States have variable lengths

A set of influences need to be considered to calculate latencies
Latencies

... for latencies of capture and match at each channel mind:

- only the time-base resolution (all channels are evaluated independently and in parallel)

... for the functions associated with individual channels mind the:

- number of active channels (max. 16)
- number of channels on each priority level (add max. 2 µcycles for each “state-switch”)
- number of available time slots on each priority level per full scheduler-cycle (4, 2, 1 slot{s})
- number of µcycles to execute individual states of a function (2 µcycles per µinstruction)
- number of RAM accesses during the execution of a state (each access may stall for 2 CPU cycles)
- TPU clock cycle frequency
Determining latencies

assuming that all states may execute at all times!

calculate the number of µcycles per state
Determining latencies

assuming that all states may execute at all times!

calculate a schedule based on priorities
Determining latencies

☞ assuming that all states may execute at all times!

☞ add potential state switches (2 cycles)
Determining latencies

Assuming that all states may execute at all times!

Add potential stall times for each RAM-access (2 µcycles)
assuming that all states may execute at all times!

Determine maximal latencies!
Determining latencies

assuming that all states on the same level may execute at all times!

on all other levels: the longest state is always executed!
μControllers MPC565

Time-base & Real-time clock

- 64 bit time base
driven by an external clock: e.g. 20MHz ☞ resolution: 50ns; range: ≈30000 years

- Free running
(not influenced by any CPU action or resets)

- 2 reference registers
(used for compares and interrupt generation)

- Real-Time clock supplies full seconds (32bit ☞ range: ≈136 years)
(not affected by CPU, resets, and operates in all low-power modes)
µControllers MPC565

Interrupt controller

- Handles up to 48 different sources
  (32 from internal modules, 8 from timers and clocks, and 8 external vectorized sources) and supply each of them with a unique interrupt-vector

- 8 interrupt levels are distinguished by the interrupt controller
  (32 interrupt levels are supplied by the internal modules, prioritized and vectorized interrupts are supplied by external sources)

- Latency: 20 clock cycles
  + bus collisions + CPU state saving + tasking system overhead!
μControllers MPC565

NEXUS debug port (IEEE-ISTO 5001-1999)
(Real-time embedded application development interface)

On-Line mode:

- **Program trace**: via *branch trace messaging*
- **Data trace**: via *data write messaging* and *data read messaging* (can be reduced to selected areas)
- **Owner trace**: via *ownership trace messaging* (also indicates task creation and activation)
- Run-time access to *memory map* and *special CPU registers*
- **Watchpoints**: CPU watchpoint status signals are snooped and transferred with high priority

Off-line mode:

- **Read / Write access**: the READI module can take over the L-bus to manipulate data
- Access to all *CPU registers* during halt
Language requirements for interfaces

Specify the device interface (protocol and formats) in all detail
(candidates: Modula-2, Ada95, CHILL, ERLANG, …
or Macro-Assemblers and C/C++ (if platform-independence is not required))

Handling asynchronous messages from the device
many different methods to implement a context-switch
(candidates: all languages with some process-orientation:
PEARL, CHILL, ERLANG, Ada95, RT-Java, POSIX, …)
Interfaces & Controllers

Language requirements for interfaces

☞ Specify the device interface (protocol and formats) in all detail
  (candidates: Modula-2, Ada95, CHILL, ERLANG, …
  or Macro-Assemblers and C/C++ (if platform-independence is not required))

☞ Handling asynchronous messages from the device
  many different methods to implement a context-switch
  (candidates: all languages with some process-orientation:
  PEARL, CHILL, ERLANG, Ada95, RT-Java, POSIX, …)

☞ The term “high-level languages” in the real-time interface context:
  Allow for abstractions while being specific
  down to the actual level of interface realities
Basic sampling control mechanisms

- **Status driven:** The computer polls for information (used in dedicated µcontrollers and pre-scheduled hard real-time environments)

- **Interrupt driven:** The data generating device may issue an interrupt when new data had been detected / converted or when internal buffers are full
  - **Program controlled:** The interrupts are handled by the CPU directly (by changing tasks, calling a procedure, raising an exception, free tasks on a semaphore, sending a message to a task, ...)
  - **Program initiated:** The interrupts are handled by a DMA-controller. No processing is performed. Depending on the DMA setup, cycle stealing can occur and needs to be considered for the worst case computing times.
  - **Channel program controlled:** The interrupts are handled by a dedicated channel device. The data is transferred and processed. Optional memory-based communication with the CPU. the channel controller is usually itself a dedicated µengine / µcontroller.
Handling device responses

After setting up an interface device ...

... responses from devices can be:

- immediate
- with a constant delay or within a defined time-frame
- or unpredictable

The device handler may thus:

- perform a ‘busy-wait’ for the response
- reschedule the device-process by a constant delay
- schedule the device-process periodically and employ different time-slots for sending control / data and receiving status / data
- activate / trigger / call / signal the device-process by interrupts from the specific device
Handling device responses

The central issue for hard real-time environments:

How to embed the unpredictable in predictable systems?

Obviously: a infinite flood of interrupts cannot be accepted, so:

- either the asynchronous events need to be synchronized with the remaining system-tasks or
- there are exclusive processing resources (e.g. a dedicated µcontroller) for a specific device.
Summary

Converters & Interfaces

• Analogue signal chain in a digital system
  - Sampling data \(\rightarrow\) aliasing \(\rightarrow\) Nyquist's criterion \(\rightarrow\) oversampling
  - Quantization (LSB, rms noise voltage, SNR, ENOB)
  - Missing codes, DNL, INL

• A/D converters: flash, pipelined-flash, SAR, \(\Sigma-\Delta\), n-th order \(\Sigma-\Delta\)

• Examples:
  - Fast and simple A/D converter example: National Semiconductor ADC08200
  - Multi-channel A/D data logging interface example: National Semiconductor LM12L458
  - Simple 8-bit \(\mu\)controller example: Motorola MC68HC05
  - Complex 32-bit \(\mu\)controller example: Motorola MPC565
    TPU: \(\mu\)programming, atomic states, \(\mu\)engine scheduling, max. latency analysis
    NEXUS debugging port

• General device handling / sampling control / language requirements
Time & Space

Uwe R. Zimmer – The Australian National University
References for this chapter

[Ada95RM] (link to on-line version)
Ada Working Group
ISO/IEC JTC1/SC 22/WG 9
Ada 95 Reference Manual
– Language and Standard Libraries
ISO/IEC 8652:1995(E) with COR.1:2000,
June 2001

[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.rtj.org

[Burns01]
Alan Burns and Andy Wellings
Real-Time Systems and Programming Languages
Addison Wesley, third edition, 2001

[Dourish01]
Paul Dourish
Where the Action Is
MIT Press, Cambrudge, Massachusetts,

[Kligerman86]
E. Kligerman, A. D. Stoyenko
Real-Time Euclid:
A language for reliable real-time systems
IEEE transactions on Software Engineering SE-
12(9): 941-949

all references and links are available on the course page
Notions of time and space

The big topics:

What is time? / What is embodiment?

Interfacing with time

Specifying timing requirements

Satisfying timing requirements
Notions of time and space

What is time?

Do we exist in time, or is time part of our existence?

☞ Is time an intrinsic property of nature? ☞ Platonism

Time is an external phenomenon. Thus simultaneous events happen at the same exact absolute time. There is the underlying assumption that time is progressing, even if no changes can be observed.

☞ Is time a construct which is based on observable events? ☞ Reductionism

Time is the observation of distinguishable events. If the observed events are ‘regular’, a useful time-reference can be constructed. If all possible observers detect one event before another one, they are said to be in sequence. If this cannot be assumed for all possible observers, they are said to be simultaneous. Therefore the notion of time is reduced to a notion of causality.

☞ Is time ‘linear’ between observable events?
Real-Time & Embedded Systems

Notions of time and space

What is time?

A mathematical notion of time

- Transitivity: \( x < y \land y < z \Rightarrow x < z \)
- Linearity: \( x < y \lor x > y \Rightarrow x \neq y \)
- Irreflexivity: \( \neg (x < x) \)
- Density: \( x < y \Rightarrow \exists z \mid x < z \land z < y \)

Real clocks have limited resolutions and are running asynchronously!
Notions of time and space

What is time? – A physical notion of time

- 1676: Rømer proofs the existence of the speed of light (and measures it).

- 1687: Newton’s “Principia Mathematica” assumes an absolute time, independent of space itself and independent of events.

- 1905: The concept of absolute time is destroyed first by Einstein and (a few weeks later) by Poincaré.

- 1915: Einstein’s general theory of relativity eliminates the independence of time (space) and events in time (space).

☞ One principal consequence for measurements of time:

Clocks under higher gravity or in faster observation frames are slower

Practical consequences: clocks in satellites need to be adjusted accordingly.
### Notions of time and space

#### What time is it?

<table>
<thead>
<tr>
<th>Time System</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UT0</td>
<td>1884: Mean solar time at Greenwich meridian</td>
</tr>
<tr>
<td>UT1</td>
<td>Corrected UT0, polar motion</td>
</tr>
<tr>
<td>UT2</td>
<td>Corrected UT1, variations in the speed of rotation of the earth</td>
</tr>
<tr>
<td>IAT</td>
<td>A caesium 133 atomic clock (current accuracies: one miss in $10^{13}$ ticks, e.g. approximately once every 300,000 years)</td>
</tr>
<tr>
<td>UTC</td>
<td>A IAT clock, which is synchronized to UT2 (by introducing occasional leap ticks) – difference between UTC and IAT is $&lt; 0.5$ s</td>
</tr>
</tbody>
</table>

- ‘1 sec.’
  - $1/86400$ of a mean solar day
  - $1/31566925.9747$ of the tropical year for 1900 (Ephemeris Time defined 1955)
  - $9192631770$ periods of the radiation corresponding to the transition between two hyperfine levels of the ground state of the caesium 133 atom
Notions of time and space

What is time?

How real is real-time?

• ‘Real’-time ☞ usually: time as given by external sources.

*Engineering:* it is of no importance how time is defined and understood, as long as an ‘external reference’ is given and used as ‘real’-time.
Notions of time and space

What time is it?

Time frames:

- Generating a time frame?
  - by a timer generating a regular interrupt
  - by employing a RTC-module

- Using an existing time-frame?
  - by employing time-stamps or sequence numbers of received sensor-readings
  - by a radio receiver for UTC or IAT (available in some countries)
Notions of time and space

What time is it?

Generating a time frame?

- by a timer generating a regular interrupt
- by employing a RTC-module

Using an existing time-frame?

- by employing time-stamps or sequence numbers of received sensor-readings
- by a radio receiver for UTC or IAT (available in some countries)

Common programming languages guarantee a ‘resolution’ and ‘accuracy’ of time (in reference to ‘a second’), not its origin or meaning.
Notions of time and space

What is embodiment?

Working hypothesis:

☞ Embodied phenomena are those that by their very nature occur in real time and real space.
Notions of time and space

What is embodiment?

Phenomenology:

- The phenomena of experience as the central aspects and building blocks of understanding.

  (rather than: “finding the ‘truth’”)

applied to and trying to combine aspects of

☞ **Ontology** (about the nature of being and categories of existence)

☞ **Epistemology** (the study of knowledge)
Real-Time & Embedded Systems

Notions of time and space – Phenomenology

**What is embodiment?**

Edmund Husserl (1859-1938, Vienna, Halle, Göttingen, Freiburg):

- Founder of the phenomenological tradition
  ... as a trial to establish modern science which is firmly grounded on the phenomena of experience (instead of being an abstract mathematical construct).
- Phenomenology originally as a method to examine the nature of intentionality
- Coined the terms
  - **Noema**: the objects of consciousness
  - **Noesis**: the mental experiences of those objects
  - **Lebenswelt** (life-world): the inter-subjective world of everyday-experience

☞ Husserl rejected pure abstract and formalized reasoning
What is embodiment?

Martin Heidegger (1889-1976, Freiburg):

- Moved phenomenology from a discussion about mental phenomena separated from the physical world (Cartesian dualism) to a discussion about connected physical and mental phenomena.
- Moved the central questions from epistemology to ontology (‘Being and Time’, 1927)

The meaning is not ‘in the head’ but in the world!

- Coined terms:
  - **Dasein** (being-in-the-world): ‘being as inseparable from the world in which it occurs’
    - being as always purposeful and active
    - the world as an unconscious but accessible background
  - **Zuhanden** (ready-to-hand): ‘equipment as a part of actual interaction with the world’
  - **Vorhanden** (present-at-hand): ‘equipment as a conscious model’
Notions of time and space – Phenomenology

What is embodiment?

Maurice Merleau-Ponty (1908-1961, Paris (Sorbonne)):

• ‘The Phenomenology of Perception’ (1945)
• Embodiment has three implications:
  • a body as a physical entity
  • a body as a set of physical skills and situated responses gained from the physical world
  • a body as a set of ‘cultural skills’ gained from the cultural world in which it is embedded
• Embodied perception as a bi-directional sensation and a basis for empathy
  (Perception in itself does not exist).
☞ see also: Phenomenology of Jean-Paul Sartre

Recent works in robotics (and insights about biological sensors) blur the line between action and perception even further.
Notions of time and space

What is embodiment?

back to the working hypothesis:
☞ Embodied phenomena are those that by their very nature occur in real time and real space.

refinement:
☞ Embodiment is the property of any engagement with the real world which (may) makes this engagement meaningful.

(Paul Dourish)
☞ Embodied phenomena are the essence of meaningful interaction

(Real-time and embedded systems are the technical instantiations of embodiment)
Real-Time & Embedded Systems

Notions of time and space

What is embodiment?

Implications:

☞ There is no such thing as
‘intelligence’, ‘autonomy’ or any other cognitive process,
which is independent of a physical environment.

☞ There is no such thing as a
universal system or body (mechanical design, robot, device, …)
which is operational in all physical environments.
Notions of time and space

What is embodiment?

Meaningfully embedded systems are part of an ‘ecological niche’ (Rolf Pfeifer):

- The operational environment is supportive and employed by the system
- The embedded system is constructed as a part of the operational environment and according to the task
- The task is meaningful considering the morphology and cognitive ability of the system as well as the response from the environment

☞ i.e. being situated, embodied, and self-sufficient
Notions of time and space

What is embodiment?

- Embodied skills depend on
  a tight coupling between perception and action
  ... up to the level where the distinction between both can become difficult

- Tight coupling between perception and action means
to operate under real-time constraints
  ... and to construct meaningful morphologies
Notions of time and space

The big topics:

**What is time? / What is embodiment?**

**Interfacing with time**

**Specifying timing requirements**

**Satisfying timing requirements**
## Interfacing with time

### What time is it in …

<table>
<thead>
<tr>
<th></th>
<th>Resolution (syntactical)</th>
<th>Range (all time variables)</th>
<th>Requested resolution (clock ticks)</th>
<th>Actual resolution (detectable?)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA</td>
<td>ms</td>
<td>undefined</td>
<td>undefined</td>
<td>no</td>
</tr>
<tr>
<td>RT JAVA</td>
<td>ms, ns</td>
<td>undefined</td>
<td>undefined</td>
<td>yes</td>
</tr>
<tr>
<td>Ada95</td>
<td>ms, µs, ns</td>
<td>&gt;50 years</td>
<td>&lt;1 ms</td>
<td>yes</td>
</tr>
<tr>
<td>POSIX threads</td>
<td>ms, ns</td>
<td>undefined</td>
<td>&lt;20 ms</td>
<td>yes</td>
</tr>
<tr>
<td>C</td>
<td>integer (as seconds)</td>
<td>undefined</td>
<td>undefined</td>
<td>no</td>
</tr>
</tbody>
</table>
package Ada.Real_Time is
  type Time is private;
  Time_First : constant Time;
  Time_Last  : constant Time;
  Time_Unit  : constant := 10#1.0#E-9; -- ns

  type Time_Span is private;
  Time_Span_First : constant Time_Span;
  Time_Span_Last  : constant Time_Span;
  Time_Span_Zero  : constant Time_Span;
  Time_Span_Unit  : constant Time_Span;
  Tick : constant Time_Span; -- actual clock resolution < 1 ms

  function Clock return Time;
  (... operations on and conversions with time and time_span ...)
end Ada.Real_time
Real-Time & Embedded Systems

Notions of time in RT-Java

RT-Java time classes

Time root class:

```java
public abstract class HighResolutionTime implements java.lang.Comparable
```

direct known subclasses: `AbsoluteTime`, `RelativeTime`, `RationalTime`

- similar to Ada.Real-Time, but no requested accuracy
- adds the concept of frequency (‘rational time’),
  but does not guarantee for equidistant instantiations.

Clock Class:

```java
public abstract class Clock
{
    public static Clock getRealtimeClock ();
    public abstract RelativeTime getResolution () ;
    public abstract void setResolution (RelativeTime resolution); 
}
```
Notions of time in POSIX

Real-time clock interface in POSIX

```c
#define CLOCK_REALTIME ...; // clockid_t type

struct timespec {
    time_t tv_sec;    /* number of seconds */
    long   tv_nsec;   /* number of nanoseconds */
};

typedef ... clockid_t;

int clock_gettime (clockid_t clock_id,       struct timespec *tp);
int clock_settime (clockid_t clock_id, const struct timespec *tp);
int clock_getres   (clockid_t clock_id,       struct timespec *res);
int clock_getcpuclockid (pid_t pid , clockid_t *clock_id);
int clock_getcpuclockid (pthread_t thread_id, clockid_t *clock_id);
int nanosleep (const struct timespec *rqtp, struct timespec *rmtp);
/* nanosleep return -1 if the sleep is interrupted by a signal. In this case, rmt has the remaining sleep time */
```
Interfacing with time

Programming primitive ‘Delay’

- Alternative for ‘busy-wait’ and interrupts: Suspend a task for a fixed time
Interfacing with time

Programming primitive ‘Delay’

- Alternative for ‘busy-wait’ and interrupts: Suspend a task for a fixed time
  - but all these process or thread delays are precise only in their lower bound!

![Diagram showing actual normal delay, scheduled delay, interrupts disabled, and granularity difference.](image)
Interfacing with time

Programming primitive ‘Delay’

• Alternative for ‘busy-wait’ and interrupts: Suspend a task for a fixed time

  • **local drift:** sum of all additional actual delays

![Diagram showing intended delay, actual normal delay, and local drift.](image)
Interfacing with time

Programming primitive ‘Delay’

- Alternative for ‘busy-wait’ and interrupts: Suspend a task for a fixed time
  - if task $i$ is enabled for interrupts, it can also be activated earlier!

![Diagram showing intended delay and actual normal delay with scheduled, interrupts disabled, and granularity difference annotations.]
Interfacing with time

Relative delay

(Ada95)

task T;
task body T is
  begin
    loop
      Action;
      delay 5.0; -- sec.
    end loop;
  end T;

This loop will delay at least for 5 seconds:

local and cumulative drift effects here!
Interfacing with time

Relative as absolute delay?

(Ada95)

task body T is
  Interval: constant Duration:= 5.0; -- sec.
  Start_Time: Time;

  begin
    loop
      Start_Time:= Clock;
      Action;
      delay Interval - (Clock - Start_Time);
    end loop;
  end T;

Delay time calculation is not atomic!
Interfacing with time

Relative as absolute delay?

\[(Ada95)\]

```ada
task body T is
  Interval: constant Duration:= 5.0; -- sec.
  Start_Time: Time;

  begin
    loop
      Start_Time:= Clock;
      Action;
      delay Interval - (Clock - Start_Time);
    end loop;
  end T;
```

don't!

Delay time calculation is not atomic!
Interfacing with time

Absolute delay

(Ada95)

task body T is

Interval: constant Duration:= 5.0; -- sec.
Next_Time: Time;

begin

Next_Time:= Clock + Interval;
loop

Action;

delay until Next_Time;
Next_Time:= Next_Time + Interval;
end loop;
end T;

note that this also holds, if ‘Action’ is sporadically longer than 5 s

This loop will delay on average for 5 seconds:
Interfacing with time

Zero delay?

(Ada95)

task body T is
  begin
    loop
      Action;
      delay 0.0;
      end loop;
  end T;

• the delay statement does not only suspend the current task, but also potentially activates other tasks
☞ may be used to enable other processes on the same priority level

Allow explicitly for a task-switch
Interfacing with time

Delays

Absolute & relative delays are available in:
- Real-Time Java, Pearl, Ada95, … and many other process-oriented languages

Only relative delays are available in some ‘low-level’ environments:
- POSIX: `nanosleep` (absolute delays need to be constructed employing timers and signals)

Only absolute delays are available in some ‘harder’ real-time environments:
- Occam2, Ada95 (Ravenscar profile), …
Interfacing with time

Timeouts

As a third alternative to busy-waiting and infinite blocking, timeouts are implemented in:

- **Shared variable** communications
  - Semaphore
  - Conditional critical regions
  - Monitors
  - Protected objects

- **Message passing** between processes
  - Asynchronous and synchronous message transfers
  - Remote procedure calls
  - Remote objects

- Actions
Interfacing with time

Timeouts on semaphore

POSIX:

```c
if (sem_timedwait (&call, &timeout) < 0) {
    if (errno == ETIMEDOUT) {
        /* timeout occurred, try something else */
    }
    else {
        /* some other error occurred, do something about it */
    }
} else {
    /* semaphore is locked successfully, go ahead */
}
```

Suspend current process until the semaphore `call` is open or the time-span `timeout` has passed
Interfacing with time

Timeouts on entry calls

(same for task-entry calls (message passing) and protected object calls (monitors))

task body Sensor is
T : Temperature;
begin
  loop
    -- find temperature T somewhere
    select
      Controller.Call(T);
    or
      delay 0.5; -- sec.
    -- action if temperature could not be delivered in time
    end select;
  end select;
end loop;
end Sensor;

Try calling for 500 ms
Interfacing with time

Timeouts on incoming calls

task body Controller is
  Current_Temp : Temperature;
begin
  loop
    select
      accept Call (T: Temperature) do
        Current_Temp := T;
      end Call;
    or
      delay 1.0; -- sec.
      -- action if the temperature was not available in time
    end select;
    -- normal processing
  end loop;
end Controller;

accept calls for a limited time-span of 1s
Interfacing with time

Timeouts on incoming calls

task body Controller is
   -- declarations
begin
   loop
      select
         accept Call (T: Temperature) do
            Current_Temp := T;
            end Call;
      or
         delay until Deadline;
      end select;
      -- no further calls before the deadline
      end select;
   end loop;
   end Controller;

accept any number of calls until a closing time ‘Deadline’ for this entry
Interfacing with time

No-wait on incoming calls

```vhdl
task body Controller is
  -- declarations
begin
  loop
    select
      accept Shutdown do
        -- termination actions
        exit;
      end Shutdown;
    else
      -- normal operation
    end select;
  end select;
  -- synchronize
  end loop;
end Controller;
```

synchronous alternative for an interrupt acceptance
Interfacing with time

No-wait on entry calls

(same for task-entry calls (message passing) and protected object calls (monitors))

task body Sensor is
T : Temperature;
begin
  loop
    -- measure temperature T
    select
      Controller.Call(T);
    else
      -- action if temperature can not yet be delivered,
      -- e.g. further refine the measurement
      end select;
  end loop;
end Sensor;

Try delivering, else refine the result further
Interfacing with time

Timeout on actions

• All above timeouts suspend / activate a process / task / thread at a synchronization point.
• Up to now, there wasn’t any abortion of on-going actions, due to a timeout
to achieve this there need to be an
☞ asynchronous change of control flows
• on the level of code-blocks: ☞ Timeout on actions
• on the level of processes / tasks / threads: (investigated later)
Monitor a code block:

```plaintext
select
delay until Deadline;

-- computations did not finish in time: take measures
then abort

-- hard to predict sequence of computations
end select;
```
Interfacing with time

Timeout on actions

☞ Common RT-systems concept:

**Timeliness** is often more important than **Precision**

1. **Get a first approximation** in fixed amount of time and well before the deadline

2. Inspect the deadline and if there is enough spare time:

3. **Improve the result** and keep a record of improvements (while keeping an eye on the deadline)
   
   3-a If the most precise result is delivered before the deadline occurs: ✔
   3-b else use the closest approximation so far: ✔

☞ The deadline is fulfilled and there is a result in any case
Interfacing with time

Timeout on actions

Get a first approximation and employ spare time for refinements:

Deadline := ... -- set an absolute deadline for the computations
-- compulsory computations (save first result)

select
  delay until Deadline;
  Precise_Result := False;
then abort
  while Result_Can_Be_Improved loop
    -- optimising computations (save results after each iteration)
    end loop;
  Precise_Result := True;
end select;
-- use result

Take a first guess

improve, improve, improve, ...

Continue in time and with the best result possible
Interfacing with time

Timeout on actions

Time-base can also be given externally, e.g., via a protected call:

```
loop
  select
    Get_New_Data (Current_Sensor_Data);
    -- employ precise results based on previous data
    -- compulsory computations (save first result)
    -- employ first result on current data
  then abort
    while Result_Can_Be_Improved loop
      -- optimising computations (save results after each iteration)
    end loop;
  end select;
end loop;
```

- New data?  ☞  abort current iterations
- Precise reactions
- Take a first guess
- First reactions
- Improve, – improve, – improve, …
Interfacing with time

Timeout on actions

(RT-Java)

Similar methods in RT-Java:

```java
public class Timed extends AsynchronouslyInterruptedException
    implements java.io.Serializable {

    public Timed (HighResolutionTime time) throws IllegalArgumentException;

    public boolean doInterruptible (Interruptible logic);

    public void resetTime (HighResolutionTime time);

}
```

☞ see full example of imprecise computations in RT-Java in the course textbook.

(timeouts on actions in POSIX need to be emulated employing timers and signals)
Notions of time and space

The big topics:

What is time? / What is embodiment?

Interfacing with time

Specifying timing requirements

Satisfying timing requirements
Specifying timing requirements

Temporal scopes

Common attributes:

- Minimal & maximal delay after creation
- Maximal elapsed time
- Maximal execution time
- Absolute deadline
Real-Time & Embedded Systems

Specifying timing requirements

Temporal scopes

Common attributes:

- Minimal & maximal delay after creation
- Maximal elapsed time
- Maximal execution time
- Absolute deadline

Diagram:
- Task $i$
- Time line: $t$
- Created: 1
- Activated: 10
- Maximal delay
- Minimal delay
- Maximal execution time
- Maximal elapsed time
- Absolute deadline
Specifying timing requirements

Temporal scopes

Common attributes:

- Minimal & maximal delay after creation
- Maximal elapsed time
- Maximal execution time
- Absolute deadline
Specifying timing requirements

Temporal scopes

Common attributes:
- Minimal & maximal delay after creation
- Maximal elapsed time
- Maximal execution time
- Absolute deadline
Specifying timing requirements

Some common scope attributes

Temporal Scopes can be:

<table>
<thead>
<tr>
<th>Periodic</th>
<th>e.g. controllers, samplers, monitors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aperiodic</td>
<td>‘periodic on average’ tasks, burst requests</td>
</tr>
<tr>
<td>Sporadic / Transient</td>
<td>mode changes, occasional services</td>
</tr>
</tbody>
</table>

Deadlines (absolute, elapse, or execution time) can be:

<table>
<thead>
<tr>
<th>Hard</th>
<th>single failure leads to severe malfunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firm</td>
<td>results are meaningless after the deadline</td>
</tr>
<tr>
<td>Soft</td>
<td>only multiple or permanent failures threaten the whole system</td>
</tr>
<tr>
<td></td>
<td>results may still be useful after the deadline</td>
</tr>
</tbody>
</table>
Specifying timing requirements

Some common scope attributes

Temporal Scopes can be:

<table>
<thead>
<tr>
<th>Periodic</th>
<th>e.g. controllers, samplers, monitors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aperiodic</td>
<td>e.g. ‘periodic on average’ tasks, burst requests</td>
</tr>
<tr>
<td>Sporadic / Transient</td>
<td>e.g. mode changes, occasional services</td>
</tr>
</tbody>
</table>

Deadlines (absolute, elapse, or execution time) can be:

<table>
<thead>
<tr>
<th>Hard</th>
<th>single failure leads to severe malfunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firm</td>
<td>deadline</td>
</tr>
<tr>
<td>Soft</td>
<td>distinction is not so obvious in practical systems</td>
</tr>
<tr>
<td>Firm</td>
<td>deadline</td>
</tr>
<tr>
<td>Soft</td>
<td>results may still be useful after the deadline</td>
</tr>
</tbody>
</table>
Specifying timing requirements

Language support for specifying temporal scopes

- Real-time Euclid
- CRL
- Pearl
- Esterel
- DSP
- Ada95
- POSIX
- Real-time Java
- Real-time Euclid
- CRL
- Pearl
- Esterel
- DSP
- Ada95
- POSIX
- Real-time Java
Specifying timing requirements

Real-time Euclid

Language features:

- Recursions and goto-statements are prohibited.
- Loops are restricted to time bounded loops.
- Processes are static and non-nested.

Time scopes:

- periodic <Frameinfo> first activation <TimeOrEvent>
- atEvent <ConditionalId> <Frameinfo>
Specifying timing requirements

Real-time Euclid

Periodic process example:

\[
\text{realTimeUnit} := 1.0 \% = 1 \text{ seconds}
\]
\[
\text{var Reactor} : \text{module}
\]
\[
\text{var startMonitoring} : \text{activation condition atLocation 16#A10D}
\]
\[
\text{process TempController} : \text{periodic}
\]
\[
\quad \text{frame 60}
\]
\[
\quad \text{first activation atTime 600}
\]
\[
\quad \text{or atEvent startMonitoring}
\]
\[
\% \text{import list}
\]
\[
\% \text{execution part}
\]
\[
\text{end TempController}
\]
\[
\text{end Reactor}
\]
Specifying timing requirements

Real-time Euclid

task body Temp_Controller is
    Next_Release : Duration;
begin
    select accept Start_Monitoring;
    or     delay 600.0; -- sec.
    end select;
    Next_Release := Clock + 60.0; -- sec.
    loop
        -- execution part
        delay until Next_Release;
        Next_Release := Next_Release + 60.0; -- sec.
    end loop;
end Temp_Controller;

Ada-emulation of the above RT-Euclid program
task body Temp_Controller is

  Next_Release : Duration;

begin

  select accept Start_Monitoring;
  or     delay 600.0; -- sec.
  end select;

  Next_Release := Clock + 60.0; -- sec.

  loop
      -- execution part
      delay until Next_Release;
      Next_Release := Next_Release + 60.0; -- sec.
  end loop;

end Temp_Controller;

Ada-emulation of the above RT-Euclid program

but:
no formal time-scope specification!
no schedulability analysis!
Real-Time & Embedded Systems

Specifying timing requirements

Real-time Euclid

State:

Real-Time Euclid was suggested by Kligerman and Stoyenko in 1986
☞ Additional schedulability analysis modules became available
☞ Stayed in an academic context, but influenced many more recent RT-systems.
Specifying timing requirements

CRL
(a language for complex real-time systems)

Ways to handle ‘dangerous constructs’ (loops, recursions, synchronisations, …):

• Exclude them
  ☞ Real-time Euclid

• Expand them to become individually safe
  (e.g. by adding mandatory timeout)

• Attribute the code with additional constraints,
  enabling a full pre-runtime analysis ☞ CRL
Specifying timing requirements

CRL
(a language for complex real-time systems)

Constraints in CRL on:

- **Time**:
  \[\text{timeconstraint} \{\text{use} | \text{nosoonerthan} | \text{nolaterthan} \langle\text{abs\_time}\rangle\} \text{endtimeconstraint}\]
  (also relative constraints)

- **Iterations**: assert lower and upper limits for the number of iterations per block

- **Activations**:
  \[\text{activationdeactivationconstraint} \]
  \[\text{periodic } \langle\text{Frameinfo}\rangle \text{ firstactive } \langle\text{TimeOrEvent}\rangle \]
  \[\lor \text{ atEvent } \langle\text{ConditionalId}\rangle \langle\text{Frameinfo}\rangle\]
  \[\text{endactivationdeactivationconstraint}\]

- **Direct recursions**: assert lower and upper recursion limits (general recursions are not covered).
Specifying timing requirements

CRL

(a language for complex real-time systems)

Evaluation of the constraints and assertions:

- **Timing:**
  - verified at compile-time

- **Activation / Deactivation:**
  - checked for schedulability at compile-time and enforced by the scheduler at run-time.

