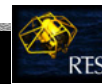


RES

5

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.rti.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			Watch- dog	8/12	Timer	Sync	V _{IN-}				V _{IN+}				Pause	Loop	
0	1	1	1																				
0	0	0	0																				
0	1	1	1	Instruction RAM (RAM Pointer = 01)	R/W	Don't Care					>/<	Sign	Limit #1										
0	0	0	0																				
0	1	1	1																				
0	0	0	0	Instruction RAM (RAM Pointer = 10)	R/W	Don't Care					>/<	Sign	Limit #2										
0	1	1	1																				
1	0	0	0																				
1	0	0	0	Configuration Register	R/W	Don't Care			DIAG	Test = 0	RAM Pointer	I/O Sel	Auto Zero _{oc}	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 to Generate INT2					Sequencer 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	Sign	Conversion Data: MSBs							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		
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		

• 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					Sequencer 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		

• Read interrupt status (interrupt status register, 15 bits):

- indicates the current interrupt conditions, incl. the actual number of conversions and the currently processed instruction

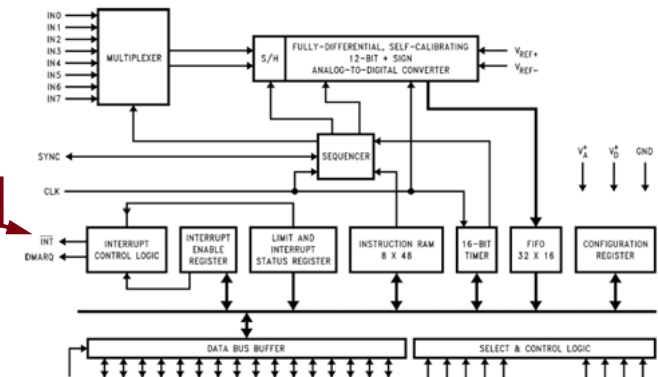
☞ all of the status bits (0-5, 7) are reset with any read access to this register!

Interrupts

LM12L458

(National Semiconductor)

Interrupt signal



☞ only one interrupt signal line available!

☞ in order to identify the interrupt reason, an additional read cycle is required!



Interrupt control:

... at the individual device level

... at the system interrupt controller level

... at the operating system level



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)



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



(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:

intConnect	Connect a routine to an interrupt vector
intLevelSet	Set the interrupt mask level
intLock	Disable interrupts (besides NMI)
intUnlock	Enable interrupts
intVecBaseSet	Set the interrupt vector base address
intVecBaseGet	Get the interrupt vector base address
intVecSet	Set an interrupt vector
intVecGet	Get an interrupt vector

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 ⇨ '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



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.



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.



Interrupts ⇔ 'Signals'

Some common UNIX OS entries:

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



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)



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).



Historic exception handling methods:

- Use an 'unusual return value' and a global variable convention: '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
};

```



Historic exception handling methods:

- Use an 'unusual return value' and a global variable convention: '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
};

```

- ⚠ *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)



Assembler level exception handling methods:

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

Caller:

```

jsr pc, PRINT_CHAR
jmp IO_ERROR
jmp DEVICE_NOT_ENABLED
# normal processing

```

Subroutine:

```

% 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

```

Asynchronism

Exceptions

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.

Asynchronism

Exception indication

Four cases of modern exception indication:

raised:	from:	run-time environment	application
synchronously		*	*
asynchronously		*	*

Asynchronism

Exception indication

Ada95:

raised:	from:	run-time environment	application
synchronously	exceptions		
asynchronously	interrupt/signal handler	asynchronous transfer of control	

Asynchronism

Exception indication

Real-time Java:

raised:	from:	run-time environment	application
synchronously	exceptions		
asynchronously	asynchronous exceptions		



Asynchronism

Exception indication

POSIX:

raised:	from:	run-time environment	application
synchronously		N/A	(global variables)
asynchronously		(signals)	



Asynchronism

Exception granularity

at block level:

Ada95:

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

or in RT-Java

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

all exceptions need to be (or are) declared!



Asynchronism

Exception granularity

at block level:

Ada95:

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

or in RT-Java

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

handlers can catch all! but 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;
```

might be too unspecific



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 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 C
    end;
  begin
    -- read pressure sensor
    exception -- handler for Constraint_Error for pressure
    end;
  ...
  exception -- handler for other possible exceptions
  end;

```

becomes quickly unreadable



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;

```

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).



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).



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!)



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.



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.



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 ☞ real-time systems.

Problem:

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



Exception handling: resumption or termination?

Hybrid-model (offered in: Mesa):

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



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:

```

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.