- **Iterations and recursion:**
  - either verified at compile-time or checked at run-time.
Specifying timing requirements

CRL
(a language for complex real-time systems)

State:

CRL was suggested by Stoyenko, Marlowe and Younis in 1995

- Full featured language
- Compiled to attributed C++

Stayed also in an academic context.
Specifying timing requirements

Pearl

Explicit time-scope expressions:

\[
\text{TaskStart} ::= \text{[StartCondition]} \text{ ACTIVATE } \langle \text{task} \rangle
\]

\[
\text{StartCondition} ::= \text{AT } \langle \text{time} \rangle \text{ [Frequency]} | \\
\text{AFTER } \langle \text{duration} \rangle \text{ [Frequency]} | \\
\text{WHEN } \langle \text{interrupt} \rangle \text{ [AFTER } \langle \text{duration} \rangle \text{ [Frequency]]} | \\
\text{Frequency}
\]

\[
\text{Frequency} ::= \text{ALL } \langle \text{duration} \rangle \\
\quad \{\{\text{UNTIL } \langle \text{time} \rangle \} | \\
\quad \{\text{DURING } \langle \text{duration} \rangle\}\}
\]

☞ Schedulability analysis (at \textit{compile-time} or \textit{run-time}) possible
(although not defined by the language)

☞ Pearl refinement: a combination of Pearl and Real-time Euclid \textit{High-Integrity Pearl}
Specifying timing requirements

DSP: Distributed Programming System

Explicit time-scope expressions at the ‘statement level’:

E.g. time-scope for a software-engineer:

```
from 11:00 to 19:30 every 45 do
    start elapse 10 do
        setup_coffee_machine
        power_coffee_machine_up
        find_favorite_cup
        put_coffee_in_favorite_cup
        clean_coffee_machine
    end
    start after 3 elapse 25 by 20:00 do
        drink_coffee
    end
end
```

DSP-compiler: breaks the sequences down in processes and schedules.
Real-Time & Embedded Systems

Specifying timing requirements

Real-time Java

Real-time Java comes with:

- multiple sets of predefined time-scope parameters
- a scheduler class (with a predefined priority scheduler)

☞ Schedulability (feasibility) analysis possible.
Formulates an on-line schedulability analysis!
Specifying timing requirements

Real-time Java

```java
public class PriorityScheduler extends Scheduler {
    public static final int MAX_PRIORITY;
    public static final int MIN_PRIORITY;

    protected PriorityScheduler ();

    protected boolean addToFeasibility (Schedulable s);

    public void fireSchedulable (Schedulable s);

    public boolean isFeasible ();

    protected boolean removeFromFeasibility (Schedulable s);

    public boolean setIfFeasible (Schedulable s, ReleaseParameters r, MemoryParameters m);

    ...
}
```

This PriorityScheduler is the only requested instantiation.
Real-Time & Embedded Systems

Specifying timing requirements

Real-time Java

```java
public class RealtimeThread extends java.lang.Thread
    implements Schedulable {
    public RealtimeThread (SchedulingParameters s,
                           ReleaseParameters r,
                           MemoryParameters m,
                           MemoryArea a);

    public synchronized void addToFeasibility ();
    public synchronized void addIfFeasible ();
    public static RealtimeThread currentRealtimeThread () throws ...
    deschedulePeriodic ();
    public boolean waitForNextPeriod () throws …;
    public synchronized void interrupt ();
    public static void sleep (…) throws …;
    …
}
```

Priority

Periodic, aperiodic, or sporadic parameters
Specifying timing requirements

Real-time Java

```java
public class NoHeapRealtimeThread extends RealtimeThread {
    public RealtimeThread (SchedulingParameters s,
                           ReleaseParameters r,
                           MemoryArea a) throws ...
    }

This thread is allowed to interrupt any
garbage collector at any time!
(since it doesn’t depend on it itself)
```
Specifying timing requirements

Real-time Java

```java
public abstract class SchedulingParameters {
    public SchedulingParameters ();
}

public class PriorityParameters extends SchedulingParameters {
    public PriorityParameters (int priority);
    public int getPriority ();
    public void setPriority (int priority) throws ...
    ...
}
```

‘Priority’ is the only default scheduling parameter
public abstract class ReleaseParameters {
    protected ReleaseParameters (RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler);
    public RelativeTime      getCost();
    public AsyncEventHandler getCostOverrunHandler();
    public RelativeTime      getDeadline();
    public AsyncEventHandler getDeadlineMissHandler();
}

Cost is an estimate of the max. execution time
Measuring execution time is not requested, i.e. the overrunHandler might never be activated!
Specifying timing requirements

Real-time Java

```java
public class PeriodicParameters extends ReleaseParameters {
    public PeriodicParameters
        (HighResolutionTime start,
         RelativeTime period,
         RelativeTime cost,
         RelativeTime deadline,
         AsyncEventHandler overrunHandler,
         AsyncEventHandler missHandler);

    public RelativeTime getPeriod ();
    public HighResolutionTime getStart ();
    public void setPeriod (RelativeTime period);
    public void setStart (HighResolutionTime start);
}
```

most frequently used release parameters
Real-Time & Embedded Systems

Specifying timing requirements

Real-time Java

```java
public class AperiodicParameters extends ReleaseParameters {
    public AperiodicParameters (RelativeTime cost,
                                  RelativeTime deadline,
                                  AsyncEventHandler overrunHandler,
                                  AsyncEventHandler missHandler);
}
```

these are the minimum release parameters (while cost might be used for feasibility analysis only)

the deadline-missHandler need to be supplied in any implementation
Real-Time & Embedded Systems

Specifying timing requirements

Real-time Java

```java
public class SporadicParameters extends AperiodicParameters {
    public SporadicParameters (RelativeTime minInterarrival,
                               RelativeTime cost,
                               RelativeTime deadline,
                               AsyncEventHandler overrunHandler,
                               AsyncEventHandler missHandler);

    public RelativeTime getMinimumInterarrival () ;
    public void setMinimumInterarrival (RelativeTime minimum);
}
```

Sporadic events are not allowed to come in bursts!
Specifying timing requirements

Real-time Java

The minimal required implementation supplies:

☞ Priority scheduling
☞ On-line schedulability analysis
☞ Deadline violation handlers
☞ (max. execution time deadline checks are suggested but not required)
☞ a sporadic scheduler is not required (although the sporadic release parameter set is).

Hard real-time environments require the exclusive usage of
‘No heap real-time threads’
(synchronization with an ‘object-oriented’ thread invalidates the feasibility assurances)
Real-Time & Embedded Systems

Specifying timing requirements

Ada95

Ada95 has *no* explicit time-scope expressions at task-level.

Ada95 offers ...

- tasking
- a priority scheduler (the only required scheduler)
- synchronisation and communication primitives
- asynchronous transfer of control, timed calls, timeout on actions ... ... ... etc. pp.
- ... but no hardware timers!

... to create the basis for most kinds of hard real-time-scopes manually.

but no automatic schedulability analysis!
Since Esterel is a synchronous language, ... all actions and communications take zero time by definition.

☞ There is no expression for continuous, non-zero time-scopes.
☞ Time is interpreted in the reductionistical way as a sequence of events
☞ Time-scopes translate to signal-relations and signal-counters

Continuous time scopes need to be taken into account while
1. **analysing and reducing** the problem to a zero-time atomic system
2. **implementing** the synchronous system on an actual system.

☞ Continuous time-scopes for the ...

... **validation** of the zero-time assumption!
Specifying timing requirements

POSIX

the usual:  

common combination:

usage of Ada95 together with POSIX timers as a basis for hard-real-time-scopes and schedulers
Notions of time and space

The big topics:

What is time? / What is embodiment?

Interfacing with time

Specifying timing requirements

Satisfying timing requirements
Satisfying timing requirements

Two paths towards fulfilling rt-requirements:

☞ Real-time logic approach
  formal, correct in its specifications, & offers calculus for asynchronous, real-time systems
  \- needs to ignore most real world effects, like jitters, drifts, failures, interferences, etc. pp.
  \- gives a correct solution according to the specification

☞ Complex systems oriented approach
  deals with existing computer systems, sensors, & offers a set of approximating methods
  \- not complete or correct in any formal sense
  \- deals with real-world systems, gives ‘robust’ systems, passes rigorous experiments
Satisfying timing requirements

Fulfilling rt-requirements:

Complex systems oriented approach:

- System identification and compile-time analysis:
  - Calculate or limit statement durations
  - Calculate or limit iterations and recursions
  - Analyse potential dead- or life-locks \textit{chapter 8}
  - Calculate schedulability \textit{chapter 7}

- Run-time analysis and checks:
  - Dynamic scheduling schemes: Re-validate schedulability \textit{chapter 7}
  - Check for all constraints and assertions at run-time \textit{chapter 9}

- Supply fault-tolerant behaviours:
  - Error recoveries, mode changes, … \textit{chapter 9}
Satisfying timing requirements

Fulfilling rt-requirements:

Real-time logic approach:

- **Reduce the problem:**
  - Reduce any asynchronous, analogue, dynamical, fractal, jitter-, drift, or failure-affected parts of the system to a fully synchronous and discrete system. *(chapters 5 and 6)*
  - Formulate the specification on the basis of the reduced synchronous system.

- **Verify the reduced system:**
  - Verify the reduced synchronous against the specifications. *(not covered in this course)*

- **Compile the reduced system to an actual system:**
  - The resulting actual system will be executable on a real machine and employ real devices

  Re-check the actual system (e.g. by means of a complex systems-approach) ...
Summary

Time & Space

- What is time? / What is embodiment?
  - Approaches by different faculties to understand the basis for this course

- Interfacing with time
  - Formulating local time-dependent constraints
  - Access time, delay processes, detect timeouts (in different languages)

- Specifying timing requirements
  - Formulating global timing-constraints
  - Understanding time-scope parameters (and expressing them in different languages)

- Satisfying timing requirements
  - Real-time logic and complex systems approach
Asynchronism

Uwe R. Zimmer – The Australian National University
References for this chapter

[Ada95RM] (link to on-line version)
Ada Working Group
ISO/IEC JTC1/SC 22/WG 9
Ada 95 Reference Manual
– Language and Standard Libraries
ISO/IEC 8652:1995(E) with COR.1:2000,
June 2001

[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.rtj.org

[Burns01]
Alan Burns and Andy Wellings
Real-Time Systems and Programming Languages
Addison Wesley, third edition, 2001

all references and links are available on the course page
Asynchronism

Interrupts

Required mechanisms for interrupt driven programming:

- **Interrupt control**: grouping, encoding, prioritising, and en-/disabling interrupt sources

- **Context switching**: mechanisms for cpu-state saving and restoring + task-switching

- **Interrupt identification**: Interrupt vectors, interrupt states

hardware-supported
Asynchronism

Interrupts

Interrupt control:

... at the individual device level

... at the system interrupt controller level

... at the operating system level
## LM12L458 – accessible registers

| A4 A3 A2 A1 | Purpose | Type | D15 D14 D13 D12 D11 | D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0 |
|------------|---------|------|----------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 0 0    | Instruction RAM (RAM Pointer = 00) | R/W | Acquisition Time | Watchdog | Timer | Sync | V<sub>IN-</sub> | V<sub>IN+</sub> | Pause | Loop |
| 0 0 0 0    | Instruction RAM (RAM Pointer = 01) | R/W | Don’t Care | >/≤ | Sign | Limit #1 |
| 0 0 0 0    | Instruction RAM (RAM Pointer = 10) | R/W | Don’t Care | >/≤ | Sign | Limit #2 |
| 1 0 0 0    | Configuration Register | R/W | Don’t Care | DIAG | Test = 0 | RAM Pointer | I/O Sel | Auto Zero<sub>eq</sub> | Chan Mask | Stand-by | Full CAL | Auto-Zero | Reset | Start |
| 1 0 0 1    | Interrupt Enable Register | R/W | Number of Conversions in Conversion FIFO | Address to Generate INT1 | INT7 | Don’t Care | INT5 | INT4 | INT3 | INT2 | INT1 | INT0 |
| 1 0 1 0    | Interrupt Status Register | R | Actual Number of Conversion Results in Conversion FIFO | Address of Sequencer Instruction being Executed | INST7 | “0” | INST5 | INST4 | INST3 | INST2 | INST1 | INST0 |
| 1 0 1 1    | Timer Register | R/W | Timer Preset High Byte | Timer Preset Low Byte |
| 1 1 0 0    | Conversion FIFO | R | Address or Sign | Conversion Data: LSBs |
| 1 1 0 1    | Limit Status Register | R | Limit #2: Status | Limit #1: Status |
### LM12L458 – interrupt registers

| A4 A3 A2 A1 | Purpose                        | Type | D15 D14 D13 D12 D11 | D10 D9 D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|-------------|--------------------------------|------|----------------------|-----------|---------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 1 0 0 1     | Interrupt Enable Register      | R/W  | Number of Conversions in Conversion FIFO to Generate INT2 |           | Sequencer Address to Generate INT1 | INT7 | Don’t Care | INT5 | INT4 | INT3 | INT2 | INT1 | INT0 |
|             |                                | R    | Actual Number of Conversion Results in Conversion FIFO |           | Address of Sequencer Instruction being Executed | INST7 | "0" | INST5 | INST4 | INST3 | INST2 | INST1 | INST0 |

- **Select interrupt sources** (interrupt enable register, 15 bits):
  - **Watchdog**: limit conditions are fulfilled
  - **Instruction**: the instruction pointer equals a pre-programmed value (bits 8-10)
  - **Conversions**: a specified number of conversions (bits 11-15) have been performed
  - **Auto-Zero**: short calibration has been performed
  - **Full-Calibration**: long calibration has been performed
  - **Pause**: Sequencer arrived at a pause state
  - **Active**: Controller returned from power-down to active mode
**LM12L458 – interrupt registers**

| A4 | A3 | A2 | A1 | Purpose | Type | D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|----|----|----|----|---------|------|-----|-----|-----|-----|-----|-----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 1  | Interrupt Enable Register | R/W |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|    |    |    |    | Number of Conversions in Conversion FIFO to Generate INT2 | R/W |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 1  | 0  | 1  | 0  | Interrupt Status Register | R   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|    |    |    |    | Actual Number of Conversion Results in Conversion FIFO | R   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

- **Read interrupt status** (interrupt status register, 15 bits):
  - indicates the current interrupt conditions, incl. the actual number of conversions and the currently processed instruction
  - all of the status bits (0-5, 7) are reset with any read access to this register!
Interrupts

**LM12L458**  
(National Semiconductor)

☞ only one interrupt signal line available!
☞ in order to identify the interrupt reason, an additional read cycle is required!
Asynchronism

Interrupts

Interrupt control:

... at the individual device level

... at the system interrupt controller level

... at the operating system level
Asynchronism

Interrupts

System interrupt controller:

- collects the interrupt signal lines from all managed devices
- identifies the device status, if not given already as an interrupt vector
- encodes all interrupt signals into a common interrupt vector or status scheme
- orders and masks all interrupts according to priority levels
- triggers the actual CPU or controller interrupt

☞ three common forms:

- the interrupt controller as an intrinsic part of a complex µcontroller
- a dedicated interrupt controller for a set of similar or identical devices (e.g. a hard disk array)
- a universal interrupt controller (usually unable to fetch interrupt status information)
Asynchronism

Interrupts

Interrupt control:

... at the individual device level

... at the system interrupt controller level

... at the operating system level
  • beyond task-level
  • communicating interrupts to task
  • transforming interrupts to signals
Asynchronism

Interrupt service routines

(available only in some OSs, e.g. VxWorks)

Purpose:

- Allow full access to the interrupt controller (interrupt vectors, priorities).
- Change to an interrupt service routine in a predictable amount of time.

☞ Cannot operate on the level of threads or tasks!

☞ Limitations regarding the accessibility of some OS-facilities (task level system calls).

- Real-time-operating systems and real-time-languages provide this access.
Asynchronism

Interrupt service routines

(available only in some OSs, e.g. VxWorks)

Some VxWorks OS entries:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intConnect</td>
<td>Connect a routine to an interrupt vector</td>
</tr>
<tr>
<td>intLevelSet</td>
<td>Set the interrupt mask level</td>
</tr>
<tr>
<td>intLock</td>
<td>Disable interrupts (besides NMI)</td>
</tr>
<tr>
<td>intUnlock</td>
<td>Enable interrupts</td>
</tr>
<tr>
<td>intVecBaseSet</td>
<td>Set the interrupt vector base address</td>
</tr>
<tr>
<td>intVecBaseGet</td>
<td>Get the interrupt vector base address</td>
</tr>
<tr>
<td>intVecSet</td>
<td>Set an interrupt vector</td>
</tr>
<tr>
<td>intVecGet</td>
<td>Get an interrupt vector</td>
</tr>
</tbody>
</table>

these calls are employed by the language run-time environment or used directly from ‘C’-code
Asynchronism

Interrupt service routines

(available only in some OSs, e.g. VxWorks)

Minimal hardware support (supplied by the cpu):

- save essential CPU registers (IP, condition flags)
- jump to the vectorized interrupt service routine

Minimal wrapper (supplied by the real-time-os):

- save remaining CPU registers
- save stack-frame
- execute user level interrupts service code
- restore stack-frame
- restore CPU registers
- restore IP
Asynchronism

Interrupt service routines

(available only in some OSs, e.g. VxWorks)

Interrupt service routine to task communication methods:

- **Shard memory and ring buffers**: most low level communication scheme (should be avoided)
- **Semaphore**: trigger a semaphore, where a task has been blocked before.
- **Monitors**: free a task, which is blocked at a monitor entry (standard Ada-method: protected object).
- **Message queues**: Send messages to a task (if queue is not full).
- **Pipes**: Write to a pipe (if pipe is not full).
- **Signals**: indicate an asynchronous task switch to the scheduler

☞ in all of the above: the interrupt service routines cannot block!
Asynchronism

Interrupts \(\Rightarrow\) ‘Signals’

Interrupt control:

... at the individual device level

... at the system interrupt controller level

... at the operating system level

- beyond task-level
- communicating interrupts to task
- transforming interrupts to signals
Asynchronism

Interrupts ⇨ ‘Signals’

Some characteristics of signals:

- Involve a full task-switch operation
- Hard to predict timing behaviour
- Limited information about the interrupt-source
- Traditionally used to ‘kill’ processes
- Concept stems from a time before thread models, therefore the signal-to-thread propagation is implementation dependent and sometimes tricky.
## Asynchronism

### Interrupts \(\Leftrightarrow\) ‘Signals’

Some common UNIX OS entries:

<table>
<thead>
<tr>
<th>POSIX 1003.1b</th>
<th>BSD-UNIX</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal (...)</td>
<td>signal (...)</td>
<td>Specify the handler associated with a signal</td>
</tr>
<tr>
<td>sigaction (...)</td>
<td>sigvec (...)</td>
<td>Examine or set the signal handler for a signal</td>
</tr>
<tr>
<td>kill (...)</td>
<td>kill (...)</td>
<td>Send a signal (overwrite all other pending signals)</td>
</tr>
<tr>
<td>sigqueue (...)</td>
<td>N/A</td>
<td>Send a queued signal</td>
</tr>
<tr>
<td>sigsuspend (...)</td>
<td>pause (...)</td>
<td>Wait for a signal</td>
</tr>
<tr>
<td>sigwaitinfo (...)</td>
<td></td>
<td>Wait for a signal, but do not involve the handler</td>
</tr>
<tr>
<td>sigtimedwait (...)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sigemptyset (...)</td>
<td>sigsetmask (...)</td>
<td>Manipulate and set the mask of blocked signals</td>
</tr>
<tr>
<td>sigprocmask (...)</td>
<td>sigblock (...)</td>
<td>Add to a set of blocked signals</td>
</tr>
</tbody>
</table>
Asynchronism

Interrupts ⇸ ‘Signals’

• Signals are originally process-level synchronization methods (‘kill’) and have been expanded to be used for everything from hardware-interrupts and timers to asynchronous task messaging.

☞ Signals are passed through a global task-scheduler.
☞ in many OSs: unpredictable ‘work-arounds’ for missing direct hardware interrupt propagation.

☞ make sure that you understand the attached strings in your OS, before employing any signals.
Asynchronism

Structured interruptions

RT-environments always impose restrictions for the interrupt handler.

The handler then either …

- … deals with the situation itself (employing its limited capabilities)

... or …

- … initiates a general change in the control flow (involving other parts of the system).

Formulate restrictions with respect to the interruptible code!

e.g. Ada: ‘asynchronous transfer of control’, Real-time Java: ‘Interrupted exceptions’

(exception handling concepts and atomic actions will be introduced first, then the discussion about asynchronous transfer of control methods is continued)
Asynchronism

Exceptions

Wish list for exception handling in real-time programming languages:

1. Exception facilities should **not obscure the understanding** of the normal, exception-free control flow.

2. Exception facilities should produce **no or minimal run-time overhead**, until an exception actually occurs.

3. Exceptions should produce **predictable run-time overhead** when an exception occurs.

4. Exceptions indicated by the run-time environment and by the program itself should be **treated uniformly**.

5. The exception mechanism should be applicable to **asynchronous** and **synchronous exceptions** (might be hard to achieve with respect to wish 3).

6. The exception mechanism should allow for **appropriate recoveries** (supply sufficient information and appropriate re-entry possibilities).
Asynchronism

Exceptions

Historic exception handling methods:

- Use an ‘unusual return value’ and a global variable convention: ‘C’:

```c
if (function_call (parameters) < NORMAL.Return.VALUE) {
    if (errno == SOME.KNOWN.ERROR) {
        /* react to the known error condition */
    } else {
        /* try to improvise something */
    }
} else {
    /* normal control flow */
}
```
Asynchronism

Exceptions

Historic exception handling methods:

- Use an ‘unusual return value’ and a global variable convention: ‘C’:

```c
if (function_call (parameters) < NORMAL_RETURN_VALUE) {
    if (errno == SOMEKNOWN_ERROR) {
        /* react to the known error condition */
    } else {
        /* try to improvise something */
    }
} else {
    /* normal control flow */
}
```

- *inflexible* (exceptions from the environment cannot be detected)

- *error-prone* (lots of chances to forget checking or to use the wrong constants)

- *obstructive* (all fragments are commingled)
Asynchronism

Exceptions

Assembler level exception handling methods:

- Provide a jump table and manipulate the return address on the stack:

Caller:

```assembly
jsr pc, PRINT_CHAR
jmp IO_ERROR
jmp DEVICE_NOT_ENABLED
* normal processing
```

Subroutine:

```assembly
% indicate an exception:
%    increment the return address on the stack
%    by the exception number
%    to employ the caller-provided exception handling
% indicate normal operation:
%    increment the return address on the stack
%    by the max. exception number + 1
```
Emulating exception handling methods (in older languages):

- Unrecoverable exceptions: provide a *jump label*.
- Recoverable exception: provide a *procedure variable*.

Historic, since all current real-time suitable languages provide some means of dedicated exception handling.
### Exception indication

Four cases of modern exception indication:

<table>
<thead>
<tr>
<th>raised:</th>
<th>from:</th>
<th>run-time environment</th>
<th>application</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronously</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>asynchronously</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
Asynchronism

Exception indication

Ada95:

<table>
<thead>
<tr>
<th>raised:</th>
<th>from:</th>
<th>run-time environment</th>
<th>application</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronously</td>
<td>exceptions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asynchronously</td>
<td>interrupt/signal handler</td>
<td>asynchronous transfer of control</td>
<td></td>
</tr>
</tbody>
</table>
## Asynchronism

### Exception indication

Real-time Java:

<table>
<thead>
<tr>
<th>raised:</th>
<th>from:</th>
<th>run-time environment</th>
<th>application</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronously</td>
<td></td>
<td></td>
<td>exceptions</td>
</tr>
<tr>
<td>asynchronously</td>
<td></td>
<td></td>
<td>asynchronous exceptions</td>
</tr>
</tbody>
</table>
## Asynchronism

### Exception indication

**POSIX:**

<table>
<thead>
<tr>
<th>raised:</th>
<th>from:</th>
<th>run-time environment</th>
<th>application</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronously</td>
<td>N/A</td>
<td>(global variables)</td>
<td></td>
</tr>
<tr>
<td>asynchronously</td>
<td>(signals)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Asynchronism

Exception granularity

at block level:

Ada95:

```ada
begin
  -- do something dangerous
  exception
  when E: Constraint_Error => Deal with it;
end;
```

or in RT-Java

```java
try {
  // do something dangerous
}
catch (ExceptionType e) {
  // handle exception e
}
```

deployment

all exceptions need to be (or are) declared!
Asynchronism

Exception granularity

at block level:

Ada95:

```ada
begin
  -- do something dangerous
  exception
    when E: others => Deal with it;
end;
```

or in RT-Java

```java
try {
  // do something dangerous
}
catch (Exception e) {
  // handle exception e
}
```

handlers can catch all!
buts don’t need to catch any
Asynchronism

Exception granularity

large blocks:

```
declare
subtype Temperature is Integer range 0 .. 100;
subtype Pressure is Integer range 0 .. 50;
subtype Flow is Integer range 0 .. 200;

begin
  -- read temperature sensor and calculate its value
  -- read pressure sensor and calculate its value
  -- read flow sensor and calculate its value
  -- adjust temperature, pressure and flow
  -- according to requirements

exception
  -- handler for Constraint_Error
end;
```
Asynchronism

Exception granularity

small blocks:

declarer
...
begin
begin
-- read temperature sensor and calculate its value
exception -- handler for Constraint_Error for temperature
end;

begin
-- read pressure sensor and calculate its value
exception -- handler for Constraint_Error for pressure
end;
...

exception -- handler for other possible exceptions
end;
Asynchronism

Exception granularity

small blocks:

```
declare
...
begin
  begin
    -- read temperature sensor and calculate its value
    exception -- handler for Constraint_Error for temperature
    end;

  begin
    -- read pressure sensor
    exception -- handler for Constraint_Error for pressure
    end;

  ...

  exception -- handler for other possible exceptions
end;
```
Asynchronism

Exception granularity
statement level:

☞ e.g. supported in CHILL

A : temperature;
B, C : integer;

begin
  A := B + C on
      (overflow ): ...;
      (rangefail): ...;
      else ...;
    end;
  ...
end;

☞ exceptions in CHILL can also be handled at block, procedure, or process level

All exceptions in CHILL need to be handled

Exceptions are not propagated

Handlers are determined at compile-time

☞ exceptions in CHILL can also be handled at block, procedure, or process level
Asynchronism

Exception granularity

attach parameters:

Instead of catching exceptions after each statement:

 Exceptions can have parameters with information about its source:

 Real-time Java:
 Exception can carry any number of user-defined parameters

 Ada95:
 The environment automatically attaches additional information to exceptions, which are indicating the position of the exception-occurrence and other observed conditions (implementation dependent, inflexible, useful for debugging).
Asynchronism

Exception propagation

1. All procedures and functions declare every potentially raised exceptions (requested in CHILL, requested in Real-time Java for user-defined exceptions):

☞ If an appropriate exception handler can not be determined at compile-time:
   
   Either …
   
   • … treat it as a programmer error and stop compilation (☞ CHILL)
   
   or …
   
   • … propagate the exception at run-time outside its static scope (☞ Real-time Java)

2. Exceptions are declared for whole modules — not specifically for methods (Ada95):

☞ The handler is determined at run-time in any case (either by propagation or in the static scope).
Asynchronism

Exception propagation
(tasks)

If the handler can neither be found by propagation or in the static context of a task:

☞ Stop the specific task — exceptions are not propagated to the parent-process (Ada95).

or

☞ Propagate the exception to the parent-task (and potentially stall the whole system).

Common ‘safety-line’ for each process:

☞ Use a ‘catch all’-handler (possible in Ada95 and Real-time Java) at the highest level.

   (often last emergency level for this process ☞ common reaction: PANIC!)
Asynchronism

Exception handling: resumption or termination?

Resumption-model (offered in: Pearl, Mesa):

1. Find an exception handler.
2. Execute the exception handler (and potentially raise another exception recursion).
3. After completion of the exception handler (and possibly other handlers):
   return to the invoker and try resume processing as if nothing happened.

Feature:

- In case of an asynchronous exception, there is little impact on the current control-flow.

Problems:

- some errors cannot be ‘repaired’ (especially all timing related errors).
- exceptions can be raised in the middle of evaluations, which will be hard to restore.
Asynchronism

Exception handling: resumption or termination?

‘Block resumption’-model (offered in: Eiffel):

☞ Re-execute the complete code-block after exception handling

Feature:

• Intended to keep the formulated contract for this method.

Problems:

• Local variables must not be re-initialized (otherwise the exception will probably occur again)
• The code needs to be aware of all possible combinations of half-evaluated processing states.
• Trying the same method again (and again) is usually not the suitable way for real-time systems.
Asynchronism

Exception handling: resumption or termination?

Termination-model (only model in Ada95, Real-time Java; offered in: Pearl, Mesa):

- The control is *not* returned to the point of invocation.
- Instead the block / function / procedure is assumed to terminated in an exceptional state, and the control is returned to the calling or enclosing scope of the activated exception handler.
- If the calling block wish to re-try the same operation, it need to start over at the visible entry-points and with re-initialized local variables.

Feature:

- The method of choice, if exceptions imply that the operation (statement, block, process) was *not successful* and *something else* need to be done now in real-time systems.

Problem:

- There is no way to continue, in case that the exception could be identified as of minor impact.
Asynchronism

Exception handling: resumption or termination?

Hybrid-model (offered in: Mesa):

- The exception handler can decide at run-time whether to terminate or to resume.
Asynchronism

Cleaning up before exception-handling:

Assuming a block is holding a number of resources, and occurring exceptions need to be handled at the caller level:

```plaintext
procedure Allocate (Number : Devices) is
begin
  -- request each device be allocated in turn
  -- noting which requests are granted
exception
  when others =>
    -- e.g. deallocate those devices allocated
    raise; -- re-raise the exception
end Allocate;
☞ helpful to keep a consistent system-state and to avoid dead-locks (all-or-nothing allocation).

... in Real-time Java: the ‘finally’ clause takes care of block consistent finalization.
```
Asynchronism

Issues when handling exceptions in Ada95:

☞ Exceptions are declared at package level,
   i.e. it is unclear, which functions may raise which exceptions!

☞ Exceptions may be propagated outside the scope of their declaration,
   i.e. only ‘when others’ can handle them (might also be further propagated back in scope again).

☞ Parameter passing limited to one string.

☞ Exception in task bodies are never propagated to the parent task
   i.e. if there could not be any handler identified in the task, the task will ‘die silently’.

☞ Exception in task declarations are always propagated to the parent task.

☞ Exceptions in task rendezvous, which are not handled in the accept statement,
   are propagated to both involved tasks.

☞ Traps, which need to be taken care of!

most expensive not caught exception up to now: half a billion dollars (maiden crash of Ariane 5, ‘96)
Asynchronism

Exception handling in Real-time Java

- Throwable
- Errors
  - Linkage
  - VM
  - RunTime
- Exceptions
  - User def.

- checked: need to be declared per method (runtime exceptions can occur undeclared)
- unchecked: (errors are unrecoverable)
Exception handling in Real-time Java

Exceptions are objects in Real-time Java:

☞ Exceptions have hierarchical relations
☞ Exceptions handlers can catch
  • one individual exception
  • all exceptions out of a finite list of exceptions
  • all exceptions of a certain class
  • all exceptions

☞ The kinds of exceptions which are handled at a certain point can be described precisely, completely, and safely.
(exceptions are not part of the class-hierarchy in Ada95 or Eiffel)
Asynchronism

Exception handling in ‘C’ / POSIX

☞ there is no exception handling in ‘C’ / POSIX

possible work arounds by using POSIX long jumps or signals.
☞ see also macro-assembler or ‘old language’ exception handling methods.
### Asynchronism

#### Exception handling: compare sheet

<table>
<thead>
<tr>
<th></th>
<th>Terminating?</th>
<th>Handler</th>
<th>Decl. per:</th>
<th>Decl. as:</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-time Java</td>
<td>termination</td>
<td>dynamical / propagating</td>
<td>method</td>
<td>classes</td>
<td>‘try-block’</td>
</tr>
<tr>
<td>Ada95</td>
<td>termination</td>
<td>dynamical / propagating</td>
<td>package</td>
<td>static names</td>
<td>any block</td>
</tr>
<tr>
<td>CHILL</td>
<td>termination</td>
<td>static</td>
<td>via handler</td>
<td>static handler</td>
<td>statement / block / task</td>
</tr>
<tr>
<td>Mesa</td>
<td>resumption or termination</td>
<td>dynamical / propagating</td>
<td>procedure</td>
<td>static procedures</td>
<td>block</td>
</tr>
<tr>
<td>Pearl</td>
<td>resumption or termination</td>
<td>static</td>
<td>process</td>
<td>static names</td>
<td>task</td>
</tr>
<tr>
<td>Eiffel</td>
<td>class-retry</td>
<td>dynamical / propagating</td>
<td>(contract violation)</td>
<td>-</td>
<td>‘do-rescue-block’</td>
</tr>
<tr>
<td>‘C’ / POSIX</td>
<td></td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Asynchronism

Atomic actions

Atomic actions: definitions:

An action is atomic if the processes performing it …

• ... are not aware of the existence of any other active process, and no other active process is aware of the activity of the processes during the time the processes are performing the action.

• ... do not communicate with other processes while the action is being performed.

• ... cannot detect any outside state change and do not reveal their own state changes until the action is complete.

☞ ... can be considered to be indivisible and instantaneous.
Atomic actions: implications:

An atomic action ...

- ... is either performed fully, or not at all.
- ... is declared as failed, if any part of the action fails.

Thus all parts of an atomic action need to be prepared:

☞ to be interrupted (due to the failure of one of them)
☞ and to reset to their initial state at any time
  (‘no effect’ is visible to the outside)
Asynchronism

Atomic actions

Time-lines:
Asynchronism

Atomic actions

Nested atomic actions:

Action actions can be nested, iff ...

- ... all processes involved in the nested atomic actions are a true subset of the processes involved in the enclosing atomic action.
Asynchronism

Nested atomic actions

Time-lines:
Asynchronism

Atomic actions – Requirements for real-time environments:

• Well-defined boundaries
  • A \textit{start}, \textit{end} and a \textit{side} boundary:
    ☞ Define clear entry and exit points for all processes involved in the atomic action.
    Processes can enter at different time, but need are released from the action at once.
    ☞ Separate the involved processes from the rest of the system (‘side boundary’).

• Indivisibility (Isolation)
  • \textit{Prohibit or restrict communications} to outside processes and resources.
  • Employing results from one atomic action in another one ☞ implies \textit{strict serialisation}.

• Nesting
  • Atomic actions may be nested, if they form a \textit{true enclosing relation}.

• Concurrency
  • Independent atomic actions may be executed in \textit{any order} and concurrently.
Asynchronism

Atomic actions

Failure in one action part:
Asynchronism

Atomic actions

Failure in one action part: 'clean up' (restore the initial states)
Asynchronism

Nested atomic actions

Failure in one part of a nested atomic action:
Asynchronism

Atomic actions

Failure in one part of a nested atomic action (iterative failure propagation):
Asynchronism

Atomic actions

Failure in one part of a nested atomic action (iterative failure propagation):
Asynchronism

Nested atomic actions

Failure in one part of a nested atomic action:
Asynchronism

Atomic actions

Failure in one part of a nested atomic action (immediate failure propagation):
Asynchronism

Atomic actions

Failure in an enclosing atomic action:
Asynchronism

Atomic actions

Failure in an enclosing atomic action (no communication with inner action):
Asynchronism

Atomic actions

Failure in an enclosing atomic action (no communication with inner action):
Asynchronism

Atomic actions

Failure in an enclosing atomic action (no communication with inner action):
Asynchronism

Atomic actions

Failure in an enclosing atomic action:
Asynchronism

Atomic actions

Failure in an enclosing atomic action (revoking activation condition for inner action):
Asynchronism

Atomic actions

No mainstream language is supplying such a construct.

```
action A with (P2, P3, ...) do

  ...
  -- communication is restricted to P2, P3, ...
  -- exceptions and timing constraints violations are propagated
  -- to all involved processes
  ...

exception
  when exception_a => ... -- recover locally
  when exception_b => ... -- recover locally
  ...
  when others       => raise atomic_action_failure; -- and fail the action

end A;
```
Asynchronism

Atomic actions

Implementing atomic actions by creating dedicated tasks:
Asynchronism

Atomic actions in Ada95:

```ada
with Atomic_Action_Types; use Atomic_Action_Types;
generic
  Actions : in Action_Parts;
package Generic_Atomic_Action is
  procedure Perform;
  Failure_State,
  Time_Out State,
  Late_Activation_State : exception;
end Generic_Atomic_Action;
```

☞ Scope mechanism is employed to limit communication possibilities of atomic action parts.
☞ The `perform`-call is atomic for the caller and invisible for others.
☞ Failures in one part are automatically propagated in the whole atomic action.
Asynchronism

Atomic actions in Ada95:

with:

```ada
with

type Action_Part_Time_Scope is record
  Start_Delay_Min : Time_Span := Time_Span_Zero;
  Start_Delay_Max : Time_Span := Time_Span_Last;
  Max_Elapse      : Time_Span := Time_Span_Last;
  Deadline        : Time      := Time_Last;
end record;

type Action_Part_Proc is access procedure;

type Action_Part_Pros is record
  Action, Cleanup : Action_Part_Proc;
  Scope           : Action_Part_Time_Scope;
end record;

type Action_Parts is array (Positive range <>) of Action_Part_Pros;
```

☞ The Cleanup procedure is meant to restore the initial state! (‘undoing’ all effects of Action)
Asynchronism

Atomic actions in Ada95:

```ada
Actions : Action_Parts (Tasks_Index) :=
(1 => ('Action' => Action_Task_1'Access,
               'Cleanup' => Cleanup_Task_1'Access,
               'Scope' => (Start_Delay_Min => Milliseconds (33),
                            Start_Delay_Max => Milliseconds (133),
                            Max_Elapse => Time_Span_Last,
                            Deadline => Time_Last)),
   2 => ...

package Atomic_Action is new Generic_Atomic_Action (Actions);

procedure Perform is
begin
   Atomic_Action.Perform;
   exception
      when ... end Perform;
```
Asynchronism

Atomic actions in Ada95:

```
task body Action_Task is
  ... begin
    Monitor.Check_In (Task_Id);
    select
      Monitor.Failed (Task_Id);
      Actions (Task_Id).Cleanup.all;
      Atomic_Action.Monitor.Check_Out (Task_Id);
    then abort
      begin
        select
          delay To_Duration (Actions (Task_Id).Scope.Start_Delay_Max);
          raise Late_Activation_State;
        then abort
          delay To_Duration (Actions (Task_Id).Scope.Start_Delay_Min);
        end select;
```
Asynchronism

Atomic actions in Ada95:

... 

select 
    delay until Real_Deadline; -- based on Max_Elapse & Deadline
    raise Time_Out_State;

then abort 
    Actions (Task_Id).Action.all;
end select;

Atomic_Action.Monitor.Check_Out (Task_Id);

exception 
    when Time_Out_State => Monitor.Fail (Time_Out);
    when Late_Activation_State => Monitor.Fail (Late_Activation);
    when others => Monitor.Fail (Other_Exception);
end;
end Action_Task;
Asynchronism

Atomic actions in Ada95:

```ada
protected Monitor is
  entry Check_In (Task_Id : in Task_Ids);
  entry Fail   (Condition : in Atomic.Condition);
  entry Failed (Task_Id : in Task_Ids);
    -- blocking until Fail is called
  entry Check_Out (Task_Id : in Task_Ids);
    -- blocking until all parts are completed or all are cleaned up
  entry Action_Result (Condition : out Atomic.Condition);

private
  Check_List      : Task_List        := Check_List_Out;
  State           : Atomic_State     := Checking_In;
  Final_Condition : Atomic.Condition := Succeeded;
end Monitor;
```

☞ Check the course page for the complete sources
Asynchronism

Atomic actions

Mechanism can be extended to allow any task to dedicate itself to one part of an atomic action as well as to allow for nested atomic actions: Laboratories!
Asynchronism

Atomic actions:
Backward error recovery in real-time environments

Since once some parts of the action might have failed:
☞ some action-parts might re-execute the same method again,
☞ or execute alternative methods, even if the original method was locally valid.

In a real-time environment, failed atomic actions are often identical with some kind of a disaster:

• Tracking back and re-trying the same atomic action
  with modified parameters or methods is rarely useful, considering timing constraints.
☞ A ‘mode change’ and a complete different set of (atomic) actions (and goals!)
  might be more useful in many cases.
Atomic actions:
Forward error recovery in real-time environments

Backtracking is often **hardly possible** in real-time systems!

- … and even if it is, it would be **rarely useful** in a real-time context (see above).

Forward error recovery is more common in real-time systems.

(more on forward error recovery in chapter 10)
Asynchronism

Asynchronous Transfer of Control vs. Interrupts

From interrupts (sub-process level) employed in …

• … communication with slow / asynchronous / sporadic devices
• … sampling / control loops
• … closely coupled reflective systems

… to asynchronous transfer of control (process level):

• Error recovery — supporting atomic actions and forward recovery
• Mode changes — sudden changes from ‘normal’ operations to emergency measures
• Partial / imprecise computations — whenever timeliness is more important than precision
• Operator intervention — User triggered mode changes
Asynchronism

Real-time Java

supplies three basic features:

1. Binding of **handlers** to internal and external **asynchronous events**.
   1-a Specifying asynchronous events
   1-b Specifying adequate handlers

2. **Asynchronous transfer of control**.

3. Asynchronous **termination** of threads.
Asynchronism

Real-time Java: Asynchronous events

Whenever an instance of AsyncEvent occurs:

- all .run() methods of all instances of AsyncEventHandler, which are bound to this AsyncEvent are scheduled for execution.
- multiple AsyncEvents may have different impacts on the scheduling.
- an event counter (FireCount) is supplied.
- AsyncEvents and AsyncEventHandler s may be created and used by any program logic.
- More than one handler can be attached to one event.
- More than one event can be attached to one handler.

☞ Flexible, but handled as a schedulable object
Asynchronism

Real-time Java: Asynchronous events

```java
public class AsyncEvent {
    public AsyncEvent ();
    public synchronized void addHandler (AsyncEventHandler handler);
    public synchronized void removeHandler (AsyncEventHandler handler);
    public void setHandler (AsyncEventHandler handler);
    public void bindTo (java.lang.String happening);
    public ReleaseParameters createReleaseParameters ();
    public void fire ();
    ...
}
```
Asynchronism

Real-time Java: Asynchronous events

```java
public abstract class AsyncEventHandler implements Schedulable {
    public AsyncEventHandler (SchedulingParameters scheduling,
                              ReleaseParameters release,
                              MemoryParameters memory,
                              MemoryArea area,
                              ProcessingGroupParameters group);

    public void addToFeasibility ();
    public void removeFromFeasibility ();

    protected int getAndClearPendingFireCount ();

    public abstract void handleAsyncEvent (); -- called by run while FireCount ≠ 0

    public final void run ();

    ...
}
```
Asynchronism

Real-time Java: Asynchronous events

public class BoundAsyncEventHandler extends AsyncEventHandler {
    public BoundAsyncEventHandler (); // inherit modes from the current thread
    public BoundAsyncEventHandler (SchedulingParameters scheduling,
                                   ReleaseParameters release,
                                   MemoryParameters memory,
                                   MemoryArea area,
                                   GroupParameters group,
                                   boolean nonheap,
                                   java.langRunnable logic);
}

bind the current thread to an AsyncEventHandler
(otherwise the AsyncEventHandler will execute in a thread of its own).
Asynchronism

Java: Interrupting exception

While an AsyncEvent just schedules a handler for execution, other event classes are needed to alter the control flow more directly:

Standard Java:

the InterruptedException indicates the wish to interrupt a thread, which itself need to poll the isInterrupted method to find out, whether it is supposed to be interrupted.
If the thread is currently executing and ignoring the flag:
☞ there is no effect on the actual control flow!

If the thread, which is to be interrupted is currently blocking:
☞ it is activated and receives an InterruptedException.

☞ too weak to be employed in an asynchronous transfer of control!
Asynchronism

Real-time Java: Asynchronously interrupting exception

While an AsyncEvent just schedules a handler for execution, other event classes are needed to alter the control flow more directly:

Real-time Java:

the AsynchronouslyInterruptedException might intercept a control flow directly, while:

- *The code regions*, which are interruptible *need to be indicated*.
- Synchronized blocks, task creations and finalizations are not interruptible
- The response time of a thread must be within ‘a bounded number of bytecodes’ (which itself need to be documented).
- If the interruptible thread is *currently blocked* (in a java.io.* operation) then *the thread is either released or kept in the block* (definition of a reasonable blocking time or situation is implementation dependent).
Asynchronism

Real-time Java: Asynchronously interrupting exception

Cases of operations if a `AsynchronouslyInterruptedException` (AIE) occurs:

1. If control is in a synchronized section (within an interruptible section):
   ☞ the AIE is put into a pending state

2. If control is in an interruptible section:
   ☞ the control is transferred to the nearest dynamically enclosing catch clause, which is in a non-interruptible section.

3. If control is in `wait`, `sleep`, or `join`:
   ☞ the thread is activated and 1. or 2. is followed.

4. If control is in a non-interruptible section:
   ☞ nothing happens until an interruptible section is reached

5. If an interruptible section is reached while propagating an exception:
   ☞ the original exception is discarded (!) and replaced by the AIE.
Real-time Java: Asynchronously interrupting exception

```java
public class AsynchronouslyInterruptedException extends java.lang.InterruptedException {
    { ...
        public synchronized void disable();
        public synchronized boolean enable();
        public synchronized boolean fire();
        public boolean doInterruptible (Interruptible logic);
        public boolean happened (boolean propagate);
        public static AsynchronouslyInterruptedException getGeneric();
            // returns the AsynchronouslyInterruptedException which
            // is generated when RealtimeThread.interrupt() is invoked
        public void propagate();
    }
```
Asynchronism

Real-time Java: Asynchronously interrupting exception

```java
import NonInterruptibleServices.*;
public class InterruptibleService {
    public AIE stopNow = AIE.getGeneric();
    public boolean Service() throws AIE {
        try {
            // code interdispersed with calls to NonInterruptibleServices
        } catch (AIE AI) {
            if (stopNow.happened (true)) {
                // handle the ATC
            }
        }
    }
}
```

interruptible code section

handle the `stopNow` propagate anything else
Real-time Java: Asynchronously interrupting exception

```java
import NonInterruptibleServices.*;
public class InterruptibleService {
    public AIE stopNow = AIE.getGeneric();
    public boolean Service() throws AIE {
        try {
            // code interdispersed with calls to NonInterruptibleServices
        }
        catch AIE AI {
            if (stopNow.happened (false)) {
                // handle the ATC
            } else { // cleanup
                AI.propagate
            }
        }
    }
}
```

interruptible code section

handle the `stopNow`
otherwise: clean up and propagate
Real-time Java: Asynchronous exception propagation

While asynchronously interrupting event handling is part of the standard exception handling mechanism, there are nevertheless differences in exception propagation:

- A standard exception is not propagated by default when caught by any ‘catch’ statement (an explicit re-raising of the exception is necessary to pass it on).
- A `AsynchronouslyInterruptedException` is propagated further, even if caught (an explicit propagation stop is necessary to avoid its further propagation).

☞ realized in the `AIE.happened` method

Rationale:

- local catch-all clauses might not be aware of a potential asynchronous interrupting exception.

☞ practical compromise, but destroying some integrity of the Java-exception concept.
Real-time Java: Interruptible interface

```java
public interface Interruptible {
    public void interruptAction
        (AsynchronouslyInterruptedException exception);
    public void run
        (AsynchronouslyInterruptedException exception)
        throws AsynchronouslyInterruptedException;
}
```

- An object may declare an interruptible method explicitly by implementing the above interface.
  - Only purpose: pass this implementation to a `doInterruptible` method of an AIE class.
  - The `AIE.doInterruptible` can only be called by one thread at a time!
Asynchronism

Real-time Java: Timeout on actions

```java
public class Timed extends AsynchronouslyInterruptedException
    implements java.io.Serializable
{
    public Timed (HighResolutionTime time) throws IllegalArgumentException;
    public boolean doInterruptible (Interruptible logic);
    public void resetTime (HighResolutionTime time);
}
```

- The timer is started sometime between the invocation of the `doInterruptible` itself and the `run` method of the interruptible interface.
- A generic `interrupt()` is thrown at the expiration of the timer.
- `time` can be absolute or relative.
Asynchronism

Ada95

provides:

- Exception handling (synchronous only).
- Asynchronous transfer of control
- Task aborts
- Interrupt handling (close to the hardware).

☞ a set of different methods to handle different kinds of events!
package Ada.Interrupts is

  type Interrupt_ID          is implementation-defined;
  type Parameterless_Handler is access protected procedure;

  function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
  function Is_Attached (Interrupt : Interrupt_ID) return Boolean;

  function Current_Handler (Interrupt : Interrupt_ID)
    return Parameterless_Handler;

  procedure Attach_Handler   (New_Handler : in  Parameterless_Handler;
                               Interrupt   : in  Interrupt_ID);

  procedure Exchange_Handler (Old_Handler : out Parameterless_Handler;
                               New_Handler : in  Parameterless_Handler;
                               Interrupt   : in  Interrupt_ID);

  procedure Detach_Handler   (Interrupt   : in  Interrupt_ID);

  function Reference (Interrupt : Interrupt_ID) return System.Address;

end Ada.Interrupts;
Asynchronism

Ada95: Interrupt handlers

package Ada.Interrupts is

    type Interrupt_ID is implementation-defined;
    type Parameterless_Handler is access protected procedure;

    function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
    function Is_Attached (Interrupt : Interrupt_ID) return Boolean;
    function Current_Handler (Interrupt : Interrupt_ID) return Parameterless_Handler;

    procedure Attach_Handler (New_Handler : in Parameterless_Handler; Interrupt : in Interrupt_ID);
    procedure Detach_Handler (Interrupt : in Interrupt_ID);

    function Reference (Interrupt : Interrupt_ID) return System.Address;

end Ada.Interrupts;

Protected procedures need to qualify as an interrupt handler:

1. use pragma Interrupt_Handler
2. let the compiler evaluate the suitability of the routine as an interrupt handler.
Asynchronism

Ada95: Interrupt handlers

package Ada.Interrupts is
  type Interrupt_ID is implementation-defined;
  type Parameterless_Handler is access protected procedure;
  function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
  function Is_Attached (Interrupt : Interrupt_ID) return Boolean;
  function Current_Handler (Interrupt : Interrupt_ID) return Parameterless_Handler;
  procedure Attach_Handler (New_Handler : in  Parameterless_Handler; Interrupt : in  Interrupt_ID);
  procedure Detach_Handler (Interrupt : in  Interrupt_ID);
  function Reference (Interrupt : Interrupt_ID) return System.Address;
end Ada.Interrupts;

Protected procedures can also be attached statically to an interrupt:

use pragma Interrupt_Handler_Attach
Asynchronism

Ada95: Interrupt handlers

package Ada.Interrupts is
  type Interrupt_ID is implementation-defined;
  type Parameterless_Handler is access protected procedure;
  function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
  function Is_Attached (Interrupt : Interrupt_ID) return Boolean;
  function Current_Handler (Interrupt : Interrupt_ID) return Parameterless_Handler;
  procedure Attach_Handler (New_Handler : in Parameterless_Handler; Interrupt : in Interrupt_ID);
  procedure Detach_Handler (Interrupt : in Interrupt_ID);
  function Reference (Interrupt : Interrupt_ID) return System.Address;
end Ada.Interrupts;

The mechanism to invoke an interrupt handler may be different from calling a protected procedure from a task.

*Implementation advice*: Whenever possible, the implementation should allow interrupt handlers to be called directly by the hardware.
Asynchronism

Ada95: Interrupt handlers

package Ada.Interrupts is
  type Interrupt_ID is implementation-defined;
  type Parameterless_Handler is access protected procedure;
  function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
  function Is_Attached (Interrupt : Interrupt_ID) return Boolean;
  function Current_Handler (Interrupt : Interrupt_ID) return Parameterless_Handler;
  procedure Attach_Handler (New_Handler : in Parameterless_Handler; Interrupt : in Interrupt_ID);
  procedure Detach_Handler (Interrupt : in Interrupt_ID);
  function Reference (Interrupt : Interrupt_ID) return System.Address;
end Ada.Interrupts;

Metrics: The implementation shall document the worst case overhead for an interrupt handler invocation (in clock cycles).
package Ada.Interrupts is
  type Interrupt_ID is implementation-defined;
  type Parameterless_Handler is access protected procedure;
  function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
  function Is_Attached (Interrupt : Interrupt_ID) return Boolean;
  function Current_Handler (Interrupt : Interrupt_ID) return Parameterless_Handler;
  procedure Attach_Handler (New_Handler : in Parameterless_Handler; Interrupt : in Interrupt_ID);
  procedure Detach_Handler (Interrupt : in Interrupt_ID);
  function Reference (Interrupt : Interrupt_ID) return System.Address;
end Ada.Interrupts;

Direct access to the invocation address:
May be used to connect task-entries to interrupts
☞ risky! — use with special care.
Asynchronism

Ada95: Asynchronous Transfer of Control

asynchronous_select ::= select
    triggering_alternative
    then abort
    abortable_part
end select;

triggering_alternative ::= triggering_statement [sequence_of_statement]

triggering_statement ::= entry_call_statement | delay_statement

abortable_part ::= sequence_of_statements

☞ cannot contain an accept statement.
Asynchronism

Ada95: Asynchronous Transfer of Control

Execute the trigger (entry.call or delay), then:

1. If the trigger is going through and can be completed: the optional statements following the trigger are executed and the select statement is completed (the abortable part is never started).

2. If the trigger is blocked or requeued to a blocked entry: the statements in the abortable part are executed:

   2-a If the abortable part completes before the trigger is completed, an attempt is made to revoke the triggering statement. The select statement is completed after the cancelled or completed triggering statement.

   2-b If the trigger is completed before the abortable part is completed, the abortable part is stopped, the optional statements following the trigger are executed and the select statement is completed.

   ```
   select
       <entry-call | delay>
       [ ... statements ... ]
   then abort
       ... statements ...
   end select;
   ```
Asynchronism

Ada95: Asynchronous Transfer of Control

Exception handling:

Both parts of a select-then-abort statement can raise exceptions, but ...

☞ ... in case of an interruption of the abortable part, the exceptions from the abortable part are lost!

```ada
select
  <entry-call | delay>
  [ ... statements ... ]
then abort
  ... statements ...
end select;
```
Asynchronism

Ada95: Asynchronous Transfer of Control

task body A is
  T : Time;
  D : Duration;
begin
  ...
  select
    delay until T;
  then abort
    delay D;
  end select;
end A;

☞ are these equivalent?
Asynchronism

Ada95: Asynchronous Transfer of Control

task body A is
  T : Time;
begin
  select
    delay until T;
    ST;
  then abort
    Server.Entry1;
    SR;
  end select;
end A;

task body B is
  T : Time;
begin
  select
    Server.Entry1;
    SR;
  then abort
    delay until T;
    ST;
  end select;
end B;

task body C is
  T : Time;
begin
  select
    Server.Entry1;
    SR;
  or
    delay until T;
    ST;
  end select;
end C;

... are these equivalent?.
Asynchronism

Ada95: Asynchronous Transfer of Control

task body A is
  T : Time;
begin
  select
    delay until T;
    ST;
  then abort
    Server.Entry1;
    SR;
  end select;
end A;

task body B is
  T : Time;
begin
  select
    Server.Entry1;
    SR;
  then abort
    delay until T;
    ST;
  end select;
end B;

task body C is
  T : Time;
begin
  select
    Server.Entry1;
    SR;
or
    delay until T;
    ST;
  end select;
end C;

... if rendezvous starts and completes before timeout.
Asynchronism

Abda95: Asynchronous Transfer of Control

task body A is
T : Time;
begin
    select
delay until T;
    ST;
    then abort
    Server.Entry1;
    SR;
end select;
end A;

... if rendezvous starts and completes before timeout.

task body B is
T : Time;
begin
    select
    Server.Entry1;
    SR;
    then abort
    delay until T;
    ST;
end select;
end B;

... if rendezvous starts and completes before timeout.

task body C is
T : Time;
begin
    select
    Server.Entry1;
    SR;
    or
    delay until T;
    ST;
end select;
end C;
Asynchronism

Ada95: Asynchronous Transfer of Control

task body A is
T : Time;
begin
  select
    delay until T;
    ST;
  then abort
    Server.Entry1;
    SR;
  end select;
end A;

task body B is
T : Time;
begin
  select
    Server.Entry1;
    SR;
  then abort
    delay until T;
    ST;
  end select;
end B;

task body C is
T : Time;
begin
  select
    Server.Entry1;
    SR;
  or
    delay until T;
    ST;
  end select;
end C;

... if rendezvous starts before but finishes after timeout.
Asynchronism

Ada95: Asynchronous Transfer of Control

task body A is
  T : Time;
begin
  select
    delay until T;
    ST;
  then abort
    Server.Entry1;
    SR;
  end select;
end A;

task body B is
  T : Time;
begin
  select
    Server.Entry1;
    SR;
    delay until T;
    ST;
  then abort
    Server.Entry1;
    SR;
  end select;
end B;

task body C is
  T : Time;
begin
  select
    Server.Entry1;
    SR;
    or
    delay until T;
    ST;
    ST;
  end select;
end C;

... if rendezvous starts before but finishes after timeout.
Asynchronism

Ada95: Asynchronous Transfer of Control

task body A is

T : Time;

begin

select

delay until T;

ST;

then abort

Server.Entry1;

SR;

end select;

end A;

... timeout occurs before the rendezvous starts.

task body B is

T : Time;

begin

select

Server.Entry1;

SR;

then abort

delay until T;

ST;

end select;

end B;

... timeout occurs before the rendezvous starts.

task body C is

T : Time;

begin

select

Server.Entry1;

SR;

or

delay until T;

ST;

end select;

end C;
Asynchronism

Ada95: Asynchronous Transfer of Control

```ada
task body A is
    T : Time;
begin
    select
        delay until T;
        ST;
    then abort
        Server.Entry1;
        SR;
    end select;
end A;

task body B is
    T : Time;
begin
    select
        Server.Entry1;
        SR;
    then abort
        delay until T;
        ST;
    end select;
end B;

task body C is
    T : Time;
begin
    select
        Server.Entry1;
        SR;
    or
        delay until T;
        ST;
    end select;
end C;
```

... timeout occurs before the rendezvous starts.
Asynchronism

Ada95: Asynchronous Transfer of Control

task body A is
T : Time;
begin
select
  delay until T;
  ST;
then abort
  Server.Entry1;
  SR;
end select;
end A;

... timeout occurs before the rendezvous starts.

task body B is
T : Time;
begin
select
  Server.Entry1;
  SR;
then abort
  delay until T;
  ST;
end select;
end B;

... timeout occurs before the rendezvous starts.

task body C is
T : Time;
begin
select
  Server.Entry1;
  SR;
or
  delay until T;
  ST;
end select;
end C;
Asynchronism

Asynchronism in Ada95 and Real-time Java:

(Common features)

- ATC-enabled regions must be declared.
- Some regions are always deferred from asynchronous transfer of control (task/thread communication / finalization).
- Exceptions from the run-time environment as well as user-defined exceptions are supported.
- Asynchronous events may be triggered by the environment as well as from a task.
Asynchronism

Asynchronism in Ada95 and Real-time Java:

(Differences)

- Mechanisms:
  - In Real-time Java asynchronism is embedded into the synchronous exception scheme.
  - In Ada95 interrupts are interrupts and ATC is embedded in the ‘select’ scheme.

- Asynchronous transfer of control regions:
  - Real-time Java declares ATC-enabled regions per method and any asynchronous event is deferred until the next ATC-enabled method is executing.
  - Ada95 assumes that all code which is called from within an ATC-enabled region is ATC-enabled.

- Handler identification:
  - Real-time Java delivers asynchronous events to all enrolled handlers and propagates an asynchronous interrupting event through the closest handlers.
  - Ada95 delivers an interrupt to one global handler and each ATC-enabled region has exactly one exit point.
Summary

Asynchronism

• **Interrupts / Signals**
  - Device / system / language / operating-system level interrupt control
  - Characteristics of interrupts and signals

• **Exceptions**
  - Exception classes / granularity / parametrisation / propagation
  - Resumption and termination, specific language issues

• **Atomic Actions**
  - Definition / requirements / failure cases / implementation / error recovery

• **Asynchronous transfer of control / Interrupts in context**
  - Interrupts and ATC in real-time Java and Ada95
Synchronization

Uwe R. Zimmer – The Australian National University
References for this chapter

[Ada95RM] (link to on-line version)
Ada Working Group
ISO/IEC JTC1/SC 22/WG 9
Ada 95 Reference Manual
– Language and Standard Libraries
ISO/IEC 8652:1995(E) with COR.1:2000,
June 2001

[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.rtj.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

all references and links are available on the course page
Synchronization

Synchronization methods

• Shared memory based synchronization
  • Semaphores
  • Conditional critical regions
  • Monitors
  • Mutexes & conditional variables
  • Synchronized methods
  • Protected objects

☞ ‘C’, POSIX — Dijkstra
☞ Edison (experimental)
☞ Modula-1, Mesa — Dijkstra, Hoare, …
☞ POSIX
☞ Real-time Java
☞ Ada95

• Message based synchronization
  • Asynchronous messages
  • Synchronous messages
  • Remote invocation, remote procedure call
  • Synchronization in distributed systems

☞ e.g. POSIX, …
☞ e.g. Ada95, CHILL, Occam2
☞ e.g. Ada95, …
☞ e.g. CORBA, …
Synchronization

Synchronization in real-time systems

There are many concurrent entities in a real-time systems:

- Interrupt handlers
- Tasks
- Dispatchers
- Timers
- ...

... and ... real-time systems are often complex and is possibly expanded at a later stage ...

Thus all data is declared ...

- either local (and protected by language-, os-, or hardware-mechanisms)
- or it is ‘out in the open’ and all access need to be synchronized!
Synchronization

Synchronization in real-time systems

Synchronization: the run-time overhead?

Is the potential overhead justified for simple data-structures:

```c
int i;
......
i++; {in one thread}  |  i=0; {in another thread}
```

- Are those operations atomic?
- Do we really need to introduce full featured synchronization methods here?
Synchronization in real-time systems

```
int i;
......
i++;
{in one thread}       |       i=0;
{in another thread}
```

- Depending on the hardware and the compiler, it might be atomic, it might be not:
  - Handling a 64-bit integer on a 8- or 16-bit controller will not be atomic
    … but perhaps it is an 8-bit integer.
  - Any manipulations on the main memory will not be atomic
    … but perhaps it is a register.
  - Broken down to a load-operate-store cycle, the operations will not be atomic
    … but perhaps the processor supplies atomic operations for the actual case.

- Assuming that all ‘perihes’ are applying: how to expand this code?
Synchronization

Synchronization in real-time systems

```c
int i;
......
i++;  {in one thread}  |  i=0;  {in another thread}
```

Unfortunately: the chances that such programming errors turn out are usually small and some implicit by chance synchronization in the rest of the system might prevent them at all.

- Many effects stemming from asynchronous memory accesses are interpreted as (hardware) ‘glitches’, since they are rare and effect usually only some parts of the data.
- On assembler level: synchronization by employing knowledge about the atomicity of CPU-operations and interrupt structures is nevertheless possible and done frequently.

In anything higher than assembler level on small, predictable µcontrollers:

*Measures for synchronization are required!*
Synchronization

Some synchronization terms:

- **Condition synchronization:**
  synchronize a task with an event given by another task.

- **Critical sections:**
  code fragments which contain access to shared resources and need to be executed without interference with other critical sections, sharing the same resources.

- **Mutual exclusion:**
  protection against asynchronous access to critical sections.

- **Atomic operations:**
  the set of operations, which atomicity is guaranteed by the underlying system (e.g. hardware).

  there must be a set of atomic operations to start with!
Synchronization

Synchronization by flags

Word-access atomicity:

Assuming that any access to a word in the system is an atomic operation:

e.g. assigning two values (not wider than the size of word) to a memory cell simultaneously:

\[
\text{Task 1: } x := 0; \quad | \quad \text{Task 2: } x := 5;
\]

will result in either \(x = 0\) xor \(x = 5\) — and no other value is ever observable.
Synchronization

Synchronization by flags

Assuming further that there is a shared memory area between two processes:

- A set of processes agree on a (word-size) atomic variable operating as a flag to indicate synchronization conditions.
Synchronization

Condition synchronization by flags

```pascal
var Flag : boolean := false;

process P1;
  statement X;
  repeat until Flag;
  statement Y;
end P1;

process P2;
  statement A;
  Flag := true;
  statement B;
end P2;
```

Sequence of operations: [A | X] → [B | Y]
Synchronization

Synchronization by flags

Assuming further that there is a shared memory between two processes:

- A set of processes agree on a (word-size) atomic variable operating as a flag to indicate synchronization conditions:

  Memory flag method is ok for simple condition synchronization, but …

  … is not sufficient for general mutual exclusion in critical sections!

  … busy-waiting is required to poll the synchronization condition!

  More powerful synchronization operations are required for critical sections
Synchronization

Synchronization by semaphores

(Dijkstra 1968)

Assuming further that there is a shared memory between two processes:

• a set of processes agree on a variable \( S \) operating as a flag to indicate synchronization conditions ... and ...

• an atomic operation \( P \) on \( S \) — \( P \) stands for ‘passeren’ (Dutch for ‘pass’):
  
  \[
  P: \begin{cases} \text{if } S > 0 \text{ then } S := S - 1 \end{cases} \quad \text{also: ‘Wait’, ‘Suspend\_Until\_True’}
  \]

• an atomic operation \( V \) on \( S \) — \( V \) stands for ‘vrygeven’ (Dutch for ‘to release’):
  
  \[
  V: [S := S + 1] \quad \text{also: ‘Signal’, ‘Set\_True’}
  \]

☞ the variable \( S \) is then called a semaphore.

OS-level: \( P \) is usually also suspending the current task until \( S > 0 \).
CPU-level: \( P \) indicates whether it was successful, but the operation is not blocking.
Synchronization

Condition synchronization by semaphores

```
var sync : semaphore := 0;

process P1;
    statement X;
    wait (sync);
    statement Y;
end P1;

process P2;
    statement A;
    signal (sync);
    statement B;
end P2;
```

Sequence of operations: \([A \mid X] \rightarrow [B \mid Y]\)
Synchronization

Mutual exclusion by semaphores

```pascal
var mutex : semaphore := 1;

process P1;
  statement X;
  wait (mutex);
  statement Y;
  signal (mutex);
  statement Z;
end P1;

process P2;
  statement A;
  wait (mutex);
  statement B;
  signal (mutex);
  statement C;
end P2;
```

Sequence of operations: [A | X] → [B → Y xor Y → B] → [C | Z]
Synchronization

Semaphores

Types of semaphores:

- **General semaphores** *(counting semaphores)*: non-negative number; (range limited by the system) \( P \) and \( V \) increment and decrement the semaphore by one.

- **Binary semaphores**: restricted to \([0, 1]\); Multiple \( V \) (\text{Signal}) calls have the same effect than 1 call.
  - binary semaphores are sufficient to create all other semaphore forms.
  - atomic ‘test-and-set’ operations at hardware level are usually binary semaphores.

- **Quantity semaphores**: The increment (and decrement) value for the semaphore is specified as a parameter with \( P \) and \( V \).
Synchronization

Semaphores in Ada95

package Ada.Synchronous_Task_Control is
    type Suspension_Object is limited private;
    procedure Set_True  (S : in out Suspension_Object);
    procedure Set_False (S : in out Suspension_Object);
    function Current_State (S : Suspension_Object) return Boolean;
    procedure Suspend_Until_True (S : in out Suspension_Object);

private
    ... -- not specified by the language
end Ada.Synchronous_Task_Control;

• only one task can be blocked at **Suspend_Until_True**! (**strict version of a binary semaphore**) (*Program_Error* will be raised with the second task trying to suspend itself)

☞ no queues! ☞ minimal run-time overhead
Synchronization

Semaphores in Ada95

package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set_True  (S : in out Suspension_Object);
  procedure Set_False (S : in out Suspension_Object);
  function Current_State (S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True (S : in out Suspension_Object);
private
  ... -- not specified by the language
end Ada.Synchronous_Task_Control;

• only one task can be blocked at Suspend_Until_True! (strict version of a binary semaphore) (Program_Error will be raised with the second task trying to suspend itself)
☞ no queues ☞ minimal run-time overhead


Real-Time & Embedded Systems

Synchronization

Semaphores in POSIX

```
int sem_init (sem_t *sem_location, int pshared, unsigned int value);
int sem_destroy (sem_t *sem_location);
int sem_wait (sem_t *sem_location);
int sem_trywait (sem_t *sem_location);
int sem_timedwait (sem_t *sem_location, const struct timespec *abstime);
int sem_post (sem_t *sem_location);
int sem_getvalue (sem_t *sem_location, int *value);
```

generate semaphore for usage between processes
(otherwise for threads of the same process only)
Synchronization

Semaphores in POSIX

int sem_init      (sem_t *sem_location, int pshared, unsigned int value);
int sem_destroy   (sem_t *sem_location);
int sem_wait      (sem_t *sem_location);
int sem_trywait   (sem_t *sem_location);
int sem_timedwait (sem_t *sem_location, const struct timespec *abstime);
int sem_post      (sem_t *sem_location);
int sem_getvalue  (sem_t *sem_location, int *value);

delivers the number of waiting processes as a negative integer,
if there are processes waiting on this semaphore
Synchronization

Semaphores in POSIX

```c
void allocate (priority_t P)
{
    sem_wait (&mutex);
    if (busy) {
        sem_post (&mutex);
        sem_wait (&cond[P]);
    }
    busy = 1;
    sem_post (&mutex);
}

sem_t mutex, cond[2];
typedef emun {low, high} priority_t;
int waiting
int busy

void deallocate (priority_t P)
{
    sem_wait (&mutex);
    busy = 0;
    sem_getvalue (&cond[high],
                   &waiting);
    if (waiting < 0) {
        sem_post (&cond[high]);
    }
    else {
        sem_getvalue (&cond[low],
                       &waiting);
        if (waiting < 0) {
            sem_post (&cond[low]);
        }
        else {
            sem_post (&mutex);
        }
    }
}
```
Synchronization

Deadlock by semaphores

with Ada.Synchronous_Task_Control; use Ada.Synchronous_Task_Control;
X, Y : Suspension_Object;

task B;
  task body B is
  begin
    ...
    Suspend_Until_True (Y);
    Suspend_Until_True (X);
  ...
end B;

☞ could raise a Program_Error in Ada95.

☞ produces a potential deadlock when implemented with general semaphores.

☞ Deadlocks can be generated by all kinds of synchronization methods.
Synchronization

Criticism of semaphores

- Semaphores are not bound to any resource or method or region
  - Adding or deleting a single semaphore operation some place might stall the whole system
- Semaphores are scattered all over the code
  - hard to read, error-prone

Semaphores are considered not adequate for the real-time domain.
(all concurrent and real-time languages offer more abstract and safer synchronization methods).
Synchronization

Conditional critical regions

Basic idea:

- Critical regions are a set of code sections in different processes, which are guaranteed to be executed in mutual exclusion:
  - Shared data structures are grouped in named regions and are tagged as being private resources.
  - Processes are prohibited from entering a critical region, when another process is active in any associated critical region.

- Condition synchronisation is provided by guards:
  - When a process wishes to enter a critical region it evaluates the guard (under mutual exclusion). If the guard evaluates false, the process is suspended / delayed.
  - As with semaphores, no access order can be assumed.
Synchronization

Conditional critical regions

buffer : buffer_t;
resource critical_buffer_region : buffer;

process producer;
  loop
    region critical_buffer_region
      when buffer.size < N do
        -- place in buffer etc.
      end region
    end loop;
  end producer

process consumer;
  loop
    region critical_buffer_region
      when buffer.size > 0 do
        -- take from buffer etc.
      end region
    end loop;
  end consumer
Synchronization

Criticism of conditional critical regions

• All guards need to be re-evaluated, when any conditional critical region is left:
  - all involved processes are activated to test their guards
  - there is no order in the re-evaluation phase potential livelocks

• As with semaphores the conditional critical regions are scattered all over the code.
  - on a larger scale: same problems as with semaphores.

The language Edison uses conditional critical regions for synchronization in a multiprocessor environment (each process is associated with exactly one processor).
Synchronization

Monitors

(Modula-1, Mesa — Dijkstra, Hoare)

Basic idea:

- Collect all operations and data-structures shared in critical regions in one place, the monitor.
- Formulate all operations as procedures or functions.
- Prohibit access to data-structures, other than by the monitor-procedures.
- Assure mutual exclusion of the monitor-procedures.
Synchronization

Monitors

```
monitor buffer;
    export append, take;
    var (* declare protected vars *)
    procedure append (I : integer);
        ...
    procedure take (var I : integer);
        ...
begin
    (* initialisation *)
end;
```

How to realize conditional synchronization?
Synchronization

Monitors with condition synchronization

(Hoare)

Hoare-monitors:

- Condition variables are implemented by semaphores (Wait and Signal).
- Queues for tasks suspended on condition variables are realized.
- A suspended task releases its lock on the monitor, enabling another task to enter.

☞ More efficient evaluation of the guards:
the task leaving the monitor can evaluate all guards and the right tasks can be activated.
☞ Blocked tasks may be ordered and livelocks prevented.
Synchronization

Monitors with condition synchronization

```
monitor buffer;
  export append, take;
  var BUF : array [ ... ] of integer;
  top, base : 0..size-1;
  NumberInBuffer : integer;
  spaceavailable, itemavailable : condition;
procedure append (I : integer);
  begin
    if NumberInBuffer = size then
      wait (spaceavailable);
    end if;
    BUF[top] := I; NumberInBuffer := NumberInBuffer+1;
    top := (top+1) mod size;
    signal (itemavailable)
  end append;  ...
```
procedure take (var I : integer);
begin
  if NumberInBuffer = 0 then
    wait (itemavailable);
  end if;
  I := BUF[base];
  base := (base+1) mod size;
  NumberInBuffer := NumberInBuffer - 1;
  signal(spaceavailable);
end take;

begin (* initialisation *)
  NumberInBuffer := 0;
  top := 0; base := 0
end;
Synchronization

Monitors with condition synchronization

Suggestions to overcome the multiple-tasks-in-monitor Problem:

- A `signal` is allowed **only as the last action** of a process before it leaves the monitor.

- A `signal` operation has the side-effect of **executing a return statement**.

- Hoare, Modula-1, POSIX: a `signal` operation which unblocks another process has the side-effect of **blocking the current process**; this process will only execute again once the monitor is unlocked again.

- A `signal` operation which unblocks a process does not block the caller, but the unblocked process must **gain access to the monitor again**.
Monitors in Modula-1

- `wait (s, r)`: delays the caller until condition variable `s` is true (`r` is the rank (or ‘priority’) of the caller).
- `send (s)`: If a process is waiting for the condition variable `s`, then the process at the top of the queue of the highest filled rank is activated (and the caller suspended).
- `awaited (s)`: check for waiting processes on `s`. 
Synchronization

Monitors in Modula-1

INTERFACE MODULE resource_control;

DEFINE allocate, deallocate;
VAR busy : BOOLEAN; free : SIGNAL;

PROCEDURE allocate;
BEGIN
  IF busy THEN WAIT (free) END;
  busy := TRUE;
END;

PROCEDURE deallocate;
BEGIN
  busy := FALSE;
  SEND (free); -- or: IF AWAITED (free) THEN SEND (free);
END;

BEGIN
  busy := false;
END.
Synchronization

Monitors in ‘C’ / POSIX
(types and creation)

Synchronization between POSIX-threads:

```c
typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;
typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;

int pthread_mutex_init (      pthread_mutex_t     *mutex,
                          const pthread_mutexattr_t *attr);
int pthread_mutex_destroy (      pthread_mutex_t     *mutex);
int pthread_cond_init       (      pthread_cond_t      *cond,
                                    const pthread_condattr_t  *attr);
int pthread_cond_destroy    (      pthread_cond_t      *cond);
...
```
Synchronization between POSIX-threads:

typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;
typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;

int pthread_mutex_init (const pthread_mutexattr_t *attr);
int pthread_mutex_destroy (const pthread_mutexattr_t *attr);
int pthread_cond_init (const pthread_condattr_t *attr);
int pthread_cond_destroy (const pthread_condattr_t *attr);
...

Attributes include:

- semantics for trying to lock a mutex which is locked already by the same thread
- sharing of mutexes and condition variables between processes
- priority ceiling
- clock used for timeouts
- ... ... ...
Synchronization

Monitors in ‘C’ / POSIX
(types and creation)

Synchronization between POSIX-threads:

```c
typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;
typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;

int pthread_mutex_init (      pthread_mutex_t     *mutex,      const pthread_mutexattr_t  *attr);
int pthread_mutex_destroy   (      pthread_mutex_t     *mutex);
int pthread_cond_init       (      pthread_cond_t      *cond,      const pthread_condattr_t  *attr);
int pthread_cond_destroy    (      pthread_cond_t      *cond);
```

Undefined, if locked

Undefined, if threads are waiting
Synchronization

Monitors in ‘C’ / POSIX

(operators)

...
Synchronization

Monitors in ‘C’ / POSIX

(operators)

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
int pthread_mutex_timedlock (pthread_mutex_t *mutex,
                            const struct timespec *abstime);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
int pthread_cond_wait (pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_timedwait (pthread_cond_t *cond,
                            pthread_mutex_t *mutex,
                            const struct timespec *abstime);
int pthread_cond_signal (pthread_cond_t *cond);
int pthread_cond_broadcast (pthread_cond_t *cond);
```

unblocking ‘at least one’ thread

unblocking all threads
Synchronization

Monitors in ‘C’ / POSIX

(operators)

...
Synchronization

Monitors in ‘C’ / POSIX

(operators)

... int pthread_mutex_lock (      pthread_mutex_t     *mutex);
int pthread_mutex_trylock   (      pthread_mutex_t     *mutex);
int pthread_mutex_timedlock (      pthread_mutex_t     *mutex,                             const struct timespec     *abstime);
int pthread_mutex_unlock    (      pthread_mutex_t     *mutex);
int pthread_cond_wait       (      pthread_cond_t      *cond,      pthread_mutex_t     *mutex);
int pthread_cond_timedwait  (      pthread_cond_t      *cond,      pthread_mutex_t     *mutex,                              const struct timespec     *abstime);
int pthread_cond_signal     (      pthread_cond_t      *cond);
int pthread_cond_broadcast  (      pthread_cond_t      *cond);

can be called any time, anywhere
(multiple lock reaction can be specified)
Synchronization

Monitors in ‘C’ / POSIX

(example, definitions)

#define BUFF_SIZE 10

typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t buffer_not_full;
    pthread_cond_t buffer_not_empty;
    int count, first, last;
    int buf[BUFF_SIZE];
} buffer;
Monitors in ‘C’ / POSIX
(example, operations)

```c
int append (int item, buffer *B) {
    PTHREAD_MUTEX_LOCK (&B->mutex);
    while (B->count == BUFF_SIZE) {
        PTHREAD_COND_WAIT (
            &B->buffer_not_full,
            &B->mutex);
    }
    PTHREAD_MUTEX_UNLOCK (&B->mutex);
    PTHREAD_COND_SIGNAL (
            &B->buffer_not_empty);
    return 0;
}

int take (int *item, buffer *B) {
    PTHREAD_MUTEX_LOCK (&B->mutex);
    while (B->count == 0) {
        PTHREAD_COND_WAIT (
            &B->buffer_not_empty,
            &B->mutex);
    }
    PTHREAD_MUTEX_UNLOCK (&B->mutex);
    PTHREAD_COND_SIGNAL (
            &B->buffer_not_full);
    return 0;
}
```
Synchronization

Monitors in Real-time Java

Java provides two mechanisms to construct monitors:

- **Synchronized methods and code blocks**
  all methods and code blocks which are using the `synchronized` tag are mutually exclusive with respect to the addressed class.

- **Notification methods:** `wait`, `notify`, and `notifyAll`
  can be used only in synchronized regions and are waking any or all threads, which are waiting in the same synchronized object.
Synchronization

Monitors in Real-time Java

Considerations:

1. Synchronized methods and code blocks:
   - In order to implement a monitor all methods in an object need to be synchronized.
     - any other standard method can break the monitor and enter at any time.
   - Methods outside the monitor-object can synchronize at this object.
     - it is impossible to analyse a monitor locally, since lock accesses can exist all over the system.
   - Static data is shared between all objects of a class.
     - access to static data need to be synchronized over the whole class.

Either in static synchronized blocks:
\[
\text{synchronized (this.getClass())} \{\ldots\}
\]

or in static methods:
\[
\text{public synchronized static <method>} \{\ldots\}
\]
Considerations:

2. Notification methods: `wait`, `notify`, and `notifyAll`

- `wait` suspends the thread and releases the local lock only
  - nested `wait`-calls will keep all enclosing locks.
- `notify` and `notifyAll` does not release the lock.
  - methods, which are activated via notification need to wait for lock-access.
- `wait`-suspended threads are hold in a queue, thus `notifyAll` is waking the threads in order
  - livelocks are prevented at this level (in opposition to Java).
- There are no explicit conditional variables.
  - every notified thread needs
to wait for the lock to be released and to re-evaluate its entry condition
Synchronization

Monitors in Real-time Java

(multiple-readers-one-writer-example)

each of the readers uses these monitor.calls:

```
startRead ();
    // read the shared data only
stopRead ();
```

each of the writers uses these monitor.calls:

```
startWrite ();
    // manipulate the shared data
stopWrite ();
```

construct a monitor, which allows
multiple readers

or

one writer

at a time inside the critical regions
**Synchronization**

**Monitors in Real-time Java**

_(multiple-readers-one-writer-example: wait-notifyAll method)_

```java
public class ReadersWriters
{
    private int readers = 0;
    private int waitingWriters = 0;
    private boolean writing = false;

    ...
```
Synchronization

Monitors in Real-time Java

(multiple-readers-one-writer-example: wait-notifyAll method)

... public synchronized void StartWrite() throws InterruptedException {
    while (readers > 0 || writing) {
        waitingWriters++;
        wait();
        waitingWriters--;
    }
    writing = true;
}

public synchronized void StopWrite() {
    writing = false;
    notifyAll();
} ...
Real-Time & Embedded Systems

Synchronization

Monitors in Real-time Java

(multiple-readers-one-writer-example: wait-notifyAll method)

```
... public synchronized void StartRead () throws InterruptedException {
    while (writing || waitingWriters > 0) {
        wait();
    }
    readers++;
}

public synchronized void StopRead() {
    readers--;
    if (readers == 0) notifyAll();
}
```

whenever a synchronized region is left:

• all thread are notified
• all threads are re-evaluating their guards
Synchronization

Monitors in Real-time Java

Standard monitor solution:

- declare the monitored data-structures private to the monitor object (non-static).
- introduce a class `ConditionVariable`:
  ```java
  public class ConditionVariable {
      public boolean wantToSleep = false;
  }
  ```
- introduce synchronization-scopes in monitor-methods:
  - synchronize on the adequate conditional variables first and
  - synchronize on the monitor-object second.
- make sure that all methods in the monitor are implementing the correct synchronizations.
- make sure that no other method in the whole system is synchronizing on this monitor-object.
Synchronization

Monitors in Real-time Java

(multiple-readers-one-writer-example: usage of external conditional variables)

```java
public class ReadersWriters {
    private int readers = 0;
    private int waitingReaders = 0;
    private int waitingWriters = 0;
    private boolean writing = false;
    ConditionVariable OkToRead = new ConditionVariable();
    ConditionVariable OkToWrite = new ConditionVariable();
    ...
```
... public void StartWrite () throws InterruptedException {
    synchronized (OkToWrite) {
        synchronized (this) {
            if (writing | readers > 0) {
                waitingWriters++;
                OkToWrite.wantToSleep = true;
            } else {
                writing = true;
                OkToWrite.wantToSleep = false;
            }
        }
        if (OkToWrite.wantToSleep) OkToWrite.wait ();
    }
} ...
Synchronization

Monitors in Real-time Java

```java
... public void StopWrite () {
    synchronized (OkToRead) {
        synchronized (OkToWrite) {
            synchronized (this) {
                if (waitingWriters > 0) {
                    waitingWriters--;          
                    OkToWrite.notify (); // wakeup one writer
                } else {                  
                    writing = false;         
                    OkToRead.notifyAll (); // wakeup all readers
                    readers = waitingReaders; 
                    waitingReaders = 0;     
                }
            }
        }
    }
}
...```
Monitors in Real-time Java

... public void StartRead () throws InterruptedException {
    synchronized (OkToRead) {
        synchronized (this) {
            if (writing | waitingWriters > 0) {
                waitingReaders++;
                OkToRead.wantToSleep = true;
            } else {
                readers++;
                OkToRead.wantToSleep = false;
            } 
        } 
        if (OkToRead.wantToSleep) OkToRead.wait (;
    } } ...
Synchronization

Monitors in Real-time Java

```java
... public void StopRead ()
{
    synchronized (OkToWrite)
    {
        synchronized (this)
        {
            readers--;
            if (readers == 0 & waitingWriters > 0) {
                waitingWriters--;
                OkToWrite.notify ();
            }
        }
    }
}
```
Synchronization

Object-orientation and synchronization

Since mutual exclusion, notification, and condition synchronization schemes need to be designed and analysed considering the implementation of all involved methods and guards:

☞ new methods cannot be added without re-evaluating the whole class!

In opposition to the general re-usage idea of object-oriented programming, the re-usage of synchronized classes (e.g. monitors) need to be considered carefully.

☞ The parent class might need to be adapted in order to suit the global synchronization scheme.

☞ Inheritance anomaly (Matsuoka & Yonezawa ‘93)

Methods to design and analyse expandible synchronized systems exist, but are fairly complex and are not provided in any current object-oriented language.
Synchronization

Monitors in POSIX & Real-time Java

 Cosby flexible and universal, but relies on conventions rather than compilers

POSIX offers conditional variables

Real-time Java is more supportive than POSIX in terms of data-encapsulation

Extreme care must be taken when employing object-oriented programming and monitors
Synchronization

Nested monitor calls

Assuming a thread in a monitor is calling an operation in another monitor and is suspended at a conditional variable there:

☞ the called monitor is aware of the suspension and allows other threads to enter.
☞ the calling monitor is possibly not aware of the suspension and keeps its lock!
☞ the unjustified locked calling monitor reduces the system performance and leads to potential deadlocks.

Suggestions to solve this situation:

• Maintain the lock anyway: e.g. POSIX, Real-time Java
• Prohibit nested procedure calls: e.g. Modula-1
• Provide constructs which specify the release of a monitor lock for remote calls, e.g. Ada95
Synchronization

Criticism of monitors

• Mutual exclusion is solved elegantly and safely.

• Conditional synchronization is on the level of semaphores still
  ☞ all criticism on semaphores apply

☞ mixture of low-level and high-level synchronization constructs.
Synchronization

Synchronization by protected objects

Combine

- the **encapsulation** feature of monitors

with

- the **coordinated entries** of conditional critical regions

to

Protected objects

- *all* controlled data and operations are encapsulated
- *all* operations are mutual exclusive
- entry guards are *attached* to operations
- the protected interface allows for operations on data
- no protected data is accessible (other than by defined operations)
- tasks are queued (according to their priorities)
Synchronization

Synchronization by protected objects in Ada95
(simultaneous read-access)

Some read-only operations do not need to be mutual exclusive:

protected type Shared_Data (Initial : Data_Item) is
  function  Read return Data_Item;
  procedure Write (New_Value : in Data_Item);
private
  The_Data : Data_Item := Initial;
end Shared_Data_Item;

- protected functions can have ‘in’ parameters only and are not allowed to alter the private data (enforced by the compiler).

protected functions allow simultaneous access (but mutual exclusive with other operations).

- there is no defined priority between functions and other protected operations in Ada95.
Synchronization

Synchronization by protected objects in Ada95

Condition synchronization is realized in the form of protected procedures combined with boolean conditional variables (barriers): \textit{entries} in Ada95:

\begin{verbatim}
Buffer_Size : constant Integer := 10;
type Index is mod Buffer_Size;
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer_T is array (Index) of Data_Item;

protected type Bounded_Buffer is
  entry Get (Item : out Data_Item);
  entry Put (Item : in Data_Item);
private
  First  : Index := Index'First;
  Last   : Index := Index'Last;
  Num    : Count := 0;
  Buffer : Buffer_T;
end Bounded_Buffer;
\end{verbatim}
Synchronization

Synchronization by protected objects in Ada95

(barriers)

protected body Bounded_Buffer is

entry Get (Item : out Data_Item) when Num > 0 is
begin
    Item  := Buffer (First);
    First := First + 1;
    Num   := Num - 1;
end Get;

entry Put (Item : in Data_Item) when Num < Buffer_Size is
begin
    Last          := Last + 1;
    Buffer (Last) := Item;
    Num           := Num + 1;
end Put;

end Bounded_Buffer;
Synchronization

Synchronization by protected objects in Ada95

Protected entries are used like task entries:

```ada
Buffer : Bounded_Buffer;

select
  Buffer.Put (Some_Data);
or
  delay 10.0;
  -- do something after 10 s.
end select;

select
  Buffer.Get (Some_Data);
else
  -- do something else
end select;

select
  delay 10.0;
  then abort
    Buffer.Put (Some_Data);
  -- try to enter for 10 s.
end select;

select
  Buffer.Get (Some_Data);
  then abort
    -- meanwhile try something else
end select;
```
Synchronization

Synchronization by protected objects in Ada95

(barrier evaluation)

Barrier evaluations and task activations:

- on **calling a protected entry**, the associated barrier is evaluated
  (only those parts of the barrier which might have changed since the last evaluation).

- on **leaving a protected procedure or entry**, related barriers with tasks queued are evaluated
  (only those parts of the barriers which might have been altered by this procedure / entry
  or which might have changed since the last evaluation).

Barriers are not evaluated **while inside** a protected object or **on leaving a protected function**.
Synchronization

Synchronization by protected objects in Ada95
(operations on entry queues)

The count attribute indicate the number of tasks waiting at a specific queue:

```ada
protected Blocker is
  entry Proceed;
private
  Release : Boolean := False;
end Blocker;

protected body Blocker is
  entry Proceed
  when Proceed'count = 5
  or Release is
  begin
    Release := Proceed'count > 0;
    end Proceed;
  end Proceed;
end Blocker;
```
Synchronization

Synchronization by protected objects in Ada95

(operations on entry queues)

The count attribute indicates the number of tasks waiting at a specific queue:

```ada
protected type Broadcast is
  entry Receive (M: out Message);
  procedure Send (M: in  Message);
private
  New_Message  : Message;
  Arrived      : Boolean := False;
end Blocker;

protected body Broadcast is
  entry Receive (M: out Message)
    when Arrived
  is
    begin
      M := New_Message
      Arrived := Receive'count > 0;
    end Proceed;
  procedure Send (M: in  Message)
  is
    begin
      New_Message := M;
      Arrived := Receive'count > 0;
    end Send;
end Blocker;
```
Synchronization

Synchronization by protected objects in Ada95

(entry families, requeue & private entries)

Further refinements on task control by:

• **Entry families:**
  a protected entry declaration can contain a discrete subtype selector, which can be evaluated by the barrier (other parameters cannot be evaluated by barriers) and implements an array of protected entries.

• **Requeue facility:**
  protected operations can use ‘requeue’ to redirect tasks to other internal, external, or private entries. The current protected operation is finished and the lock on the object is released.

  ‘Internal progress first’-rule: internally requeued tasks are placed at the head of the waiting queue!

• **Private entries:**
  protected entries which are not accessible from outside the protected object, but can be employed as destinations for requeue operations.
Synchronization

Synchronization by protected objects in Ada95

(package families)

package Modes is

type Mode_T is
  (Takeoff, Ascent, Cruising, Descent, Landing);

protected Mode_Gate is
  procedure Set_Mode
    (Mode: in Mode_T);
  entry Wait_For_Mode
    (Mode_T);

private
  Current_Mode : Mode_Type := Takeoff;

end Mode_Gate;
end Modes;

package body Modes is

protected body Mode_Gate is

  procedure Set_Mode
    (Mode: in Mode_T) is
    begin
      Current_Mode := Mode;
      end Set_Mode;

  entry Wait_For_Mode
    (for Mode in Mode_T)
      when Current_Mode = Mode is
      begin null;
      end Wait_For_Mode;

end Mode_Gate;
end Modes;
Synchronization by protected objects in Ada95
(requeue & private entries)

How to implement a queue, at which every task can be released only once per triggering event?

☞ e.g. by employing two entries:

```ada
package Single_Release is
  entry     Wait;
  procedure Trigger;
private
  Front_Door,   Main_Door  : Boolean := False;
  entry Queue;
end Single_Release;
```
Synchronization

Synchronization by protected objects in Ada95

(requeue & private entries)

package body Single_Release is

entry Wait
    when Front_Door is
    begin
    if Wait'Count = 0 then
        Front_Door := False;
        Main_Door  := True;
    end if;
    requeue Queue;
end Wait;

entry Queue
    when Main_Door is
    begin
    if Queue'count = 0 then
        Main_Door := False;
    end if;;
end Queue;

procedure Trigger is
    begin
    Front_Door := True;
end Trigger;

end Single_Release;

opening the main door before requeuing?
Synchronization

Synchronization by protected objects in Ada95
(restrictions applying to protected operations)

Code inside a protected procedure, function or entry is bound to non-blocking operations
(which would keep the whole protected object locked).

Thus the following operations are prohibited:

- entry call statements
- delay statements
- task creations or activations
- calls to sub-programs which contains a potentially blocking operation
- select statements
- accept statements

☞ The `requeue` facility allows for a potentially blocking operation, but releases the current lock!
Summary

Shared memory based synchronization

General

Criteria:

• level of abstraction
• centralized vs. distributed concepts
• support for consistency and correctness validations
• error sensitivity
• predictability
• efficiency
Summary

Shared memory based synchronization

POSIX

- all low level constructs available.
- no connection with the actual data-structures.
- error-prone.
- non-determinism introduced by ‘release some’ semantics of conditional variables (cond_signal).
Summary

Shared memory based synchronization

Real-time Java

- mutual exclusion (synchronized methods) as the only support.
- general notification feature (no conditional variables)
- non-restricted object oriented extension introduces hard to predict timing behaviours.
Summary

Shared memory based synchronization

Modula-1, CHILL

- full monitor implementation (Dijkstra-Hoare monitor concept).
  
  ... no more, no less, ...

☞ all features of and criticism about monitors apply.
Summary

Shared memory based synchronization

Ada95

- complete synchronization support
- low-level semaphores for very special cases.
- predictable timing (scheduler).

Most memory oriented synchronization conditions are realized by the compiler or the run-time environment directly rather than the programmer.

(Ada95 is currently without any mainstream competitor in this field)
Synchronization

Message-based synchronization

- Synchronization model
  - Asynchronous
  - Synchronous
  - Remote invocation

- Addressing (name space)
  - direct communication
  - mail-box communication

- Message structure
  - arbitrary
  - restricted to ‘basic’ types
  - restricted to un-typed communications
Message-based synchronization

Asynchronous messages

If there is a listener:
☞ send the message directly
Synchronization

Message-based synchronization

Asynchronous messages

If the receiver becomes available at a later stage:
- the message needs to be buffered
Message-based synchronization

Synchronous messages

Delay the sender:
- until the receiver got the message

Diagram:
- P1 sends a synchronous message to P2.
- P2 receives the message.
- Time progresses.
Synchronization

Message-based synchronization

Synchronous messages

Delay the sender:

- until the receiver got the message
- two asynchronous messages required
Synchronization

Message-based synchronization

Synchronous messages

Delay the sender until:

- a receiver is available
- a receiver got the message
Synchronization

Message-based synchronization

Synchronous messages

If the receiver becomes available at a later stage:
- messages need to be buffered
Message-based synchronization

Remote invocation

Delay the sender, until:

• a receiver got the message
• a receiver executed an addressed routine
Synchronization

Message-based synchronization

Remote invocation

Delay the sender, until:

- a receiver got the message
- a receiver executed an addressed routine
Synchronization

Message-based synchronization

Remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine
Synchronization

Message-based synchronization

Remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine
Synchronization

Message-based synchronization

Asynchronous remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
Synchronization

Message-based synchronization

Asynchronous remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
Synchronization

Synchronous vs. asynchronous communications

Purpose ‘synchronization’: synchronous messages / remote invocations
Purpose ‘in-time delivery’: asynchronous messages / asynchronous remote invocations

☞ ‘Real’ synchronous message passing in distributed systems requires hardware support.
☞ Asynchronous message passing requires the usage of (infinite?) buffers.

• Synchronous communications are emulated by a combination of asynchronous messages in some systems.
• Asynchronous communications can be emulated in synchronized message passing systems by introducing ‘buffer-tasks’ (de-coupling sender and receiver as well as allowing for broadcasts).
Synchronization

Addressing (name space)

Direct vs. indirect:

send \langle message \rangle \text{ to } \langle process-name \rangle
wait for \langle message \rangle \text{ from } \langle process-name \rangle
send \langle message \rangle \text{ to } \langle mailbox \rangle
wait for \langle message \rangle \text{ from } \langle mailbox \rangle

Asymmetrical addressing:

send \langle message \rangle \text{ to } \ldots
wait for \langle message \rangle

☞ Client-server paradigm
## Synchronization

### Addressing (name space)

Communication medium:

<table>
<thead>
<tr>
<th>Connections</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>one-to-one</td>
<td>buffer, queue, synchronization</td>
</tr>
<tr>
<td>one-to-many</td>
<td>multicast</td>
</tr>
<tr>
<td>one-to-all</td>
<td>broadcast</td>
</tr>
<tr>
<td>many-to-one</td>
<td>local server, synchronization</td>
</tr>
<tr>
<td>all-to-one</td>
<td>general server, synchronization</td>
</tr>
<tr>
<td>many-to-many</td>
<td>general network- or bus-system</td>
</tr>
</tbody>
</table>
Synchronization

Message structure

- Machine dependent representations need to be taken care of in a distributed environment.
- Communication system is often outside the typed language environment.

Most communication systems are handling streams (packets) of a basic element type only.

Conversion routines for data-structures other than the basic element type are supplied ...

... manually (POSIX)
... semi-automatic (Real-time CORBA)
... automatic and are typed-persistent (Ada95)
package Ada.Streams is
pragma Pure (Streams);

  type Root_Stream_Type is abstract tagged limited private;
  type Stream_Element is mod implementation-defined;
  type Stream_Element_Offset is range implementation-defined;
  subtype Stream_Element_Count is
    Stream_Element_Offset range 0..Stream_Element_Offset'Last;

  type Stream_Element_Array is
    array (Stream_Element_Offset range <> ) of Stream_Element;

  procedure Read (…) is abstract;
  procedure Write (…) is abstract;

private
  … -- not specified by the language
end Ada.Streams;
Synchronization

Message structure (Ada95)

Reading and writing values of any type to a stream:

```ada
procedure S'Write(
  Stream : access Ada.Streams.Root_Stream_Type'Class; Item : in  T);
procedure S'Class'Write(
  Stream : access Ada.Streams.Root_Stream_Type'Class; Item : in  T'Class);
procedure S'Read(
  Stream : access Ada.Streams.Root_Stream_Type'Class; Item : out T);
procedure S'Class'Read(
  Stream : access Ada.Streams.Root_Stream_Type'Class; Item : out T'Class)
```

Reading and writing values, bounds and discriminants of any type to a stream:

```ada
procedure S'Output(
  Stream : access Ada.Streams.Root_Stream_Type'Class; Item : in  T);
function S'Input(
  Stream : access Ada.Streams.Root_Stream_Type'Class) return T;
```
Synchronization

Message-based synchronization

Practical message-passing systems:

<table>
<thead>
<tr>
<th>System</th>
<th>Type Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSIX</td>
<td>“message queues”: ordered indirect [asymmetrical</td>
</tr>
<tr>
<td>CHILL</td>
<td>“buffers”, “signals”: ordered indirect [asymmetrical</td>
</tr>
<tr>
<td>Occam2</td>
<td>“channels”: indirect symmetrical synchronous fully-typed one-to-one message passing</td>
</tr>
<tr>
<td>Ada95</td>
<td>“(extended) rendezvous”: ordered direct asymmetrical [synchronous</td>
</tr>
<tr>
<td>Real-time Java</td>
<td>no communication via messages available</td>
</tr>
</tbody>
</table>
## Synchronization

### Message-based synchronization

Practical message-passing systems:

<table>
<thead>
<tr>
<th>System</th>
<th>ordered</th>
<th>symmetrical</th>
<th>asymmetrical</th>
<th>synchronous</th>
<th>asynchronous</th>
<th>direct</th>
<th>indirect</th>
<th>one-to-one</th>
<th>many-to-one</th>
<th>many-to-many</th>
<th>method</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSIX:</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>message passing</td>
</tr>
<tr>
<td>CHILL:</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>message passing</td>
</tr>
<tr>
<td>Occam2:</td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>message passing</td>
</tr>
<tr>
<td>Ada95:</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>remote invocation</td>
</tr>
<tr>
<td>Real-time Java:</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>no communication via messages available</td>
</tr>
</tbody>
</table>

- POSIX: *Fully typed* message passing
- CHILL: *Fully typed* message passing
- Occam2: *Fully typed* message passing
- Ada95: *Fully typed* remote invocation
- Real-time Java: no communication via messages available
### Synchronization

**Message-based synchronization**

Practical message-passing systems for synchronisation purposes:

<table>
<thead>
<tr>
<th>Method</th>
<th>Contents</th>
<th>One-to-one</th>
<th>Many-to-one</th>
<th>Many-to-many</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSIX:</td>
<td>bytes</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHILL:</td>
<td>typed</td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Occam2:</td>
<td>fully typed</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>Ada95:</td>
<td>fully typed</td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Real-time Java:</td>
<td>no communication via messages available</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* POSIX: message passing
* CHILL: message passing
* Occam2: message passing
* Ada95: remote invocation
* Real-time Java: no communication via messages available
Synchronization

Message-based synchronization in Occam2

Communication is ensured by means of a ‘channel’, which:

- can be used by one writer and one reader process only
- and is synchronous:

CHAN OF INT SensorChannel:

PAR

  INT reading:
  SEQ i = 0 FOR 1000
  SEQ
    -- generate reading
    SensorChannel ! reading

  INT data:
  SEQ i = 0 FOR 1000
  SEQ
    SensorChannel ? data
    -- employ data
tasks are synchronized at these points
Synchronization

Message-based synchronization in CHILL

CHILL is the ‘CCITT High Level Language’, where CCITT is the Comité Consultatif International Télégraphique et Téléphonique. The CHILL language development was started in 1973 and standardized in 1979.

☞ strong support for concurrency, synchronization, and communication
   (monitors, buffered message passing, synchronous channels)

```chill
dcl SensorBuffer buffer (32) int;
...
send SensorBuffer (reading);

receive case
  (SensorBuffer in data) : ... 
esac;

signal SensorChannel = (int) to consumertype;
...
send SensorChannel (reading)
  to consumer

receive case
  (SensorChannel in data): ... 
esac;
```

© 2009 Uwe R. Zimmer, The Australian National University
Synchronization

Message-based synchronization in CHILL

CHILL is the ‘CCITT High Level Language’, where CCITT is the Comité Consultatif International Télégraphique et Téléphonique. The CHILL language development was started in 1973 and standardized in 1979.

- strong support for concurrency, synchronization, and communication
  (monitors, buffered message passing, synchronous channels)

```chill
dcl SensorBuffer buffer (32) int;
...
send SensorBuffer (reading);
receive case
- asynchronous
  (SensorBuffer in data) : ...
esac;

signal SensorChannel = (int) to consumertype;
...
send SensorChannel (reading) to consumer
receive case
- synchronous
  (SensorChannel in data) : ...
esac;
```

© 2009 Uwe R. Zimmer, The Australian National University
Synchronization

Message-based synchronization in Ada95

Ada95 supports remote invocations ((extended) rendezvous) in form of:

- **entry points** in tasks
- **full set of parameter profiles** supported

If the local and the remote task are on different architectures, or if an intermediate communication system is employed:

☞ parameters incl. bounds and discriminants are ‘tunnelled’ through byte-stream-formats.

Synchronization:

- both tasks are synchronized at the beginning of the remote invocation (☞ ‘rendezvous’)
- the calling task if blocked until the remote routine is completed (☞ ‘extended rendezvous’)

© 2009 Uwe R. Zimmer, The Australian National University
Message-based synchronization in Ada95

Remote invocation (Rendezvous)

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
- a receiver started an addressed routine
Synchronization

Message-based synchronization in Ada95

Remote invocation (Extended rendezvous)

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine
- a receiver passed the results
Synchronization

Message-based synchronization in Ada95

(Rendezvous)

...<entry_name>[(index)]<parameters>...
...-- waiting for synchronization...
...--
...--
synchronized
...--
accept <entry_name>[(index)]
<parameter_profile>;

...
Synchronization

Message-based synchronization in Ada95
(Rendezvous)

```ada
accept <entry_name> [(index)] <parameter_profile>;
-- waiting for synchronization
```

```
<entry_name> [(index)] <parameters>
```

```
synchronized
```

...
Synchronization

Message-based synchronization in Ada95

(Extended rendezvous)

```plaintext
... <entry_name> [(index)] <parameters> ...
    ... -- waiting for synchronization ...
    ... -- ...
    ... -- ...
    synchronized ...
    ... -- blocked ...
    ... -- ...
    return results ...

accept <entry_name> [(index)] <parameter_profile> do
    ... -- ...
    ... -- remote invocation ...
    ... -- ...
end <entry_name>;
... 
```
Synchronization

Message-based synchronization in Ada95
(Extended rendezvous)

```ada
accept <entry_name>[<index>][<parameters>] do
  ... -- waiting for synchronization
end <entry_name>;
```

```
synchronized
```

```
return results
```

```
... -- remote invocation
... --
... --
... --
... --
... --
... --
... --
... --

...
**Synchronization**

**Message-based synchronization in Ada95**

Some things to consider for task-entries:

- In contrast to protected-object-entries, task-entries can call other blocking operations.
- Accept statements can be nested (but need to be different).
  - Helpful e.g. to synchronize more than two tasks.
- Accept statements can have a dedicated exception handler (like any other code-block).
  - Exceptions, which are not handled during the rendezvous phase are propagated to all involved tasks.
- Parameters cannot be direct 'access' parameters, but can be access-types.
- 'Count' on task-entries is defined, but is only accessible from inside the tasks owning the entry.
- **Entry families** (arrays of entries) are supported.
- **Private entries** (accessible for internal tasks) are supported.
Synchronization

Selective waiting

Dijkstra’s guarded commands:

```plaintext
if x <= y -> m := x
fi
```

☞ the programmer needs to design the alternatives as ‘parallel’ options:
  - all cases need to be covered and overlapping conditions need to lead to the same result

Extremely different philosophy: ‘C’-switch:

```plaintext
switch (x) {
    case 1: r := 3;
    case 2: r := 2; break;
    case 3: r := 1;
}
```

☞ the sequence of alternatives has a crucial role.
Synchronization

Selective waiting in Occam2

\(\text{ALT} \quad \text{Guard}_1 \quad \text{Process}_1 \quad \text{Guard}_2 \quad \text{Process}_2 \quad \ldots\)

- Guards are referring to boolean expressions and/or channel input operations.
- The boolean expressions are local expressions, i.e. if none of them evaluates to true at the time of the evaluation of the \text{ALT}-statement, then the process is stopped.
- If all triggered channel input operations evaluate to false, the process is suspended until further activity on one of the named channels.
- Any Occam2 process can be employed in the \text{ALT}-statement
- The \text{ALT}-statement is non-deterministic (there is also a deterministic version: \text{PRI ALT}).
Synchronization

Selective waiting in Occam2

ALT

\[
\begin{align*}
\text{NumberInBuffer} & \leftarrow \text{Size} \& \text{Append} \ ? \ \text{Buffer} \ [\text{Top}] \\
\text{SEQ} & \\
\text{NumberInBuffer} & := \text{NumberInBuffer} + 1 \\
\text{Top} & := (\text{Top} + 1) \ \text{REM} \ \text{Size} \\
\text{NumberInBuffer} & > 0 \ & \text{Request} \ ? \ \text{ANY} \\
\text{SEQ} & \\
\text{Take} & ! \ \text{Buffer} \ [\text{Base}] \\
\text{NumberInBuffer} & := \text{NumberInBuffer} - 1 \\
\text{Base} & := (\text{Base} + 1) \ \text{REM} \ \text{Size}
\end{align*}
\]

- synchronization on input-channels only:

☞ to initiate the sending of data (Take ! Buffer [Base]),
  a request need to be made first (Request ? ANY)

CSP (Hoare) also supports non-deterministic selective waiting
Synchronization

Message-based selective synchronization in Ada95

Forms of selective waiting:

\[
\text{select_statement ::= selective_accept | conditional_entry_call | timed_entry_call | asynchronous_select}
\]

... underlying concept: Dijkstra’s guarded commands

selective_accept implements ...

- ... wait for more than a single rendezvous at any one time
- ... time-out if no rendezvous is forthcoming within a specified time
- ... withdraw its offer to communicate if no rendezvous is available immediately
- ... terminate if no clients can possibly call its entries
Synchronization

Message-based selective synchronization in Ada95

selective_accept in its full syntactical form in Ada95:

```
selective_accept ::= select
    [guard] selective_accept_alternative
    { or [guard] selective_accept_alternative
    [ else sequence_of_statements ]
    end select;

guard ::= when <condition> =>

selective_accept_alternative ::= accept_alternative    |
                             delay_alternative     |
                             terminate_alternative

accept_alternative ::= accept_statement [ sequence_of_statements ]
delay_alternative ::= delay_statement [ sequence_of_statements ]
terminate_alternative ::= terminate;
```
Synchronization

Basic forms of selective synchronization

(select-or)

```plaintext
select
    accept ... do ...  
end ...

or
accept ... do ...
end ...

or
accept ... do ...
end ...

or
accept ... do ...
end ...

...

end select;
```

- If none of the named entries have been called, the task is suspended until one of the entries is addressed by another task.
- The selection of an accept is non-deterministic, in case that multiple entries are called.
- The selection can be controlled by means of the real-time systems annex.
- The select statement is completed, when at least one of the entries has been called and those accept-block has been executed.
Synchronization

Basic forms of selective synchronization

(guarded select-or)

```pascal
select
  when <condition> =>
    accept ... do ...
  end ...

or
  when <condition> =>
    accept ... do ...
  end ...

or
  when <condition> =>
    accept ... do ...
  end ...

...
end select;
```

• Analogue to Dijkstra’s guarded commands
• all accepts closed will raise a Program_Error
☞ set of conditions need to be complete
Synchronization

Basic forms of selective synchronization

(guarded select-or-else)

```
select
   [ when <condition> => ]
      accept ... do ...    
      end ...            
or
   [ when <condition> => ]
      accept ... do ...    
      end ...            
or
   [ when <condition> => ]
      accept ... do ...    
      end ...            
else
   <statements>
```

- If none of the open entries can be accepted immediately, the else alternative is selected.
- There can be only one else alternative and it cannot be guarded.
Synchronization

Basic forms of selective synchronization

(guarded select-or-delay)

select
    [ when <condition> => ]
    accept ... do ...
    end ...

or
    [ when <condition> => ]
    delay ...
    <statements>

or
    [ when <condition> => ]
    delay ...
    <statements>

... end select;

• If none of the open entries has been called before the amount of time specified in the earliest open delay alternative, this delay alternative is selected.

• There can be multiple delay alternatives if more than one delay alternative expires simultaneously, either one may be chosen.

• delay and delay until can be employed.
Synchronization

Basic forms of selective synchronization

(guarded select-or-terminate)

```
select
   [ when <condition> => ]
      accept ... do ...
   end ...

or

   [ when <condition> => ]
      accept ... do ...
   end ...

or

   [ when <condition> => ]
      terminate;

... end select;
```

The terminate alternative is chosen if none of the entries can ever be called again, i.e.:

- all tasks which can possibly call any of the named entries are terminated.

or

- all remaining active tasks which can possibly call any of the named entries are waiting on selective terminate statements and none of their open entries can be called any longer.

☞ This task and all its dependent waiting-for-termination tasks are terminated together.
Synchronization

Basic forms of selective synchronization

(guarded select-or-else select-or-delay select-or-terminate)

```
select
    [ when <condition> => ]      accept ... do ...      end ...
else
    <statements>...end select;
```

```
select
    [ when <condition> => ]      accept ... do ...      end ...
else
    <statements>...end select;
```

```
select
    [ when <condition> => ]      accept ... do ...      end ...
end ...
```

```
[ when <condition> => ] delay ...
<statements>
end select;
```

```
else-delay-terminate
alternatives
cannot be mixed!
```

```
else
    <statements>
... end select;
```

```
select
    [ when <condition> => ]      accept ... do ...      end ...
end ...
```

```
[ when <condition> => ]
    terminate;
... end select;
```
Synchronization

Non-determinism in selective synchronizations

If equal alternatives are given, then the program correctness (incl. the timing specifications) must not be affected by the actual selection.

- If alternatives have different priorities, this can be expressed e.g. by means of the Ada real-time annex.

- Non-determinism in concurrent systems is or can be also introduced by:
  - non-ordered monitor or other queues
  - buffering / routing message passing systems
  - non-deterministic schedulers
  - timer quantization
  - … any form of asynchronism
Synchronization

Conditional & timed entry-calls

\[
\text{conditional\_entry\_call ::= select entry\_call\_statement [sequence\_of\_statements] else sequence\_of\_statements end select;}
\]

\[
\text{timed\_entry\_call ::= select entry\_call\_statement [sequence\_of\_statements] or delay\_alternative end select;}
\]

\[
\text{select Light\_Monitor.Wait\_for\_Light; Lux := True; else Lux := False; end select;}
\]

\[
\text{select Controller.Request (Medium) (Some\_Item); -- process data or delay 45.0; -- try something else end select;}
\]
Conditional & timed entry-calls

conditional_entry_call ::= select
  entry_call_statement [sequence_of_statements]
else
  sequence_of_statements
end select;

select
  Light_Monitor.Wait_for, Lux := True;
else
  Lux := False;
end;

timed_entry_call ::= select
  entry_call_statement [sequence_of_statements]
or
  delay_alternative
end select;

select
  Controller.Request (Medium) (Some_Item); -- process data
  delay 45.0; -- try something else
end select;

There is only
  one entry call
and either
  one 'else'
or
  one 'or delay'
end select;
Synchronization

Conditional & timed entry-calls

conditional_entry_call ::= 
  select 
  entry_call_statement 
  [sequence_of_statements] 
  else 
  sequence_of_statements 
end select;

timed_entry_call ::= 
  select 
  entry_call_statement 
  [sequence_of_statements] 
  or 
  delay_alternative 
end select;

select 
  Light_Monitor.Wait_for_Light; 
  Lux := True; 
else 
  Lux := False; 
end select;

The idea in both cases is to withdraw a synchronization request and not to implement polling or busy-waiting.

select 
  Controller.Request (Medium) 
  (Some_Item); 
  -- process data 
  or 
  delay 45.0; 
  -- try something else 
end select;
Summary

Synchronization

• Shared memory based synchronization
  • Flags, condition variables, semaphores, ...
    … conditional critical regions, monitors, protected objects.
  • Guard evaluation times, nested monitor calls, deadlocks, ...
    … simultaneous reading, queue management.
  • Synchronization and object orientation, blocking operations and re-queuing.

• Message based synchronization
  • Synchronization models, addressing modes, message structures
  • Selective accepts, selective calls
  • Indeterminism in message based synchronization
Real-Time Scheduling

Uwe R. Zimmer – The Australian National University
References for this chapter

[Burns01]
Alan Burns and Andy Wellings
*Real-Time Systems and Programming Languages*
Addison Wesley, third edition, 2001

[Murthy01]
C. Siva Ram Murthy, G. Manimaran
*Resource Management in Real-time Systems and Networks*

all references and links are available on the course page
Scheduling in Real-Time Systems

Concurrency may lead to non-determinism

Non-determinism may make it harder to predict the timing behaviour

RT-Scheduling schemes reduce non-determinism
Scheduling

A scheduling scheme provides two features:

- **Ordering the use of resources** (e.g. CPUs, networks)
- **Predicting the worst-case behaviour** of the system when the scheduling algorithm is applied

The prediction can then be used

☞ at compile-run:
   - to **confirm the overall temporal requirements** of the application

or

☞ at run-time:
   - to **permit acceptance** of additional usage/reservation requests.
Scheduling schemes

• **Static**
  all predictions and schedules are done off-line
  
  often better predictability \(\Rightarrow\) most hard real-time systems

• **Dynamic**
  run-time situation is taken into account
  
  more flexible, more efficient \(\Rightarrow\) most soft real-time systems
Scheduling as task queuing

- **create** → batch
- Ready state:
  - ready
  - suspended
  - pre-emption or cycle done
- Blocked state:
  - blocked
  - suspended
  - block or synchronize
- Termination (term.)
Real-time scheduling as task queuing

- Create batch
- Batch ready
- Ready admission new tasks if set is still schedulable
- CPU dispatching according to deadlines, priorities, or values
- Block or synchronize
- Pre-emption or cycle done
- Term.
Real-time scheduling

A simple process model

- The number of processes in the system is fixed.
- All processes are periodic and all periods are known.
- All processes are independent.
- The task-switching overhead is negligible.
- All deadlines are identical with the process cycle times (periods).
- The worst case execution time is known for all processes.
- All processes are released at once.

This model can only be applied to a specific group of hard real-time systems. (Extensions to this model will be discussed later in this chapter.)
Real-time scheduling

Example: Requested times

\[(T_i, C_i)\]

- (16, 8)
- (12, 3)
- (4, 1)
Real-time scheduling

Example: Deadlines
Dynamic scheduling

Earliest deadline first (EDF)

1. Determine (one of) the process(es) with the closest deadline.

2. Execute this process
   
   2-a until it finishes
   
   2-b or until another process’ deadline is found closer than the current one.

☞ Pre-emptive scheme
☞ Dynamic scheme,
   since the dispatched process is selected at run-time, due to the current deadlines.
Dynamic scheduling: Earliest Deadline First (EDF)

Earliest deadline first

1. Schedule the earliest deadline first
2. Avoid task switches (in case of equal deadlines)
Dynamic scheduling: Earliest Deadline First (EDF)

Earliest deadline first: Response times

worst case response times $R_i$ (maximal time in which the request from task $T_i$ is served).
Dynamic scheduling: Earliest Deadline First (EDF)

Earliest deadline first: Response times

worst case response times $R_i$ (maximal time in which the request from task $T_i$ is served):

- can be close or identical to deadlines.
- small or none spare capacity, if any task misses its expected computation time.
Dynamic scheduling: Earliest Deadline First (EDF)

Earliest deadline first: Maximal utilization

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \quad \iff \text{sufficient & necessary test!}$$

with $C_i$, $T_i$ the computation and cycle times of task $i$

(maximal possible utilization: the deadlines $D_i$ are assumed to be identical with the cycles times $T_i$ here)
Real-Time & Embedded Systems

Static scheduling

Fixed Priority Scheduling (FPS), rate monotonic

1. Each process is assigned a fixed priority according to its cycle time $T_i$:

   $$ T_i < T_j \Rightarrow P_i > P_j $$

2. At any point in time: dispatch the process with the highest priority

   - Pre-emptive scheme
   - Static scheme, since the order dispatch order of processes is fixed and calculated at off-line.

   - Rate monotonic ordering is **optimal** (in the framework of fixed priority schedulers), i.e. if a process set is schedulable under a FPS-scheme, it is also schedulable by applying rate monotonic priorities.
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic priorities

assign task priorities according to the cycle times $T_i$ (identical to deadline $D_i$).
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic priorities

assign task priorities according to the cycle times $T_i$ (identical to deadline $D_i$).
assign task priorities according to the cycle times $T_i$ (identical to deadline $D_i$).
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic priorities

max. utilization test: \[ \sum_{i=1}^{n} \frac{C_i}{T_i} \leq N \left( \frac{1}{2^N - 1} \right) \]

☞ sufficient, but not necessary test!
Real-Time & Embedded Systems

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic priorities

utilization test: \[ \sum_{i=1}^{n} \frac{C_i}{T_i} = 1 > 0.779 \approx N \left( \frac{1}{2^N} - 1 \right) \]

not guaranteed!
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic priorities *(reduced requests)*

\[
\text{max. utilization test: } \sum_{i=1}^{n} \frac{C_i}{T_i} \leq N \left( \frac{1}{2^N} - 1 \right)
\]
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic priorities \( (\textit{reduced requests}) \)

Utilization: \[ \frac{6}{16} + \frac{3}{12} + \frac{1}{4} = 0.875 > 0.779 \approx 3 \left( \frac{1}{2^3} - 1 \right) \]

\[ \sum_{i=1}^{n} \frac{C_i}{T_i} \leq N \left( \frac{1}{2^N} - 1 \right) \]

\( \text{not guaranteed!} \)
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic priorities (reduced requests)

utilization: \( \frac{6}{16} + \frac{3}{12} + \frac{1}{4} = 0.875 > 0.779 \approx 3 \left( \frac{1}{2^3} - 1 \right) \); \( \sum_{i=1}^{n} \frac{C_i}{T_i} \leq N \left( \frac{1}{2^N} - 1 \right) \)

☞ not guaranteed!
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic priorities (further reduced requests)

max. utilization test: \[ \sum_{i=1}^{n} \frac{C_i}{T_i} \leq N \left( \frac{1}{2^N} - 1 \right) \]
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic priorities (further reduced requests)

Utilization: \[
\frac{4}{16} + \frac{3}{12} + \frac{1}{4} = 0.75 \leq 0.779 \approx 3 \left( \frac{1}{2^3} - 1 \right);
\sum_{i=1}^{n} \frac{C_i}{T_i} \leq N \left( \frac{1}{2^N} - 1 \right) \leq \text{guaranteed!}
\]
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic priorities (further reduced requests)

\[ \text{utilization: } \frac{4}{16} + \frac{3}{12} + \frac{1}{4} = 0.75 \leq 0.779 \approx 3 \left( \frac{1}{2^3} - 1 \right) ; \quad \frac{n}{\sum_{i=1}^{n} \frac{C_i}{T_i}} \leq N \left( \frac{1}{2^N} - 1 \right) \quad \text{guaranteed!} \]
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Utilisation based Analysis for FPS rate monotonic

$$U \equiv \sum_{i=1}^{n} \frac{C_i}{T_i} \leq N \left(2^{\frac{1}{N}} - 1\right) \equiv U_{\text{max}}$$

- with $C_i$ the computation time and $T_i$ the length of the period for task $i$ out of $N$ tasks and assuming that the deadline $D_i = T_i$

- sufficient, but not necessary
- $O(n)$ complexity
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (further reduced requests)

- calculate the worst case response times for each task individually.
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (further reduced requests)

R₃

(16,4)
(12,3)
(4,1)

(Tᵢ, Cᵢ)

for the highest priority task: R₃ = C₃
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (further reduced requests)

\[ R_i = C_i + I_i = \text{computation } C_i + \text{interference } I_i \]
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (further reduced requests)

for other tasks: $R_i = C_i + \sum_{j>i} \left[ \frac{R_j}{T_j} \right] C_j$
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis

\[ R_i = C_i + \sum_{j > i} \left[ \frac{R_i}{T_j} \right] C_j \]

☞ fixed-point equation!

☞ form recurrent equation: \( R_i^{k+1} = C_i + \sum_{j > i} \left[ \frac{R_i^k}{T_j} \right] C_j \) (1)

☞ starting with \( R_i^0 = C_i \)

☞ Iterate (1) until \( R_i^{k+1} = R_i^k \) or \( R_i^{k+1} > T_i \)
Real-Time & Embedded Systems

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis

\[
R_i^{k+1} = C_i + \sum_{j > i} \left( R_i^k \right) \frac{C_j}{T_j} (1); \quad R_i^0 = C_i \quad \text{- Iterate (1) until } R_i^{k+1} = R_i^k \text{ or } R_i^{k+1} > T_i
\]

Example (further reduced requests):

• set of tasks: \{ (T_i, C_i) \} = \{ (16, 4); (12, 3); (4, 1) \} at priorities \{ 1; 2; 3 \}; \ R_3^0 = 1

\[
R_3^0 = 1 (\checkmark)
\]
Response time analysis

\[ R_i^{k+1} = C_i + \sum_{j > i} \left[ \frac{R_i^k}{T_j} \right] C_j \] (1); \( R_i^0 = C_i \) — Iterate (1) until \( R_i^{k+1} = R_i^k \) or \( R_i^{k+1} > T_i \)

Example (further reduced requests):
- set of tasks: \( \{(T_i, C_i)\} = \{(16, 4);(12, 3);(4, 1)\} \) at priorities \( \{1;2;3\}; R_2^0 = 3 \)

\[ R_2^1 = 3 + \left[ \frac{3}{4} \right] 1 = 4 \]
\[ R_2^1 = 3 + \left[ \frac{4}{4} \right] 1 = 4(\checkmark) \]
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis

\[ R_i^{k+1} = C_i + \sum_{j > i} \left( \frac{R_i^k}{T_j} \right) C_j \] (1); \[ R_i^0 = C_i \]

Iterate (1) until \[ R_i^{k+1} = R_i^k \] or \[ R_i^{k+1} > T_i \]

Example (further reduced requests):

- set of tasks: \{ (T_i, C_i) \} = \{ (16, 4); (12, 3); (4, 1) \} at priorities \{ 1; 2; 3 \}; \[ R_1^0 = 4 \]

\[ R_1^1 = 4 + \left[ \frac{4}{12} \right] 3 + \left[ \frac{4}{4} \right] 1 = 8 \]

\[ R_1^2 = 4 + \left[ \frac{8}{12} \right] 3 + \left[ \frac{8}{4} \right] 1 = 9 \]

\[ R_1^3 = 4 + \left[ \frac{9}{12} \right] 3 + \left[ \frac{9}{4} \right] 1 = 10 \]

\[ R_1^4 = 4 + \left[ \frac{10}{12} \right] 3 + \left[ \frac{10}{4} \right] 1 = 10 (√) \]
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis

\[
R_i^{k+1} = C_i + \sum_{j > i} \left[ \frac{R_i^k}{T_j} \right] C_j \quad (1) \quad R_i^0 = C_i \quad \text{Iterate (1) until } R_i^{k+1} = R_i^k \text{ or } R_i^{k+1} > T_i
\]

Example (reduced requests):

- set of tasks: \{ (T_i, C_i) \} = \{ (16, 6); (12, 3); (4, 1) \} at priorities \{ 1; 2; 3 \}; \quad R_1^0 = 6

\[
R_1^1 = 6 + \left[ \frac{6}{12} \right] 3 + \left[ \frac{6}{4} \right] 1 = 11
\]

\[
R_1^2 = 6 + \left[ \frac{11}{12} \right] 3 + \left[ \frac{11}{4} \right] 1 = 12
\]

\[
R_1^3 = 6 + \left[ \frac{12}{12} \right] 3 + \left[ \frac{12}{4} \right] 1 = 12 (\checkmark)
\]
Real-Time & Embedded Systems

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis

\[ R_i^{k+1} = C_i + \sum_{j>i} \left\lceil \frac{R_j^k}{T_j} \right\rceil C_j \] (1); \[ R_i^0 = C_i \] — Iterate (1) until \[ R_i^{k+1} = R_i^k \] or \[ R_i^{k+1} > T_i \]

Example (full requests):
• set of tasks: \((T_i, C_i)\) = \{(16, 8);(12, 3);(4, 1)\} at priorities \{1;2;3\}; \( R_i^0 = 8 \)

\[ R_1^1 = 8 + \left\lceil \frac{8}{12} \right\rceil 3 + \left\lceil \frac{8}{4} \right\rceil 1 = 13 \]

\[ R_1^2 = 8 + \left\lceil \frac{13}{12} \right\rceil 3 + \left\lceil \frac{13}{4} \right\rceil 1 = 18(\color{red} \times) \]

\[ R_1^2 = 8 + \left\lceil \frac{18}{12} \right\rceil 3 + \left\lceil \frac{18}{4} \right\rceil 1 = 19(\color{red} \times) \]

\[ R_1^2 = 8 + \left\lceil \frac{19}{12} \right\rceil 3 + \left\lceil \frac{19}{4} \right\rceil 1 = 19(\color{red} \times) \]
Dynamic scheduling: Earliest Deadline First (EDF)

Response time analysis

The worst case for EDF is not necessarily when all tasks are released at once!
☞ all possible combinations in a full hyper-cycle need to be considered!

- The response times are bounded by the cycle times as long as the maximal utilization is ≤ 1.
- Other tasks need to be considered only, if their deadline is closer or equal to the current task.
Dynamic scheduling: Earliest Deadline First (EDF)

Response time analysis

\[ R_i(a) = \left[ \frac{a}{T_i} + 1 \right] C_i + \sum_{j \neq i_{\text{min}}} \left\{ \left[ \frac{R_i(a)}{T_j} \right], \left[ \frac{a + T_i - T_j}{T_j} \right] + 1 \right\} C_j \]
Dynamic scheduling: Earliest Deadline First (EDF)

Response time analysis

\[ R_i(a) = \left[ \frac{a}{T_i} + 1 \right] C_i + \sum_{j \neq i_{\text{min}}} \left\{ \left[ \frac{R_i(a)}{T_j} \right] \text{max}, \left\{ 0, \left[ \frac{a + T_i - T_j}{T_j} \right] + 1 \right\} \right\} C_j \]

\[ R_i^{k+1}(a) = \left[ \frac{a}{T_i} + 1 \right] C_i + \sum_{j \neq i_{\text{min}}} \left\{ \left[ \frac{R_i^k(a)}{T_j} \right] \text{max}, \left\{ 0, \left[ \frac{a + T_i - T_j}{T_j} \right] + 1 \right\} \right\} C_j \quad (2) \]

starting with \( R_i^0(a) = a + C_i \)

Iterate (2) until \( R_i^{k+1}(a) = R_i^k(a) \) (or \( R_i^{k+1}(a) > a + T_i \) utilization beyond 100%)
Dynamic scheduling: Earliest Deadline First (EDF)

Response time analysis

\[ R_i(a) = \left[ \frac{a}{T_i} + 1 \right] C_i + \sum_{j \neq i_{\text{min}}} \left\{ \left[ \frac{R_i(a)}{T_j} \right] \max \left\{ 0, \left[ \frac{a + T_i - T_j}{T_j} \right] + 1 \right\} \right\} C_j \]

\[ R_i^{k+1}(a) = \left[ \frac{a}{T_i} + 1 \right] C_i + \sum_{j \neq i_{\text{min}}} \left\{ \left[ R_i^k(a) \right] \max \left\{ 0, \left[ \frac{a + T_i - T_j}{T_j} \right] + 1 \right\} \right\} C_j \quad (2) \]

Starting with \( R_i^0(a) = a + C_i \)

Iterate (2) until \( R_i^{k+1}(a) = R_i^k(a) \)

\[ R_i = \max \left\{ R_i(a) - a \right\}_{a \in A} ; \quad \text{where} \ A = \text{scm}\{T_i\} \]
Response time analysis (further reduced requests)

\[ R_i = C_i + \sum_{j>i} \left[ \frac{R_j}{T_j} \right] C_j; \quad R_3 = 1\checkmark; \quad R_2 = 4\checkmark; \quad R_1 = 10\checkmark \text{ and } \sum_{i=1}^{n} \frac{C_i}{T_i} \leq N \left( \frac{1}{2^N} - 1 \right) \checkmark \]
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (reduced requests)

\[ R_i = C_i + \sum_{j > i} \left[ \frac{R_j}{T_j} \right] C_j; \quad R_3 = 1 \checkmark; \quad R_2 = 4 \checkmark; \quad R_1 = 12 \checkmark \quad \text{but} \quad \sum_{i=1}^{n} \frac{C_i}{T_i} > N \left( \frac{1}{2^N} - 1 \right) \]
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (full requests)

\[ R_i = C_i + \sum_{j > i} \left[ \frac{R_j}{T_j} \right] C_j; \]  
\[ R_3 = 1 \checkmark; R_2 = 4 \checkmark; R_1 = 19 \times \]  
and  
\[ \sum_{i=1}^{n} \frac{C_i}{T_i} > N \left( \frac{1}{2^N} - 1 \right) \times \]
Dynamic scheduling: Earliest Deadline First (EDF)

Response time analysis (full requests)

Testing all combinations in a hyper-period: LCM of \( \{ T_i \} \) — here: 48
Dynamic scheduling: Earliest Deadline First (EDF)

Response time analysis (full requests)

Testing all combinations in a hyper-period: LCM of \( \{T_i\} \) — here: 48

\[
R_-: 16 \leq 16 \checkmark = T_-; \quad R_-: 12 \leq 12 \checkmark = T_-; \quad R_-: 4 \leq 4 \checkmark = T_-
\]
Dynamic scheduling: Earliest Deadline First (EDF)

Response time analysis (reduced requests)

.relaxed task-set changes:

\[ R_\cdot: 16 \rightarrow 12 \leq 16 \checkmark = T_\cdot; \quad R_\cdot: 12 \rightarrow 8 \leq 12 \checkmark = T_\cdot; \quad R_\cdot: 4 \rightarrow 1 \leq 4 \checkmark = T_\cdot \]
Dynamic scheduling: Earliest Deadline First (EDF)

Response time analysis (further reduced requests)

Further relaxed task-set changes:

\[ R_{-} : 12 \rightarrow 10 \leq 16 = T_{-} ; \quad R_{-} : 8 \rightarrow 6 \leq 12 = T_{-} ; \quad R_{-} : 1 \rightarrow 1 \leq 4 = T_{-} \]
### Real-time scheduling

#### Response time analysis (comparison)

<table>
<thead>
<tr>
<th></th>
<th>Fixed Priority Scheduling</th>
<th>Earliest Deadline First</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>utilization test</td>
<td>response times ${R_i}$</td>
</tr>
<tr>
<td>$(T_i, C_i)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(16, 8);(12, 3);(4, 1)$</td>
<td>$\times (1.000)$</td>
<td>${ \times, 4, 1 }$</td>
</tr>
<tr>
<td>$(16, 6);(12, 3);(4, 1)$</td>
<td>$\times (0.875)$</td>
<td>${ 12, 4, 1 }$</td>
</tr>
<tr>
<td>$(16, 4);(12, 3);(4, 1)$</td>
<td>$\checkmark (0.750)$</td>
<td>${ 10, 4, 1 }$</td>
</tr>
</tbody>
</table>

\[
\frac{n}{\sum_{i=1}^{n} C_i / T_i} \leq N \left( \frac{1}{2^N} - 1 \right)
\]

\[
C_i + \sum_{j > i} \left[ \frac{R_i}{T_j} \right] C_j
\]

\[
\frac{n}{\sum_{i=1}^{n} C_i / T_i} \leq 1
\]

check full hyper-cycle

© 2009 Uwe R. Zimmer, The Australian National University
Real-time scheduling

Fixed Priority Scheduling $\leftrightarrow$ Earliest Deadline First

- EDF can handle higher (full) utilization than FPS.
- FPS is easier to implement and implies less run-time overhead

- Graceful degradation features (resource is over-booked):
  - FPS: processes with lower priorities will always miss their deadlines first.
  - EDF: any process can miss its deadline and can trigger a cascade of failed deadlines.

- Response time analysis and utilization tests:
  - FPS: $O(n)$ utilization test — response time analysis: fixed point equation
  - EDS: $O(n)$ utilization test — response time analysis: fixed point equation in hyper-cycle
Scheduling

Constraints which we used up to here:

- tasks are periodic

- deadlines are identical with task’s period time ($D = T$)

- pre-emptive scheduling

- worst case execution times are known

- tasks are independent
Extensions which we will introduce:

• tasks are periodic
  ➥ we will introduce sporadic and aperiodic processes

• deadlines are identical with task’s period time \( (D = T) \)
  ➥ we will introduce arbitrary deadlines

• pre-emptive scheduling
  ➥ we will introduce (briefly) cooperative scheduling

• worst case execution times are known
  ➥ we will introduce fault tolerant scheduling

• tasks are independent
  ➥ we will introduce schedules for interacting tasks
Scheduling — real-world considerations

Including

aperiodic, sporadic & ‘soft’ real-time tasks
Real-Time & Embedded Systems

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Hard real-time tasks
Introducing soft real-time tasks
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Introducing soft real-time tasks

- set can be scheduled using average computation and period times
- hard real-time tasks can be scheduled under worst case conditions (including worst case behaviours of soft real-time tasks)
**Static scheduling: FPS, rate monotonic + server**

**Introducing a server task**

Server is established at a high priority
Deferrable Server (DS): Capacity replenished every $T_s$ (here: 8)
Introducing a server task: Sporadic Server

Sporadic Server (SS): Capacity replenished $T_s$ units after $t_s$ ♦ POSIX
Static scheduling: Fixed Priority Scheduling (FPS), dual-priorities

Introducing dual priorities

Start hard rt-tasks in low priorities; promote them in time to higher ones.
Dynamic scheduling: Earliest Deadline First+ aperiodic server

Introducing a server task to EDF

(T_i, C_i, D_i)
Dynamic scheduling: Earliest Deadline First + aperiodic server

Introducing a server task to EDF
Dynamic scheduling: Earliest Deadline First + aperiodic tasks

Switching between EDF & Earliest Deadline Last (EDL)

\[ (T_i, C_i) \]

1 \hspace{1cm} 5 \hspace{1cm} 10 \hspace{1cm} 15 \hspace{1cm} 20 \hspace{1cm} 25 \hspace{1cm} 30 \hspace{1cm} 35 \hspace{1cm} 40 \hspace{1cm} 45 \hspace{1cm} 50 \]

Sporadic
Hard RT

EDL
EDL
Scheduling — real-world considerations

Including

tasks with deadlines shorter than their cycle time
Tasks with $D < T$

(Deadline earlier than inter-arrival period)

In fixed priority scheduling (FPS): change from:

Rate Monotonic Priority Ordering (RMPO)

to:

Deadline Monotonic Priority Ordering (DMPO)

Lemma

Any task set $Q$ which is schedulable by a FPS scheme $W$, is also schedulable by DMPO!
Any task set $Q$ which is schedulable by a FPS scheme $W$, is also schedulable by DMPO!

Proof

1. $i, j$ are two tasks in $Q$, with $(P_i = P_j + 1) \land (D_i > D_j)$ in $W \nRightarrow \neg DMPO$

2. Generate $W'$ by swapping $P_i$ and $P_j$, i.e. $(P_i' < P_j') \land (D_i > D_j) \Rightarrow DMPO$

3. $W'$ is scheduling $Q$ because:

   3-a all $t_k \in Q$ with $P_k > P_i$ or $P_k < P_j$ are unaffected

   3-b $t_j$ is schedulable in $W'$ because $P_j' > P_i \Rightarrow R_j' \leq R_j \leq D_j$

   3-c $t_i$ is schedulable in $W'$ because …
Proof

1. \( t_i, t_j \) are two tasks in \( Q \), with \( P_i > P_j \) and \( D_i > D_j \) in \( W \) \( \not\implies \) DMPO

2. Generate \( W' \) by swapping \( P_i \) and \( P_j \), i.e. \( (P_i' < P_j') \land (D_i > D_j) \) \( \not\implies \) DMPO

3. \( W' \) is scheduling \( Q \) because:

   3-a all \( t_k \in Q \) with \( P_k > P_i \) or \( P_k < P_j \) are unaffected

   3-b \( t_j \) is schedulable in \( W' \) because \( P_j' > P_i \Rightarrow R_j' \leq R_j \leq D_j \)

   3-c \( t_i \) is schedulable in \( W' \) because

   in \( W \): \( R_j \leq D_j < D_i \leq T_i \Rightarrow R_j < T_i \) i.e. \( t_i \) interfered only once with \( t_j \)

   in \( W \): \( t_j \) released once in \( R_j \), and \( R_i < R_j \)

   \( \not\implies \) in \( W' \): \( t_j \) interferes only once with \( t_i \) \( \not\implies \) \( R_i' = R_j \leq D_j < D_i \Rightarrow R_i' < D_i \)
Static scheduling: Fixed Priority Scheduling (FPS), DMPO

$t_i$ is still schedulable in $W'$ because:

\[
R_j \leq D_j < D_i \leq T_i \Rightarrow R_j < T_i \quad \text{i.e.} \quad t_i \text{ interfered only once with } t_j
\]

in $W$: $t_j$ released once in $R_j$, and $R_i < R_j$

in $W'$...
Static scheduling: Fixed Priority Scheduling (FPS), DMPO

$t_i$ is still schedulable in $W'$ because:

- In $W$: $t_j$ released once in $R_j$, and $R_i < R_j$
- In $W'$: $t_j$ interferes only once with $t_i$ $\Rightarrow \ R_i' = R_j \leq D_j < D_i \Rightarrow R_i' < D_i$
Static scheduling: Fixed Priority Scheduling (FPS), DMPO

Any task set $Q$ which is schedulable by a FPS scheme $W$, is also schedulable by DMPO.

- Swap all $t_i, t_j$ in $Q$, with $P_i > P_j$ and $D_i > D_j$ in $W$ resulting in all $t_i, t_j$ in $Q$, with $P_i > P_j$ have $D_i < D_j$

  Deadline Monotonic Priority Ordering (DMPO)

- Since the each swapping operation keeps schedulability, the final priority scheme (DMPO) is also schedulable.

  If FPS-DMPO is not schedulable, there is no schedulable FPS-scheme.
Scheduling — real-world considerations

Including
task interdependencies
Scheduling: Interdependencies

Schedule for independent tasks

(independent task set)
Schedule for independent tasks

(independent task set)
Scheduling: Interdependencies

Synchronized via lock

(interdependent task set $\bowtie$ lock $\Rightarrow$ shared between $\color{red}{\text{and}}$ $\color{green}{\text{and}}$ $\Rightarrow$ )
Scheduling: Interdependencies

Synchronized via lock

Priority inversion

(interdependent task set ↔ lock ▲ shared between ▲ and ▲)

© 2009 Uwe R. Zimmer, The Australian National University
Scheduling: Interdependencies

Priority inheritance

Task $t_i$ inherits the priority of $t_j$, if:

1. $P_i < P_j$

2. task $t_i$ has locked a resource $Q$

3. task $t_j$ is blocked waiting for resource $Q$ to be released
Scheduling: Interdependencies

Priority inheritance

Maximal blocking time for task $t_i$: $B_i = \sum_{r=1}^{R} \text{usage}(r, i) C(r)$

- with $R$ the number of critical sections
- $\text{usage}(r, i)$ a boolean (0/1) function indicating that $r$ is used by at least one $t_j$ with $P_j < P_i$ and at least one $t_k$ with $P_k \geq P_i$
- $C(r)$ is the worst case computation time in critical section $r$

A task can only be blocked once for each employed resource!
Scheduling: Interdependencies

Priority inheritance

( inherits priority of , when is in lock and is dispatched)
Real-Time & Embedded Systems

Scheduling: Interdependencies

Without priority inheritance

☞ Priority inversion

(Interdependent task set ☞ lock ♤ shared between ♣ and ♣)
Scheduling: Interdependencies

A more complex example

(independent task set)
Scheduling: Interdependencies

A more complex example

(independent task set)
Scheduling: Interdependencies

Interdependencies

Lock 1
Lock 2
1
2
3
Priority inversion
Scheduling: Interdependencies

Priority inheritance

( and inherit priority of , when in lock and is dispatched)
Scheduling: Interdependencies

Priority inheritance

( and inherit priority of , when in lock and is dispatched)
Scheduling: Interdependencies

One additional lock request
Scheduling: Interdependencies

One additional lock request

Deadlock
Immediate ceiling priority protocol (POSIX, Ada, RT-Java)

- Each task $t_i$ has static default priority $P_i$.
- Each resource (lock, monitor) $R_k$ has a static ceiling priority $C_k$, which is the maximum of priorities of the tasks $t_i$ which employ this resource.
  \[ C_k = \max_i \{ \text{employ}(i, k) \cdot P_i \} \]
- Each task $t_i$ has a dynamic priority $P_{iD}$, which is the maximum of its own static priority and the ceiling priorities of any resource it has locked.
  \[ P_{iD} = \max \{ P_i, \max_k \{ \text{locked}(i, k) \cdot C_k \} \} \]
Immediate ceiling priority protocol (POSIX, Ada, RT-Java)

(, and inherit the ceiling priority of or when entering the lock)
Real-Time & Embedded Systems

Scheduling: Interdependencies: Priority ceiling protocols

Immediate ceiling priority protocol *(POSIX, Ada, RT-Java)*

(, , and inherit the ceiling priority of or when entering the lock)
Immediate ceiling priority protocol (POSIX, Ada, RT-Java)

- Tasks are dispatched only if all employed resources are available.
- Deadlocks are prevented
- Number of context switches is reduced
Scheduling: Interdependencies: Priority ceiling protocols

Immediate ceiling priority protocol \((\text{POSIX, Ada, RT-Java})\)

Maximal blocking time: \(B_i = \max_{r=1}^{R} \{\text{usage}(r, i) \cdot C(r)\}\)

- with \(R\) the number of critical sections
- \(\text{usage}(r, i)\) a boolean \((0/1)\) function indicating that \(r\) is used by at least one \(t_j\) with \(P_j < P_i\) and at least one \(t_k\) with \(P_k \geq P_i\)
- \(C(r)\) is the worst case computation time in critical section \(r\)

a task can only be blocked once by any lower priority task!
Scheduling — real-world considerations

Considering

non-pre-emptive scheduling
Cooperative Scheduling

• All schemes up to here used pre-emptive dispatching.

• In interdependent task sets maximal blocking times $B_i$ can be determined for each task $t_i$ when employing a priority ceiling protocol.

• If the overall maximal blocking time $B_{max}$ can be accepted by all tasks, the number of pre-emption can significantly reduced by:

**Deferred pre-emption – Cooperative Scheduling**

☞ Each task $t_i$ is divided in $k$ non-pre-emptive blocks of $C_{i,k} \leq B_{max}$

☞ All critical sections are completely enclosed in one code-block

☞ Each task calls a ‘de-scheduling’ kernel routine at the end of each code-block, i.e. ‘offering’ a task-switch.
Cooperative Scheduling

Deferred pre-emption – Cooperative Scheduling

☞ Number of task switches is reduced
☞ Caches, pre-fetching, and pipelines are more efficient
☞ Execution times are (a bit) easier to predict
☞ Schedules are simpler
☞ Interdependent task sets are schedulable deadlock free
Deferred pre-emption – Cooperative Scheduling

Response times:

\[ R_i = R_i^n + F_i, \text{ with } R_i^{k+1} = B_{\text{max}} + C_i - F_i + \sum_{j > i} \left[ \frac{R_i}{T_j} \right] C_j \]

and \( F_i \) the execution time of the final code-block

… in the simplified case \( C = C_i = C_j = F_i = B_{\text{max}} \):

\[ R_i = R_i^n, \text{ with } R_i^{k+1} = C + \sum_{j > i} \left[ \frac{R_i}{T_j} \right] C \]

… and with even \( T = T_j \forall i \):

\[ R_i = C + \sum_{j > i} C \]
Real-Time & Embedded Systems

Cooperative Scheduling

Deferred pre-emption – Cooperative Scheduling

What’s the cost?

• Code block division need to be done very thoroughly.

• Additional protection against badly behaving (non-cooperative) tasks:
  ☞ Scheduler pre-empts tasks, which fail to offer a ‘de-schedule’ themselves.

• Due to a central $B_{max}$ value, additional tasks need to be engineered to participate in a specific cooperative schedule.

• Requires that a $B_{max}$ value can be accepted by all tasks
  ☞ very short and reactive tasks are excluded or will be treated specially.
Scheduling — real-world considerations

Considering
deadlines beyond the release period
Tasks with $D > T$

(Deadline later than inter-arrival period)

(a cross-over of a hard, periodic and a soft real-time task)

Assuming that a task $t_i$ is released only after a former release of $t_i$ is completed.

☞ In case that $R > T$ for a specific scheduling situation, the following release of task $t_i$ is delayed until completion of the former release.

☞ Mind that $R > T$ cannot hold for all release situations, otherwise the task is not schedulable.

☞ The worst case response time $R_i$ might thus be longer than $T_i$ but must still be shorter than $D_i$. 
Tasks with $D > T$

(Deadline later than inter-arrival period)

Since the response time $R$ can now be potentially greater than the cycle time $T$: more than one release $q$ of the task $t_i$ needs to be considered:

$$R_i(q) = B_i + qC_i + \sum_{j > i}^{\infty} \frac{R_j(q)}{T_j} C_j$$

where $\forall q | (R_i(q) - (q - 1)T_i \leq D_i)!$

with $q$: number of releases

$$R_i = \max \{ R_i(q) - (q - 1)T_i | q \in \{1 ... q_{\text{max}}\} \} \text{ and } q_{\text{max}} = \min \left\{ q \left| \frac{R_i(q)}{q} \leq T_i \right\} \right.$$
Scheduling — real-world considerations

Considering

‘fault-tolerance’
(additional CPU-time for exception handling and recovery)
Fault Tolerance

Exceptions and Recoveries

Task $t_i$ needs extra CPU-time $C_i^f$ for error recovery or exception handling (done at $P_i$) and the minimum inter-arrival time between faults is $T_f$:

$$R_i = B_i + C_i + \sum_{j > i} \left[\frac{R_i}{T_j}\right] C_j + \max_{k \geq i} \left\{\left[\frac{R_i}{T_f}\right] C_k^f\right\}$$

if error recovery or exception handling is performed at the highest priority:

$$R_i = B_i + C_i + \sum_{j > i} \left[\frac{R_i}{T_j}\right] C_j + \max_{k} \left\{\left[\frac{R_i}{T_f}\right] C_k^f\right\}$$
Scheduling — real-world considerations

Considering

task sets with release offsets
Task sets with offsets

Some task sets can be scheduled by introducing offsets to the release times, but ...

☞ without any further restrictions this problem is NP-hard!

... by introducing further assumptions about the **granularity** of the *period times* and *deadlines*:

☞ the schedulability analysis’ complexity can be reduced to be realistic.
Scheduling support in different real-time languages / environments
Language Support for Scheduling: Ada95

Ada95 provides:

- Task and interrupt priorities (static, dynamic, active)
- Task attributes
- Prioritized entry queues
- Priority ceiling locking (ICPP)
- Schedulers (at least FIFO within priorities (pre-emptive) is requested)

Ada95 does not provide:

- Earliest Deadline First (EDF)
- Sporadic servers (a Ada95-implementation of a sporadic server is on the course page)
- Direct task execution time measurements (e.g. POSIX or VxWorks timers)
package System is

    subtype Any_Priority is Integer
        range implementation-defined;

    subtype Priority is Any_Priority
        range Any_Priority'First .. implementation-defined;

    subtype Interrupt_Priority is Any_Priority
        range Priority'Last+1 .. Any_Priority'Last;

    Default_Priority : constant Priority :=
        (Priority'First + Priority'Last)/2;

    end System;

package Ada.Dynamic_Priorities is

    procedure Set_Priority (Priority : in System.Any_Priority;
        T        : in Ada.Task_Identification.Task_ID
            := Ada.Task_Identification.Current_Task);

    function Get_Priority (T : Ada.Task_Identification.Task_ID
        return System.Any_Priority;

end Ada.Dynamic_Priorities;
POSIX provides:

- Task and interrupt priorities (static, dynamic, active)
- Prioritized message queues
- Priority ceiling locking (ICPP)
- Schedulers, priority based with at least:
  - FIFO, Round-Robin, Sporadic Server, possibly others
- Threads can be
  - ‘system contented’ or
  - ‘process contented’ (priority scheduling unclear in this case)
- Timers
Real-Time Java provides:

- Task priorities (static, dynamic, active)
- Prioritized message queues
- Priority ceiling locking (ICPP)
- Schedulable objects (associated with threads) with
  - memory, release, and scheduling parameters
- Pre-emptive priority-oriented dispatching, possibly with a feasibility analysis
- An extendible scheduler class ☞ dynamic scheduling

Real-Time Java does not (necessarily) provide:

- Earliest Deadline First (EDF)
- Sporadic servers
- Direct task execution time measurements (might be provided)
public abstract class Scheduler
{
    protected Scheduler ();
    protected abstract boolean addToFeasibility (Schedulable s);
    public abstract void fireSchedulable (Schedulable s);
    public abstract boolean isFeasible ();
    protected abstract boolean removeFromFeasibility (Schedulable s);
    public boolean setIfFeasible (Schedulable s,
                    ReleaseParameters r,
                    MemoryParameters m);

    ...
}

Formulates an on-line schedulability analysis!
public class PriorityScheduler extends Scheduler {
    public static final int MAX_PRIORITY;
    public static final int MIN_PRIORITY;
    protected PriorityScheduler ();
    protected boolean addToFeasibility (Schedulable s);
    public void fireSchedulable (Schedulable s);
    public boolean isFeasible ();
    protected boolean removeFromFeasibility (Schedulable s);
    public boolean setIfFeasible (Schedulable s,
                                  ReleaseParameters r,
                                  MemoryParameters m);
    ...
Real-time Java

```java
public abstract class SchedulingParameters {
    public SchedulingParameters ();
}

public class PriorityParameters extends SchedulingParameters {
    public PriorityParameters (int priority);
    public int getPriority ();
    public void setPriority (int priority) throws …;
    …
}
```

`Priority` is the only default scheduling parameter
public abstract class ReleaseParameters
{
    protected ReleaseParameters (RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler);

    public RelativeTime      getCost();
    public AsyncEventHandler getCostOverrunHandler();
    public RelativeTime      getDeadline();
    public AsyncEventHandler getDeadlineMissHandler();
}
public class PeriodicParameters extends ReleaseParameters {
    public PeriodicParameters (HighResolutionTime start, RelativeTime period,
        RelativeTime cost, RelativeTime deadline,
        AsyncEventHandler overrunHandler,
        AsyncEventHandler missHandler);

    public RelativeTime getPeriod () ;
    public HighResolutionTime getStart () ;
    public void setPeriod (RelativeTime period);
    public void setStart (HighResolutionTime start);
}
public class AperiodicParameters extends ReleaseParameters {
    public AperiodicParameters (RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler);
}

these are the minimum release parameters (while cost might be used for feasibility analysis only)

the deadline-missHandler need to be supplied in any implementation
Real-time Java

```java
public class SporadicParameters extends AperiodicParameters {

    public SporadicParameters (RelativeTime minInterarrival,
                              RelativeTime cost,
                              RelativeTime deadline,
                              AsyncEventHandler overrunHandler,
                              AsyncEventHandler missHandler);

    public RelativeTime getMinimumInterarrival ();
    public void         setMinimumInterarrival (RelativeTime minimum);
}
```

Sporadic events are not allowed to come in bursts!
Summary

Scheduling

• **Basic real-time scheduling**
  - Fixed Priority Scheduling (FPS) with Rate Monotonic (RMPO) Deadline Monotonic Priority Ordering (DMPO)
  - Earliest Deadline First (EDF)

• **Real-world extensions**
  - Aperiodic, sporadic, soft real-time tasks
  - Deadlines shorter than period
  - Cooperative and deferred pre-emption scheduling
  - Fault tolerance in terms of exception handling considerations
  - Synchronized talks (priority inheritance, priority ceiling protocols)

• **Language support**
  - Ada95, POSIX static, off-line analysis mostly — RT-Java on-line, dynamic scheduling
Resource control

Uwe R. Zimmer – The Australian National University
References for this chapter

[Ada95RM] (link to on-line version)
Ada Working Group
ISO/IEC JTC1/SC 22/WG 9
Ada 95 Reference Manual
– Language and Standard Libraries

[Bloom79]
Toby Bloom
Evaluating synchronization mechanisms
Proceedings of the seventh ACM Symposium on Operating systems principles, 1979

[Burns01]
Alan Burns and Andy Wellings
Real-Time Systems and Programming Languages
Addison Wesley, third edition, 2001

[Mercer97]
Clifford W. Mercer
Operating system resource reservation for real-time and multimedia applications

[Murthy2001]
C. Siva Ram Murthy, G. Manimaran
Resource Management in Real-time Systems and Networks

all references and links are available on the course page
Resource control

Topics in real-time resource control

... from synchronization primitives and schedulers to resource management:

• Toby Bloom’s evaluation criteria for synchronization primitives

• Resource atomicity, liveliness, and double interaction

• Resource reclaiming (C. Siva Ram Murthy, G. Manimaran)

• Resource reservation schemes (Clifford W. Mercer)

(not covered here: general dead-lock prevention / avoidance / detection / recovery algorithms
☞ operating systems course)
Evaluating synchronization mechanisms

Categorizing resource/service requests
(based on Toby Bloom)

Service requests can be categorized by:

- their **type**
  (read requests might be treated very differently from update requests)

- their **time** (often: by their **order** or **relative time** only)

- their **attributes**, **parameters**, and the **priority** of the calling process
  (this includes **timing constraints**)

- the **synchronization state** of the resource
  (states which refer to the synchronisation aspect – including **timing constraints**)

- the **internal state** of the resource
  (states which refer to the actual contents and available resources– including **timing constraints**)
Evaluating synchronization mechanisms

Categorizing resource synchronization methods
(based on Toby Bloom)

Two (contradicting?) criteria:

Expressive power

☞ are all (required) forms of synchronization available?
☞ can all timing requirements be expressed?

Ease of use

☞ how error-prone are the constructs?
☞ how easy can basic methods be combined to complex resource control systems?
Evaluating synchronization mechanisms

Accepting or Avoiding?

Requests which cannot be fulfilled right now, can be handled via

Conditional wait

- accept all calls and suspend the threads internally
  - all threads are immediately inside the synchronized server
  - client threads are released from the server, only when the request is completed (can be overcome)

Avoidance synchronisation

- suspend tasks on the level of guards
  - all threads are ‘at the borders’ of the synchronized server
  - threads can easily revoke their requests
Evaluating synchronization mechanisms

Handling resource requests

Required features:

- Handling request types by priorities ✔ (Ada95, Occam2)
- Handling threads by priorities ✔ (most rt-systems)
- Handling threads in order or by their timing constraints ✔ (most systems) ✔ (Real-time Java)
- Handling requests by client-attributes ✅ (mostly: call needs to be accepted first)
- Handling requests by server state ✔ (Ada95, Occam2)
Evaluating synchronization mechanisms

Handling requests by types

```
WHILE TRUE
   PRI ALT
   ALT i=0 FOR max
      update [i] ? object
   ALT j=0 FOR max
      modify [j] ? object

pragma Queuing_Policy (Priority_Queing);
protected Resource_Manager is
   entry Update (...);
   entry Modify (...);
end Resource_Manager;

☞ serves clients with higher priority first
☞ serves entries in order of declaration
```
Evaluating synchronization mechanisms

Handling requests by types

WHILE TRUE
PRI ALT
ALT i=0 FOR max
  update [i] ? object
ALT j=0 FOR max
  modify [j] ? object

pragma Queuing_Policy (FIFO_Queing);

protected Resource_Manager is
  entry Update (…) when … is …
  entry Modify (…) when …
end Resource_Manager;

protected body Resource_Manager is
  entry Update (…) when … is …
  entry Modify (…) when …
    and Update’Count = 0 is …
end Resource_Manager;

☞ serves entries in defined order
☞ serves clients in FIFO-order (disregarding priorities)

how to control the order of requests regardless of their types?
how to control permission depending on call-parameters?
Evaluating synchronization mechanisms

Handling requests by parameters

protected body resource_control is

entry allocate(size : instances_of_resource)
  when resources_free >= size is
begin
  resource_free := resource_free - size;
end allocate;

procedure free(size : instances_of_resource) is
begin
  resource_free := resource_free + size;
end free;

end resource_control;

☞ ‘SR’ [Andrews and Olsson 1993] allows for such an direct access
☞ in most other synchronization environments: accept all and then conditional wait or requeue

NOT VALID in ADA!
Handling requests by parameters (using wrappers)

package Resource_Manager is

  Max_Resources : constant Integer := 100;
  type Resource_Range is new Integer range 1..Max_Resources;
  subtype Instances_Of_Resource is Resource_Range range 1..50;

  procedure Allocate (Size : Instances_Of_Resource);
  procedure Free   (Size : Instances_Of_Resource);

end Resource_Manager;

package body Resource_Manager is

  task Manager is
    entry Sign_In  (Size : Instances_Of_Resource);
    entry Allocate (Instances_Of_Resource);
    entry Free     (Size : Instances_Of_Resource);
  end Manager;

  procedure Allocate (Size : Instances_Of_Resource) is begin
    Manager.Sign_In  (Size);
    Manager.Allocate (Size);
  end Allocate;

  procedure Free   (Size : Instances_Of_Resource) is begin
    Manager.Free     (Size);
  end Free;

  Manager is informed about the request attributes first

  entry family

  double interaction is hidden
package Resource_Manager is

   Max_Resources : constant Integer := 100;
   type Resource_Range is new Integer range 1..Max_Resources;
   subtype Instances_Of_Resource is Resource_Range range 1..50;

   procedure Allocate (Size : Instances_Of_Resource);
   procedure Free    (Size : Instances_Of_Resource);

end Resource_Manager;

package body Resource_Manager is

   task Manager is
      entry Sign_In  (Size : Instances_Of_Resource);
      entry Allocate (Instances_Of_Resource);
      entry Free     (Size : Instances_Of_Resource);
   end Manager;

   procedure Allocate (Size : Instances_Of_Resource) is begin
     Manager.Sign_In  (Size);
     Manager.Allocate (Size);
   end Allocate;

   procedure Free    (Size : Instances_Of_Resource) is begin
     Manager.Free     (Size);
   end Free;

Manager can apply any policy to accept the ‘Allocate’ entries

entry family

double interaction is hidden
Evaluating synchronization mechanisms

Handling requests by types, attributes, and in a global order

Lack of expressive power (e.g. in Ada95) may lead to:

☞ Double Interactions

e.g. register all requests first, then serve the individual types in a global order

e.g. announce the parameters first, then serve the individual types based in parameters

☞ Requests are no longer atomic!

☞ Server deadlocked,
  when wrongly assuming that the client is going to make the second call

☞ Client deadlocked,
  when wrongly assuming that the client died and is not going to make the second call
Evaluating synchronization mechanisms

Handling requests by types, attributes, and in a global order

Lack of expressive power (e.g. in Ada95) may lead to:

☞ Double Interactions

Ways out:

- Define the double interaction by means of atomic actions and make this known to the underlying synchronization methods.
- Assume that the client will never die during a double interaction sequence
- Eliminate the double interaction by means of a attributed, single request type and requeuing
Evaluating synchronization mechanisms

Handling requests by types, attributes, and in a global order

type Request_Kinds     is (Allocate_Req, Expand_Req, Free_Req);
type Resource_Range    is ... 
type Resource_Range_Groups  is (small, medium, large);

protected Resource_Control is

    entry  Resource_Request  (Kind : Request_Kinds; Amount : Resource_Range);

private

    entry Allocate_Sign_In  (Amount : Resource_Range);
    entry Allocate          (Resource_Range_Groups);
    entry Expand_Sign_In    (Amount : Resource_Range);
    entry Expand            (Resource_Range_Groups);
    entry Free              (Amount : Resource_Range);

end Resource_Control;

☞ Server has full control over the types, parameters, and orders
Evaluating synchronization mechanisms

Handling requests by types, attributes, and in a global order

type Request_Kinds is (Allocate_Req, Expand_Req, Free_Req);
type Resource_Range is ... type Resource_Range_Groups is (small, medium, large);
protected Resource_Control is
  entry Resource_Request (Kind : Request_Kinds; Amount : Resource_Range);
private
  entry Allocate_Sign_In (Amount : Resource_Range);
  entry Allocate (Resource_Range_Groups);
  entry Expand_Sign_In (Amount : Resource_Range);
  entry Expand (Resource_Range_Groups);
  entry Free (Amount : Resource_Range);
end Resource_Control;

☞ Server has full control over the types, parameters, and orders

The clients are providing all information
Evaluating synchronization mechanisms

Handling requests by types, attributes, and in a global order

type Request_Kinds is (Allocate_Req, Expand_Req, Free_Req);
type Resource_Range is …
type Resource_Range_Groups is (small, medium, 
protected Resource_Control is
  entry Resource_Request (Kind : Request_Kinds; Amount : Resource_Range);
private
  entry Allocate_Sign_In (Amount : Resource_Range);
  entry Allocate (Resource_Range_Groups);
  entry Expand_Sign_In (Amount : Resource_Range);
  entry Expand (Resource_Range_Groups);
  entry Free (Amount : Resource_Range);
end Resource_Control;

☞ Server has full control over the types, parameters, and orders

The clients are providing all information

The protected object is arranging the suspending queues accordingly (requeue-facility)
Evaluating synchronization mechanisms

Handling requests by types, attributes, and in a global order

type Request_Kinds is (Allocate_Req, Expand_Req, Free_Req);
type Resource_Range is …
type Resource_Range_Groups is (small, medium, large);

protected Resource_Control is

  entry Resource_Request (Kind : Request_Kinds; Amount : Resource_Range);

private

  entry Allocate_Sign_In  (Amount : Resource_Range);
  entry Allocate          (Resource_Range_Groups);
  entry Expand_Sign_In    (Amount : Resource_Range);
  entry Expand            (Resource_Range_Groups);
  entry Free              (Amount : Resource_Range);

end Resource_Control;

☞ Server has full control over the types, parameters, and orders

Is the client going to lose all control?
Evaluating synchronization mechanisms

Handling requests by types and in a global order

`requeue with abort`

With a standard `requeue` statement:

- any outstanding timeout is cancelled
- the thread is no longer abortable

☞ clients losing control stemming from an ATC statement, or a timed entry-call
☞ the server can rely on the client thread no being revoked.

With a `requeue with abort` statement:

- all timeouts are maintained
- allows the client to still revoke the call
 ☞ maintains client side control

requeue can also lead to external entries!
☞ aborts need to be considered carefully
Evaluating synchronization mechanisms

Categorizing resource/service requests

(based on Toby Bloom)

Service requests can be categorized by:

- their type
- their time (often: by their order or relative time only)
- their attributes, parameters, and the priority of the calling process
- the synchronization state of the resource
- the internal state of the resource

The real-time perspective:

- take special care of failing tasks (atomic actions, deadlocks)
- determine and handle timing constraints in resource requests
Resource Reclaiming [Murthy2001]

Motivation for resource reclaiming

1. Worst case assumptions give schedulable systems, but might leave only a few spare resources.
2. Resources might not be actually used at run-time.
3. Some aspects of reliability in a real-time system rely directly on the amount of spare resources.

Resource reclaiming may enhance the system’s reliability
Resource Reclaiming [Murthy2001]

Resource reclaiming properties

- **Correctness:**
  - maintain the feasibility!

- **Inexpensiveness:**
  - resource reclaiming overhead need to be small in comparison to the possible gains

- **Bounded complexity:**
  - resource reclaiming should be included in the task’s worst case computation time
  - complexity needs to be bound by a constant

- **Effectiveness:**
  - improve the system’s actual reliability,
  - thus e.g. more failures can be handled by applying resource reclaiming
Resource Reclaiming [Murthy2001]

Expanded task-model

Each task $t_i$ has the following attributes:

- $T_i$: cycle time
- $E_i$: ready time
- $D_i$: deadline
- $C_i$: worst case computation time
- $C'_i$: actual computation time
- $R_i$: worst case response time
- a set of resource conflicts: $t_i \otimes t_j$, i.e. $t_i$ or $t_j$ requires a resource exclusively.
- a set of precedence constraints: $t_i < t_j$, i.e. $t_i$ completes always before $t_j$ may start.

Further assumptions:

- $n$ processors available
- tasks cannot migrate
- at most one task per processor
- task-queues are in shared memory
- tasks are not pre-empted
More terminology

- **Feasible (prerun) schedule** $S$:
  taking into account timing, resource, precedence constraints, and worst case computation times.

- **Postrun schedule** $S'$:
  starting from $S$ and considering the actual computation times into account.

- **Start and finish times**:
  the scheduled start $st_i$ and finish times $ft_i$ as from the feasible prerun schedule $S$,
  and the actual start $st_i'$ and finish times $ft_i'$ as depicted in the postrun schedule $S'$ of the task $t_i$.

- **Correct postrun schedule**:
  a postrun schedule is considered correct iff $\forall t_i \in Q: (st_i' \leq st_i) \land (ft_i' \leq d_i)$.

- **Passing tasks**:
  a task $t_j$ passed a task $t_i$ iff $st_j' < st_i' \land ft_j < st_i$, i.e. the strict order in $S$ is not maintained.
Resource Reclaiming \cite{Murthy2001}

Resource reclaiming algorithms

Two extreme versions:

- Dispatching according to the feasible prerun schedule $S$, i.e. no reclaiming at all – resource reclaiming cost is zero.
- Global re-scheduling, whenever reclaiming is requested, or at each release of a resource, i.e. optimal reclaiming
  – can be applied only, if the reclaiming cost is smaller than the gained resources

Optimal scheduling of dynamically arriving non-pre-emptive tasks on a multi-processor environment

\(\text{NP-hard}\)

All practical re-scheduling algorithms are approximating. The come in two classes:

- Algorithms without passing \(\text{bounded complexity}\)
- Algorithms with passing \(\text{in general: } O(\log n), \text{ but bounded with restricted passing}\)
Resource reclaiming from independent tasks

trivial: apply a greedy strategy, which dispatches tasks, whenever there are runnable tasks.

Prerun schedule $S$

- Feasible prerun schedule $S$
Resource reclaiming from independent tasks

- trivial: apply a greedy strategy, which dispatches tasks, whenever there are runnable tasks.

Postrun schedule $S'$

- Postrun schedule $S'$ without resource reclaiming
Resource reclaiming from independent tasks

- trivial: apply a greedy strategy, which dispatches tasks, whenever there are runnable tasks.

Reclaimed resources

- Postrun schedule $S'$ with resource reclaiming for independent tasks
Resource reclaiming from interdependent tasks

- greedy reclaiming

Postrun schedule $S'$

- Postrun schedule $S'$ without resource reclaiming
- Tasks $t_4 \otimes t_7; t_4 \otimes t_9; t_7 \otimes t_9; t_7 \otimes t_{12}; t_9 \otimes t_{12}$ have conflicting resource locks
Resource reclaiming from interdependent tasks

- Greedy reclaiming

Runtime anomaly

- Postrun schedule $S'$ without greedy resource reclaiming
- Tasks $t_4 \otimes t_7$; $t_4 \otimes t_9$; $t_7 \otimes t_9$; $t_7 \otimes t_{12}$; $t_9 \otimes t_{12}$ have conflicting resource requests
Resource reclaiming from interdependent tasks

- Basic reclaiming: look for simultaneous idling

Postrun schedule $S'$

- Postrun schedule $S'$ without resource reclaiming
- Tasks $t_4 \otimes t_7; t_4 \otimes t_9; t_7 \otimes t_9; t_7 \otimes t_{12}; t_9 \otimes t_{12}$ have conflicting resource locks
Resource reclaiming from interdependent tasks

Basic reclaiming

- Postrun schedule $S'$ without basic resource reclaiming
- Tasks $t_4 \otimes t_7; t_4 \otimes t_9; t_7 \otimes t_9; t_7 \otimes t_{12}; t_9 \otimes t_{12}$ have conflicting resource locks
Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

- early start algorithm

- Detect overlaps in the prerun schedule $S$:
  
  \[
  t_{<i} = \{ t_j | ft_j < st_i \} \\
  t_{>i} = \{ t_j | st_j > ft_i \} \\
  t_{\sim i} = \{ t_j | ((t_j \not\in t_{<i}) \land (t_j \not\in t_{>i})) \} \quad \text{all tasks which overlap with } t_i \text{ in } S
  \]

- Detect tasks overlapping with $t_i$ on processor $k$ and order all sets

- Allow tasks in $t_{\sim i}$ to be executed simultaneously and ensure that they do not overlap with tasks out of $t_{<i}$ or $t_{>i}$.

- Complexity $O(m^2)$; with $m$ processors.
Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

- early start algorithm

Postrun schedule \( S' \)

- Postrun schedule \( S' \) without resource reclaiming
- Tasks \( t_4 \otimes t_7; t_4 \otimes t_9; t_7 \otimes t_9; t_7 \otimes t_{12}; t_9 \otimes t_{12} \) have conflicting resource locks
Resource reclaiming from interdependent tasks

- Early start algorithm

Postrun schedule $S'$ without early start resource reclaiming

- Tasks $t_4 \otimes t_7; t_4 \otimes t_9; t_7 \otimes t_9; t_7 \otimes t_{12}; t_9 \otimes t_{12}$ have conflicting resource locks
Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

- Restriction vectors (RV) algorithm (Shen, Ramamritham, Stankovic, ‘93)

- **Restriction vector (RV):**

  \[
  RV_{i}[j] = \begin{cases} 
  t_k \in t_{<i}(j) | (\neg \exists t \in t_{<i}(j) | st > st_k) & \text{if } j = \text{proc}(i) \\
  t_m \in t_{<i}(j) | (t_m < t_i \lor t_m \otimes t_i) \land (\neg \exists t \in t_{<i}(j) | (st > st_m) \land (t_l < t_i \lor t_l \otimes t_i)) & \text{if } j \neq \text{proc}(i) \\
  - & \text{no such task}
  \end{cases}
  \]

- **Completion bit matrix (CBM):**

  \[
  CMB[i, j] = \begin{cases} 
  1 & \text{iff task } t_i \text{ has completed its scheduled execution in processor } j \\
  0 & \text{otherwise}
  \end{cases}
  \]
Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

- Restriction vectors (RV) algorithm (Shen, Ramamritham, Stankovic, ‘93)

1. compute the $RV_i(j)$ by checking the $k$ most recent tasks in $t_{<i}(j)$
2. for any task $t_i$ next to be scheduled on processor $j$:
   - fetch the most recent $CBM$
   - if $\forall t_j \in RV_i(j) | CBM(i, j) = 1$ then start $t_i$ else idle until the next $CBM$ update.

The algorithm is heuristic in the sense that it is only checking the $k$ most recent tasks in $t_{<i}(j)$

The complexity is $O(m^2)$ since $m$ processors need to check $m$ RV-entries bounded
Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

- Restriction vectors (RV) algorithm (Shen, Ramamritham, Stankovic, ‘93)

Prerun schedule $S$

- Tasks $t_4 \otimes t_7$; $t_4 \otimes t_9$; $t_7 \otimes t_9$; $t_7 \otimes t_{12}$; $t_9 \otimes t_{12}$ have conflicting resource locks.

- Tasks $t_{10} < t_8$; $t_{10} < t_4$; $t_8 < t_9$; $t_8 < t_{13}$; $t_1 < t_2$; $t_1 < t_3$; $t_2 < t_{12}$; $t_3 < t_{12}$; $t_{11} < t_{12}$ have precedence relations.
Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

Restriction vectors (RV) algorithm (Shen, Ramamritham, Stankovic, ‘93)

Prerun schedule $S$

$RVs:\ t_1: [-, -, -]; \quad t_2: [t_1, -, -]; \quad t_3: [t_2, -, -]; \quad t_4: [t_3, t_7, t_{10}]; \quad t_5: [t_4, -, -];$

$\quad t_6: [-, -, -]; \quad t_7: [-, t_6, -]; \quad t_8: [-, t_7, t_{10}]; \quad t_9: [t_4, t_8, t_{12}]; \quad t_{10}: [-, -, -];$

$\quad t_{11}: [-, -, t_{10}]; \quad t_{12}: [t_3, t_7, t_{11}]; \quad t_{13}: [-, t_8, -]$
Resource reclaiming from interdependent tasks

- Restriction vectors (RV) algorithm (Shen, Ramamritham, Stankovic, ‘93)

Postrun schedule $S'$

- RVs: $t_1: [-, -,-]$; $t_2: [t_1, -,-]$; $t_3: [t_2, -,-]$; $t_4: [t_3, t_7, t_{10}]$; $t_5: [t_4, -,-]$; $t_6: [-, -,-]$; $t_7: [-, t_6,-]$; $t_8: [-, t_7, t_{10}]$; $t_9: [t_4, t_8, t_{12}]$; $t_{10}: [-,-,-]$; $t_{11}: [-, -, t_{10}]$; $t_{12}: [t_3, t_7, t_{11}]$; $t_{13}: [-, t_8,-]$
Resource Reclaiming [Murthy2001]

Resource reclamation from interdependent tasks

Restriction vectors (RV) algorithm (Shen, Ramamritham, Stankovic, ’93)

Restriction vector based resource reclaiming

\[
\begin{align*}
\text{RVs:} & \quad t_1: [-, -, -]; \quad t_2: [t_1, -, -]; \quad t_3: [t_2, -, -]; \quad t_4: [t_3, t_7, t_{10}]; \quad t_5: [t_4, -, -]; \\
& \quad t_6: [-, -, -]; \quad t_7: [-, t_6, -]; \quad t_8: [-, t_7, t_{10}]; \quad t_9: [t_4, t_8, t_{12}]; \quad t_{10}: [-, -, -]; \\
& \quad t_{11}: [-, -, t_{10}]; \quad t_{12}: [t_3, t_7, t_{11}]; \quad t_{13}: [-, t_8, -]
\end{align*}
\]
Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

Restriction vectors (RV) algorithm (Shen, Ramamritham, Stankovic, ‘93)

Proof of Correctness

**Lemma:** Given a feasible prerun schedule $S$: if $\exists t_i \mid (st_i' > st_i)$ then passing must have occurred.

**Proof:** Assuming that no passing occurred,
then all $t \in t_{<i}$ have been dispatched before $t_i$ and all $t \in t_{>i}$ are only dispatched after $t_i$ completed. By definition of a feasible schedule all $t \in t_{\sim i}$ do not interfere with $t_i$ and can thus by no means delay the execution of $t_i$.
Therefore $st_i' \leq st_i$. \(\checkmark\)
Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

Restriction vectors (RV) algorithm (Shen, Ramamritham, Stankovic, ‘93)

Proof of Correctness

**Theorem:** The RV-algorithm gives a correct postrun schedule $S'$.

**Proof:** By the above lemma, passing occurred if $S'$ is incorrect, i.e.

$$\exists t_i, t_j \left( st_j > ft_i \right) \land \left( st_j' < st_i' \right) \land \left( st_i' > st_i \right).$$

Two cases need to be distinguished:

- **case 1:** $t_i$ and $t_j$ have resource or precedence conflicts, then $t_i$ is directly or transitively included in the restriction vector $RV_j$. Therefore this case of passing is prevented by the RV-algorithm.
- **case 2:** $t_i$ and $t_j$ have no resource or precedence conflicts. In this case $t_j$ cannot delay the execution of $t_i$ by means of passing and the postrun schedule $S'$ would be correct still.

Therefore the RV-algorithm allows for restricted forms of passing only, which does not corrupt the correctness of the postrun schedule $S'$. 
Resource Reclaiming \[\text{[Murthy2001]}\]

Resource reclaiming from interdependent tasks

- Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, ‘97)

- **Restriction vector (RV) with static processor assignment:**

\[
RV_i[j] = \begin{cases} 
  t_k \in t_{<i}(j) \setminus (\exists t_l \in t_{<i}(j) | st_l > st_k) & \text{if } j = \text{proc}(i) \\
  t_m \in t_{<i}(j) \setminus (t_m < t_i \lor t_m \otimes t_i) \land (\exists t_l \in t_{<i}(j) | (st_l > st_m) \land (t_l < t_i \lor t_l \otimes t_i)) & \text{if } j \neq \text{proc}(i) \\
  \_ & \text{no such task}
\end{cases}
\]

- **Restriction vector (RV) with dynamic processor assignment:**

\[
RV_i[j] = \begin{cases} 
  t_m \in t_{<i}(j) \setminus (t_m < t_i \lor t_m \otimes t_i) \land (\exists t_l \in t_{<i}(j) | (st_l > st_m) \land (t_l < t_i \lor t_l \otimes t_i)) & \text{if } t_m \text{ exists} \\
  \_ & \text{no such task}
\end{cases}
\]
Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

- Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, ‘97)

Prerun schedule $S$

- **RVs:**
  - $t_1: [-, -, -];$
  - $t_2: [t_1, -, -];$
  - $t_3: [t_1, -, -];$
  - $t_4: [-, t_7, t_{10}];$
  - $t_5: [-, -, -];$
  - $t_6: [-, -, -];$
  - $t_7: [-, -, -];$
  - $t_8: [-, -, t_{10}];$
  - $t_9: [t_4, t_8, t_{12}];$
  - $t_{10}: [-, -, -];$
  - $t_{11}: [-, -, -];$
  - $t_{12}: [t_3, t_7, t_{11}];$
  - $t_{13}: [-, t_8, -]$
Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

- Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, ‘97)

Prerun schedule $S$

$RVs:\; t_1: [-,-,-];\; t_2: [t_1,-,-];\; t_3: [t_1,-,-];\; t_4: [-,t_7,t_{10}];\; t_5: [-,-,-];$
$\; t_6: [-,-,-];\; t_7: [-,-,-];\; t_8: [-,-,t_{10}];\; t_9: [t_4,t_8,t_{12}];\; t_{10}: [-,-,-];$
$\; t_{11}: [-,-,-];\; t_{12}: [t_3,t_7,t_{11}];\; t_{13}: [-,t_8,-]$
Resource reclaiming from interdependent tasks

Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, ‘97)

Postrun schedule $S'$ with task migration

$\textbf{RVs:}$

$t_1: [-, -, -]$; $t_2: [t_1, -, -]$; $t_3: [t_1, -, -]$; $t_4: [-, t_7, t_{10}]$; $t_5: [-, -, -]$;
$t_6: [-, -, -]$; $t_7: [-, -, -]$; $t_8: [-, -, t_{10}]$; $t_9: [t_4, t_8, t_{12}]$; $t_{10}: [-, -, -]$;
$t_{11}: [-, -, -]$; $t_{12}: [t_3, t_7, t_{11}]$; $t_{13}: [-, t_8, -]$
Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

- Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, ‘97)

Postrun schedule $S'$ with task migration

RVs: $t_1$: [-, -, -]; $t_2$: [$t_1$, -, -]; $t_3$: [$t_1$, -, -]; $t_4$: [-, $t_7$, $t_{10}$]; $t_5$: [-, -, -];
$t_6$: [-, -, -]; $t_7$: [-, -, -]; $t_8$: [-, -, $t_{10}$]; $t_9$: [$t_4$, $t_8$, $t_{12}$]; $t_{10}$: [-, -, -];
$t_{11}$: [-, -, -]; $t_{12}$: [$t_3$, $t_7$, $t_{11}$]; $t_{13}$: [-, $t_8$, -]
Resource reclaiming from interdependent tasks

- Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, ‘97)

Postrun schedule $S'$ with task migration

RVs: $t_1: [-, -, -];$ $t_2: [t_1, -, -];$ $t_3: [t_1, -, -];$ $t_4: [-, t_7, t_{10}];$ $t_5: [-, -, -];$

$t_6: [-, -, -];$ $t_7: [-, -, -];$ $t_8: [-, -, t_{10}];$ $t_9: [t_4, t_8, t_{12}];$ $t_{10}: [-, -, -];$

$t_{11}: [-, -, -];$ $t_{12}: [t_3, t_7, t_{11}];$ $t_{13}: [-, t_8, -]$
Resource reclaiming from interdependent tasks

- Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, ‘97)

Postrun schedule $S'$ with task migration

**RVs:**

- $t_1: [\_, \_, \_]$
- $t_2: [t_1, \_, \_]$
- $t_3: [t_1, \_, \_]$
- $t_4: [\_, t_7, t_{10}]$
- $t_5: [\_, \_, \_]$
- $t_6: [\_, \_, \_]$
- $t_7: [\_, \_, \_]$
- $t_8: [\_, \_, t_{10}]$
- $t_9: [t_4, t_8, t_{12}]$
- $t_{10}: [\_, \_, \_]$
- $t_{11}: [\_, \_, \_]$
- $t_{12}: [t_3, t_7, t_{11}]$
- $t_{13}: [\_, t_8, \_]$
Resource reclaiming from interdependent tasks

Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, ‘97)

Postrun schedule \( S' \) with task migration

\[ RVs: \quad t_1: [-, -]; \quad t_2: [t_1, -]; \quad t_3: [t_1, -]; \quad t_4: [-, t_7, t_{10}]; \quad t_5: [-, -]; \]
\[ t_6: [-, -]; \quad t_7: [-, -]; \quad t_8: [-, t_{10}]; \quad t_9: [t_4, t_8, t_{12}]; \quad t_{10}: [-, -]; \]
\[ t_{11}: [-, -]; \quad t_{12}: [t_3, t_7, t_{11}]; \quad t_{13}: [-, t_8, -] \]
Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

- Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, ‘97)

Postrun schedule $S'$ with task migration

RVs:
- $t_1 : [-, -,-]$;
- $t_2 : [t_1,-,-]$;
- $t_3 : [t_1,-,-]$;
- $t_4 : [-, t_7, t_{10}]$;
- $t_5 : [-,-,-]$;
- $t_6 : [-,-,-]$;
- $t_7 : [-,-,-]$;
- $t_8 : [-,-,t_{10}]$;
- $t_9 : [t_4, t_8, t_{12}]$;
- $t_{10} : [-,-,-]$;
- $t_{11} : [-,-,-]$;
- $t_{12} : [t_3, t_7, t_{11}]$;
- $t_{13} : [-, t_8,-]$
Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

 Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, ‘97)

Postrun schedule $S'$ with task migration

$RVs: \quad t_1: [-, -, -]; \quad t_2: [t_1, -, -]; \quad t_3: [t_1, -, -]; \quad t_4: [-, t_7, t_{10}]; \quad t_5: [-, -, -];$

$\quad t_6: [-, -, -]; \quad t_7: [-, -, -]; \quad t_8: [-, -, t_{10}]; \quad t_9: [t_4, t_8, t_{12}]; \quad t_{10}: [-, -, -];$

$\quad t_{11}: [-, -, -]; \quad t_{12}: [t_3, t_7, t_{11}]; \quad t_{13}: [-, t_8, -]$
Resource reclaiming from interdependent tasks

Resource Reclaiming [Murthy2001]

Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, ’97)

Correctness of the migration process

To ensure that the swapping of dispatching queues $DQ_x \leftrightarrow DQ_y$ between processor $P_x$ and $P_y$ does not interfere with the correctness of the postrun schedule $S'$, swapping is permitted only if:

$$st_i \geq ft_j$$

the currently blocked task $t_i$ is not further delayed

(where task $t_i$ is next to be scheduled on the idling $P_x$ and task $t_j$ is currently executing on $P_y$).

The unrestricted and executable task $t_k$, which is next to be scheduled on $P_y$ is started earlier by transferring it to the idling $P_x$.

no task is delayed by swapping these dispatching queues.
Resource Reclaiming [Murthy2001]

Resource reclaiming evaluated

Some additional observables:

- **Task graph density** \( P_p \rightarrow [0...1] \), where zero indicates an independent and one a fully dependent task-set.

- **aw-ratio**: \( C_i' / C_i \) (actual to worst case ratio)

- **mig-attempts**: number of checks on dispatch queues by the RV with migration algorithm

RC computational costs (from Manimaran, Murthy, Vijay, Ramamritham ‘97):

- \( C_{RC-basic} = 1 \)

- \( C_{RC-early-start} = m C_{RC-basic} \); with \( m \) the number of processors

- \( C_{RC-RV} = C_{RC-early-start} + C_{RV} \); with \( C_{RV} \) the cost for the calculation of the RVs.

- \( C_{RC-RV-migration} = C_{RC-RV} + f(mig-attempts, C_{RC-early-start}) \)
Resource Reclaiming [Murthy2001]

Resource reclaiming evaluated

Practical measurements:

• There is a continuous improvement in terms of gained resources by applying: basic—→ early-start—→ RV-reclaiming—→ RV-reclaiming-with-task-migration—algorithms.

• In case of RV reclaiming with task migration, the extended communication/synchronization overhead can reach noticeable levels.

• There need to be a high degree of dependencies in the task-set ($P^r_p$), in order to justify the application of RV reclaiming with task migration.

Reclaiming in the introduced sense is applicable only to real-time systems which:

• allow for earlier task start times

• allow for task migration

• and where all dependencies can be expressed in terms of the introduced formalism
Real-time Resource Control [Mercer97]

Issues

Policies:

- **Priority assignment problem**
  - the mapping of the known and arising timing constraints and reliability considerations to linear priorities.

- **Overload problem**
  - predicting and protecting the system from overload conditions.

- **Flexibility problem**
  - locally adjusting the system behaviour to the current timing constraints.

Run-time environment:

- **Enforcement problem**
  - handling tasks and resources which exceeds their anticipated worst case limits.

- **Measurement problem**
  - recording all relevant information in a sufficient resolution and frequency.

- **Coordination problem**
  - synchronizing system-components which are organized according to different policies.
Summary

Resource control

• Resource synchronization primitives
  • evaluation criteria for resource synchronisation methods
  • atomicity, liveliness, and double interaction

• Resource reclaiming schemes
  • basic reclaiming, early start, and restriction vector algorithms
  • resource reclaiming with task migration

• Real-time resource control
  • policy and run-time issues to be considered
Reliability

Uwe R. Zimmer – The Australian National University
References for this chapter

[Burns98] Alan Burns, Brian Dobbing, George Romanski
The Ravenscar Tasking Profile for High Integrity Real-Time Programs

[Burns01] Alan Burns and Andy Wellings
Real-Time Systems and Programming Languages
Addison Wesley, third edition, 2001

Assuring Design Diversity in N-Version Software:
A Design Paradigm for N-Version Programming

Axioms for Real-Time Logics
Lecture Notes in Computer Science 1466, Springer-Verlag, 1999, pp. 219-236

all references and links are available on the course page
Based on a set of powerful and diverse tools …

… reconsidering the basic problems of:

- system identification / analysis
- fault prevention
- error detection
- fault tolerance

... building predictable / dependable systems …

... in a real-time domain!
Reliability

Terminology

Reliability ::= measure of success with which a system conforms to its specification or low failure rate.

Failure ::= deviation of a system from its specification
Error ::= system state which lead to failures
Fault ::= the reason for an error
Reliability

Faults on different levels

- Inconsistent or inadequate specification  
  very frequent source for disastrous faults

- Software design errors  
  very frequent source for disastrous faults

- Component & communication system failures  
  rare and mostly predictable
Reliability

Faults in the time domain

- Transient faults
  - Many communication system failures, electric interference, etc.
- Intermittent faults
  - Transient errors which occur more than once (e.g. overheating effects)
- Permanent faults
  - Stay in the system until they are repaired by some means
Reliability

Observable failures states

Failure modes

Time domain

- too early
- too late
- never (omission)

Value domain

- fail never
- fail uncontrolled

Failures in value domain:

- Constraint error
- Value error

Failures in time domain:

- fail silent
- fail stop
- fail controlled
Reliability

Achieving reliability
Reliability

System identification

Investigate:

• static applications specifications
• physical sensors and converters constraints
• constraints of the employed controller network
• constraints of the underlying run-time system
• dynamic application specifications (requested real-time behaviour)

☞ Understanding all critical real-time requirements and issues
Fault avoidance at hardware-level:

- use reliable hardware components — consider the environmental demands!
- use an adequate (hardware) system design — shock, humidity, interference, …
- ensure proper assembly and encapsulation — weak connectors, bad connectors, …

Fault avoidance at software design level:

- strict system specifications (employ format methods if applicable)
- use proven software-engineering and design methodologies
- employ languages and run-time environments with reasonable support for the requirements.
Reliability

Fault removal

Find and remove errors from the previous stage.

☞ Team programming methods like extreme programming or rigorous testing may help here.

but ...

• no re-evaluation method indicates the absence of faults (even formal methods cannot identify specification faults)

… and specifically for real-time and embedded systems …:

• often: tests cannot be performed under realistic conditions … especially exceptional conditions
• most simulation environments have a severe impact on real-time systems
• the test space for real-time system is significantly larger than for non-real-time systems
Regardless of the rigor of fault prevention methods:

the real-time system might still fail

This is specifically critical for all non-monitored systems:

- systems which are (temporary) inaccessible
- un-manned vehicles which operate autonomously by default
- systems in remote / dangerous environments

Instead (or in addition to fault prevention): enabling a ‘safe landing’:  

Fault tolerance
Reliability

Fault tolerance

• Full fault tolerance
  the system continues to operate in the presence of ‘foreseeable’ error conditions without any significant failures — also this might induct a reduced operation period.

• Graceful degradation (fail soft)
  the system continues to operate in the presence of ‘foreseeable’ error conditions, accepting a partial loss of functionality or performance.

• Fail safe
  the system halts and maintains its integrity

☞ Full fault tolerance is not maintainable for an infinite operation time!
☞ Graceful degradation might have multiple levels of reduced functionality.
Real-Time & Embedded Systems

Reliability

Hardware redundancy

Adding extra hardware resources:

- for the detection of failures and the localization of faults
- for the handling of exceptional situations and error-recovery.
- as a functional duplication or multiplication of complete (sub-)systems in order to hot-swap or select the operational one in case of a failure in one part of the (sub-)system.

Fault-detection and recovery hardware includes:
- watch-dog timers, limit switches, additional physical sensors, transient-recording-systems (emergency system dump), overload-backup-systems, or even in-circuit emulators.

- Triple Modular Redundancy (TMR) or N-Modular Redundancy (NMR) assumes:
  functionally identical components which are either:
  - static parts of the system and connected via a voting/masking/comparing system
  - or in case of a detected error-condition: dynamic parts which are swapped in.
Reliability

Hardware redundancy

any hardware redundancy adds to the overall system complexity!

In case of TMR or NMR:

the assumption that an error occurs in one part of the system only requires that either:

• the fault is based on a physical phenomenon, which applies only locally
• or the structure of the functionally identical systems is sufficiently different

For some high-risk systems this approach is applied in forms of redundant sub-systems with:

• the same specification
• different computer systems (CPUs, buses, memory systems, drives)
• different operating systems
• different real-time languages and development environments (N-Version programming)
• and by restricting the communication between the different developer teams

not too surprisingly, the outputs from the different systems are slightly different …
Reliability

Triple Modular Redundancy
(example)

3 identical primary flight computers distributed in the Boeing 777, each consisting of:

- 3 processors: AMD 29050, Motorola 68040, INTEL 80486 (called ‘lanes’)
- independent power-sources and inertia measurements
- code built by 3 different Ada compilers
- the same Ada source code (‘the specification’): around 3 million lines of code, but different monitor functions

Targeted failure probability: $< 10^{-10} / h$ (e.g. UK Seizewell B nuclear reactor (emerg.): $< 10^{-3} / h$)

No single fault on board the 777 should occur without failure identification.
No single fault on board the 777 should cause more than the loss of one primary flight computer.

Sophisticated synchronization and communication systems.

(not a single fatal event — information from November 2001)
### N-version programming

Impacts to software diversity:

<table>
<thead>
<tr>
<th></th>
<th>Development teams</th>
<th>Languages</th>
<th>Tools</th>
<th>Algorithms</th>
<th>Methodologies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coding</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(■: highest – □: high – □: low – □: lowest) (source: [Lyu92])
"The six-language project"

Joint project between the UCLA (Dependable computing and fault-tolerance systems) and the Honeywell Commercial Flight Systems Division (1992)

- The specifications (about a flight controller) were original system description documents (SDD) by Honeywell enhanced by additional cross-checking points and included some enforced diversity elements (a 64-page document).
- The development teams were isolated and any technical discussions were strictly prohibited.
- All communication and documentation is requested to follow predefined protocols (written form) defined and handled by a coordinating team.
- Specified tests were performed by the coordinating team before a version was accepted for integration.
- The $N$-Version paradigm was applied to all stages of the development cycle.
Reliability

“The six-language project”
(source: [Lyu92])

<table>
<thead>
<tr>
<th>Language</th>
<th>Sources (l.o.c.)</th>
<th>Test runs</th>
<th>Errors</th>
<th>Failure-rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>2256</td>
<td>5127400</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>‘C’</td>
<td>1531</td>
<td>— “ —</td>
<td>568</td>
<td>1.108×10^{-4}</td>
</tr>
<tr>
<td>Modula-2</td>
<td>1562</td>
<td>— “ —</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pascal</td>
<td>2331</td>
<td>— “ —</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Prolog</td>
<td>2228</td>
<td>— “ —</td>
<td>680</td>
<td>1.326×10^{-4}</td>
</tr>
<tr>
<td>T (close to Lisp)</td>
<td>1568</td>
<td>— “ —</td>
<td>680</td>
<td>1.326×10^{-4}</td>
</tr>
<tr>
<td>Average</td>
<td>1913</td>
<td>— “ —</td>
<td>321</td>
<td>0.627×10^{-4}</td>
</tr>
</tbody>
</table>

© 2009 Uwe R. Zimmer, The Australian National University
## Reliability

**“The six-language project”**

*(source: [Lyu92])*

<table>
<thead>
<tr>
<th>Failure category</th>
<th>average single version failure probabilities (5127400 cases)</th>
<th>average 3-version failure probabilities (102548000 cases)</th>
<th>average 5-version failure probabilities (30764400 cases)</th>
</tr>
</thead>
<tbody>
<tr>
<td>no errors</td>
<td>0.99993733</td>
<td>.9998409</td>
<td>.9997807</td>
</tr>
<tr>
<td>single error</td>
<td>6.27×10&lt;sup&gt;-5&lt;/sup&gt;</td>
<td>13.05×10&lt;sup&gt;-5&lt;/sup&gt;</td>
<td>19.15×10&lt;sup&gt;-5&lt;/sup&gt;</td>
</tr>
<tr>
<td>two distinct errors</td>
<td>00.20×10&lt;sup&gt;-5&lt;/sup&gt;</td>
<td>00.23×10&lt;sup&gt;-5&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>two coincident errors</td>
<td>02.65×10&lt;sup&gt;-5&lt;/sup&gt;</td>
<td></td>
<td>02.21×10&lt;sup&gt;-5&lt;/sup&gt;</td>
</tr>
<tr>
<td>three errors</td>
<td></td>
<td></td>
<td>00.34×10&lt;sup&gt;-5&lt;/sup&gt;</td>
</tr>
</tbody>
</table>
Reliability

“The six-language project”
(source: [Lyu92])

☞ The resulting 3-version and 5-version systems displayed lower failure rates than a ‘golden master’ reference implementation by Honeywell.
☞ Coincident errors involving more than two versions were never observed.
☞ A total of 93 faults were detected.

☞ Control problems are specifically suitable for n-version programming, since the error-detection and synchronization algorithms are relatively simple.

In general: diverting results do not necessarily imply any faults.
Real-Time & Embedded Systems

Reliability

N-version programming – Voting issues

• **Integer arithmetic:**
  - Integer (or any discrete sub-type) -based results will be identical ⇒ ✔

• **Real arithmetic:**
  - Real-valued results will usually be different ⇒ Comparisons need to consider tolerances.
  - If the process is not fully continuous (thresholds, quantizations, bifurcations)
    ⇒ Comparisons need to re-model the whole process in order to evaluate similarities
    ⇒ Independence ✏️ re-specify the system

• **Multiple solutions:**
  - The solution space itself allows for multiple correct, but different solutions
    ⇒ ✏️ re-specify the system
Reliability

N-version programming

some issues:

• **Specification:**
  Assuming that a good part of software faults stem from wrong or incomplete specifications ☞ *N*-version programming will not help in this case

• **Diversity assumption:**
  Diversity can be enforced and supported in some areas (demonstrated by examples), while coincident error conditions can be observed in other application domains (also documented by case-studies). The rigorous identification of adequate domains for *N*-version programming is currently part of active research.

• **Project costs:**
  Since the development costs are increasing by a factor of *N* plus coordination costs, it needs to be considered carefully whether a single version developed with the same effort shows perhaps a similar level of reliability.
Real-Time & Embedded Systems

Reliability

Dynamic redundancy

Four constituent phases (Anderson and Lee, ‘90):

1. **Error detection**
   
   Detection of a precise error state is essential.

2. **Damage confinement and assessment**
   
   Diagnosis of the damage, which occurred between the fault and the detected error state.

3. **Error recovery**
   
   Sequence of operations leading from the detected error state to an operational state.

4. **Fault treatment**
   
   In order to prevent the same error state again, the fault itself might/should be eliminated.
Real-Time & Embedded Systems

Reliability

Dynamic redundancy — Error detection

- Error states from the environment
  - **Hardware** … CPU, controllers, communication systems, …
  - **Run-time environment**

- Error states stemming from checks without the application processes
  - **Replication** – employ N-version programming to detect error states
  - **Timing** – watchdog timers and overrun detectors
  - **Reversal** – apply the reverse function and compare $x \iff f^{-1}(f(x))$
  - **Coding** – detect corrupted data via redundant information (CRC-checks, …)
  - **Reasonableness** – check assertions (e.g. in Eiffel)
  - **Structural** – check structural integrity (e.g. lists, file-systems)
  - **Continuity** – assuming a limited difference between consecutive controller values.
  - …
Real-Time & Embedded Systems

Reliability

Dynamic redundancy — Damage diagnosis

Confinement:
☞ How to avoid the transfer of fault-effects between system parts?
  • Modular decomposition
  • Atomic actions
  • ‘Firewalls’

Assessment:
☞ resulting from the location of the detected error state and the possible paths through the system which are all leading to this error state.
☞ a fine-granular system structure (error-confinement) limits the length of these possible paths.

☞ a very well structured system is the cornerstone of damage diagnosis.
Reliability

Dynamic redundancy — Error recovery

Backward error recovery:
- set checkpoints and save the system state with each passing of a checkpoint. How can system-wide consistent checkpoints be ensured?
- if an error state is detected: set back to the last consistent checkpoint.
  ☞ applicable even if the fault itself cannot be identified.
  ☞ not applicable at all, if the system contains non-reversible or -resetable components (time, …)

Forward error recovery:
- method of choice for most time critical parts of real-time and embedded systems.
  ☞ highly application dependent.
  ☞ may involve complex mode and priority changes (deadlines might still be relevant).
Dynamic redundancy — Fault treatment

- Localization of a hardware fault is usually easier and more precise than of a software fault.
- On-line fault treatment might be tricky and is usually limited to (hot) exchanges of complete modules (software as well as hardware).
- Granularity is usually finer than in static redundant systems.
- Exchange of faulty components is nevertheless usually an expensive and complex operation.

☞ the number of substitutable sub-systems in a dynamic redundant system is still limited.

(many systems will assume transient faults, log the event and continue operations … )
Safety and Dependability

- **Safety**: freedom from those conditions that can cause death, injury, occupational illness, damage to (or loss of) equipment (or property), or environmental harm (Leveson, ‘86)

- **Dependability**:
  - Availability — ready to use
  - Reliability — absence of failures
  - Safety — absence of fatal failures
  - Confidentiality — absence of unauthorized disclosures
  - Integrity — no data corruptions
  - Maintainability — accessibility to changes and improvements
Reliability

... more reliability in the design process:

Restrict, Formalise, …?

Restrict:

- limit the tools and environments to ‘safer’ operations
- e.g. Esterel, High-Integrity Pearl, Ada95 Ravenscar profile, …

Formalise:

- UML (‘the object oriented approach’)
- Temporal logic, Real-Time Logic (RTL) as an extension of predicate logic
- classical real-time design methods: MASCOT, JSD, MOON, HOOD, HRT-HOOD, CODARTS, …
Real-Time & Embedded Systems

Reliability

Ada95 Ravenscar profile (Burns, Dobbing, Romanski ‘98)

• **Task type and object declarations at the library level**
  — no hierarchy of tasks, and hence no exit protocols needed from blocks and sub-programs.

• **No dynamic allocation or unchecked de-allocation of protected and task objects**
  — removes the need for dynamic objects.

• **Tasks are assumed to be non-terminating**
  — this is primarily because task termination is generally considered to be an error for a real-time program which is long-running and defines all of its tasks at start-up.

• **Library level Protected objects with no entries**
  — these provide atomic updates to shared data and can be implemented simply.

• **Library level Protected objects with a single entry**
  — used for invocation signalling; but removes the overheads of a complicated exit protocol.
Reliability

Ada95 Ravenscar profile (Burns, Dobbing, Romanski ‘98)

- **Barrier consisting of a single boolean variable**
  - no side effects are possible and exit protocol becomes simple.

- **Only a single task may queue on an entry** — hence no queue required; this is a static property that can easily be verified, or it can lead to a bounded error at runtime.

- **No requeue** — leads to complicated protocols, significant overheads and is difficult to analyse (both functionally and temporally).

- **No Abort or ATC** — these features leads to the greatest overhead in the run-time system due to the need to protect data structures against asynchronous task actions.

- **No use of the select statement** — non-deterministic behaviour is difficult to analyse, moreover the existence of protected objects has diminished the importance of the select statement to the tasking model.

- **No use of task entries** — not necessary to program systems that can be analysed; it follows that there is no need for the accept statement.
Reliability

Ada95 Ravenscar profile (Burns, Dobbing, Romanski ‘98)

- "Delay until" statement but no "delay" statement
  - the absolute form of delay is the correct one to use for constructing periodic tasks.
- "Real-Time" package — to gain access to the real-time clock.
- No Calendar package — "Real-Time" package is sufficient.
- Atomic and Volatile pragmas — needed to enforce the correct use of shared data.
- Count attribute (but not within entry barriers)
  - can be useful for some algorithms and has low overhead.
- Ada.Task_Identification — can be useful for some algorithms and has low overhead, available in reduced form (no Abort_Task or task attribute functions Callable or Terminated).
- Task discriminants — can be useful for some algorithms and has low overhead.
- No user-defined task attributes
  - introduces a dynamic feature into the run-time that has complexity and overhead.
Reliability

Ada95 Ravenscar profile (Burns, Dobbing, Romanski ‘98)

- **No use of dynamic priorities** — ensures that the priority assigned at task creation is unchanged during the task's execution, except when the task is executing a protected operation.

- **Protected procedures as interrupt handlers** — required if interrupts are to be handled.

*commercially available and employed in hard-real-time systems*
Reliability

Temporal logic

• Extending predicate logic
• Adding the concepts of ordering for events and states
• Suitable for event driven system, reactive systems \( \text{\textcopyright} \) Esterel
Reliability

Temporal logic

• Assertions on sequences and orders of states
  employ predicate logic & a set of new operators:

  □ A: A is true for all future states
  ◊ A: A is eventually true
  ◐ A: A is true for the following state

  e.g. □(Collision_Warning ⇒ ◊Collision_Avoidance)
  or: □(Collision_Warning ⇒ ◐Collision_Avoidance)

assuming that there is a sequence of distinguishable states (or ‘time’) S.
Temporal logic

• Another temporal operator:

\( A \mu B \): \( A \) holds until the first occurrence of \( B \), which will occur eventually.

e.g.

\( ((\text{Tasks}_\text{Waiting} \mu \text{Entry}_\text{Closed}) \land (\neg \text{Tasks}_\text{Waiting} \mu \text{Entry}_\text{Open})) \)

Temporal logic expresses the order of events only and has means to express temporal scopes, deadlines, …
Reliability

Real-Time Logic

• Assertions on real-time events:
  ☞ employ predicate logic & a occurrence function:

  \[ @ (E, i) \]: denotes the time of the \( i \)-th occurrence of event (-class) \( E \)

☞ the event (-class) \( E \) is strictly ordered by instance \( (i) \) and time \( (@) \).
☞ all events of kind (class) \( E \) can be distinguished.
☞ instance order \( \Rightarrow \) order in time.
Real-Time Logic

• Occurrence times of predicates:
  \( \uparrow A \): denotes the time when \( A \) changes from false to true.
  \( \downarrow A \): denotes the time when \( A \) changes from true to false.

Examples:
\[
\forall i \exists j \ ((\uparrow A, i) \leq @((\uparrow A, j)) \land (\downarrow A, j) \leq @((\downarrow A, j-1)) \land (\uparrow A, j)) \land (\downarrow A, j-1) \leq @((\downarrow A, j)) \land (\uparrow A, j))
\]
\[
\forall \forall i @((\downarrow A, i) < @((\uparrow B, i)) \lor (\downarrow B, i) < @((\uparrow A, i)))
\]

Interpretations:
  does \( \forall i @((E, i)) \) indicate all possible, all defined, or all observed instances of \( E \)?
Reliability

Linear Temporal Logic of Real Numbers (LTR)

\[ \phi ::= p \mid \phi_1 \lor \phi_2 \mid \neg \phi \mid \phi_1 U \phi_2 \mid \phi_1 S \phi_2 \]

where

\[(\tau, t) \models p \quad \text{iff} \quad p \in \tau(t)\]

\[(\tau, t) \models \phi_1 \lor \phi_2 \quad \text{iff} \quad (\tau, t) \models \phi_1 \text{ or } (\tau, t) \models \phi_2\]

\[(\tau, t) \models \neg \phi \quad \text{iff} \quad (\tau, t) \not\models \phi\]

\[(\tau, t) \models \phi_1 U \phi_2 \quad \text{iff} \quad \exists t' > t \land t' \models \phi_2 \text{ and } \forall t'' \in (t, t'), t'' \models \phi_1 \lor \phi_2\]

\[(\tau, t) \models \phi_1 S \phi_2 \quad \text{iff} \quad \exists t' < t \land t' \models \phi_2 \text{ and } \forall t'' \in (t', t), t'' \models \phi_1 \lor \phi_2\]

\[\phi \text{ is satisfiable iff } \exists (\tau, t) \models \phi \quad \text{—} \quad \phi \text{ is valid iff } \forall (\tau, t) \models \phi\]
Reliability

Event-Clock Temporal Logic

\[ \phi ::= p | \phi_1 \lor \phi_2 | \neg \phi | \phi_1 U \phi_2 | \phi_1 S \phi_2 | \triangleleft_I \phi | \triangleright_I \phi \]

where

\[(\tau, t) \models p \quad \text{iff} \quad p \in \tau(t)\]

\[(\tau, t) \models \phi_1 \lor \phi_2 \quad \text{iff} \quad (\tau, t) \models \phi_1 \text{ or } (\tau, t) \models \phi_2\]

\[(\tau, t) \models \neg \phi \quad \text{iff} \quad (\tau, t) \not\models \phi\]

\[(\tau, t) \models \phi_1 U \phi_2 \quad \text{iff} \quad \exists t' > t \land t' \models \phi_2 \text{ and } \forall t'' \in (t, t'), t'' \models \phi_1 \lor \phi_2\]

\[(\tau, t) \models \phi_1 S \phi_2 \quad \text{iff} \quad \exists t' < t \land t' \models \phi_2 \text{ and } \forall t'' \in (t', t), t'' \models \phi_1 \lor \phi_2\]

\[(\tau, t) \models \triangleleft_I \phi \quad \text{iff} \quad \exists t' < t \land t' \models \phi \text{ and } \forall t'' \in (t - I, t), t'' \not\models \phi\]

\[(\tau, t) \models \triangleright_I \phi \quad \text{iff} \quad \exists t' > t \land t' \models \phi \text{ and } \forall t'' \in (t, t + I), t'' \not\models \phi\]
Reliability

Metric-Interval Temporal Logic

\[ \phi ::= p \mid \phi_1 \land \phi_2 \mid \neg \phi \mid \phi_1 \hat{U}_I \phi_2 \mid \phi_1 \hat{S}_I \phi_2 \]

where

\( (\tau, t) \vdash p \) iff \( p \in \tau(t) \)

\( (\tau, t) \vdash \phi_1 \lor \phi_2 \) iff \((\tau, t) \vdash \phi_1 \) or \((\tau, t) \vdash \phi_2 \)

\( (\tau, t) \vdash \neg \phi \) iff \((\tau, t) \nvDash \phi \)

\( (\tau, t) \vdash \phi_1 \hat{U}_I \phi_2 \) iff \( \exists t' \in (t, t + l) \land t' \nvDash \phi_2 \) and \( \forall t'' \in (t, t'), t'' \nvDash \phi_1 \)

\( (\tau, t) \vdash \phi_1 \hat{S}_I \phi_2 \) iff \( \exists t' \in (t - l, t) \land t' \nvDash \phi_2 \) and \( \forall t'' \in (t', t), t'' \nvDash \phi_1 \)
Summary

Reliability

• Terminology
  • Faults, Errors, Failures – Reliability

• Faults
  • Fault avoidance, removal, prevention ⇄ Fault tolerance

• Redundancy
  • Static (TMR, NMR) and dynamic redundancy
  • N-version programming, and dynamic redundancy in software design

• Reduce & Formalise
  • Ada95 Ravenscar profile
  • Real-time Logic