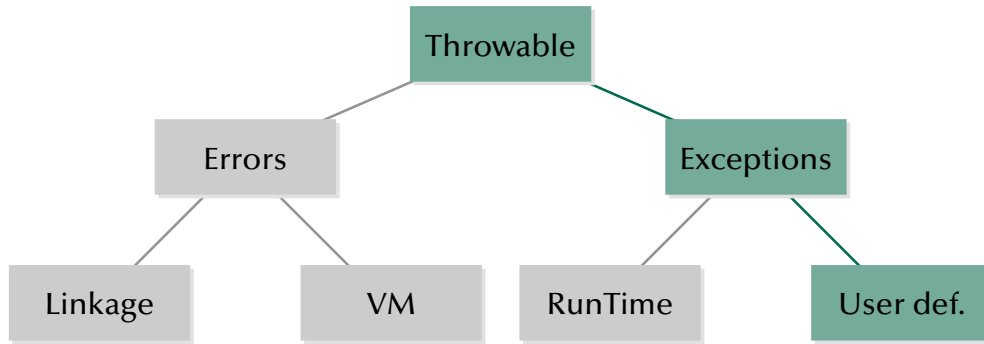
Issues when handling exceptions in Ada95:

- ☞ 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)

Exception handling in Real-time Java



☞ 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)

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.

Exception handling: compare sheet

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



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*.



Asynchronism

Atomic actions

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:

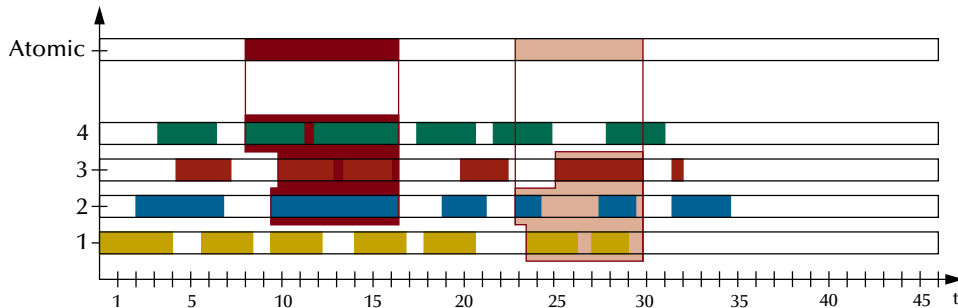
- ☞ **to be interrupted** (due to the failure of one of them)
- ☞ and **to reset** to their initial state at any time ('no effect' is visible to the outside)



Asynchronism

Atomic actions

Time-lines:



Asynchronism

Atomic actions

Nested atomic actions:

Action actions can be nested, iff ...

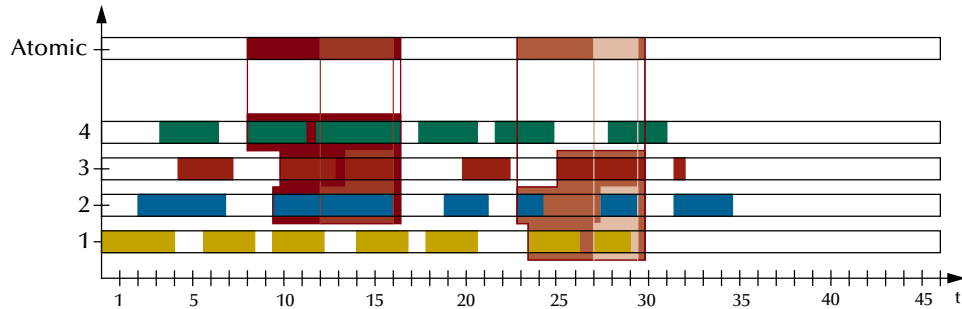
- ... all processes involved in the nested atomic actions are a true subset of the processes involved in the enclosing atomic action.



Asynchronism

Nested atomic actions

Time-lines:



Asynchronism

Atomic actions – Requirements for real-time environments:

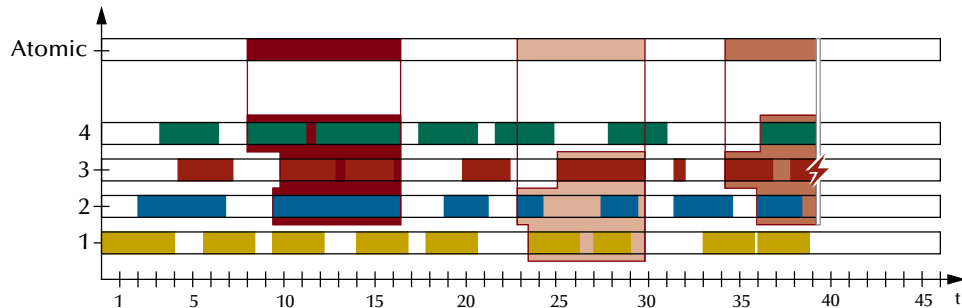
- **Well-defined boundaries**
 - A *start, end* and a *side* boundary:
 - ☞ Define clear entry and exit points for all processes involved in the atomic action. Processes can enter at different time, but need are released from the action at once.
 - ☞ Separate the involved processes from the rest of the system ('side boundary').
- **Indivisibility (Isolation)**
 - *Prohibit or restrict communications* to outside processes and resources.
 - Employing results from one atomic action in another one ☞ implies *strict serialisation*.
- **Nesting**
 - Atomic actions may be nested, if they form a *true enclosing relation*.
- **Concurrency**
 - Independent atomic actions may be executed in *any order* and concurrently.



Asynchronism

Atomic actions

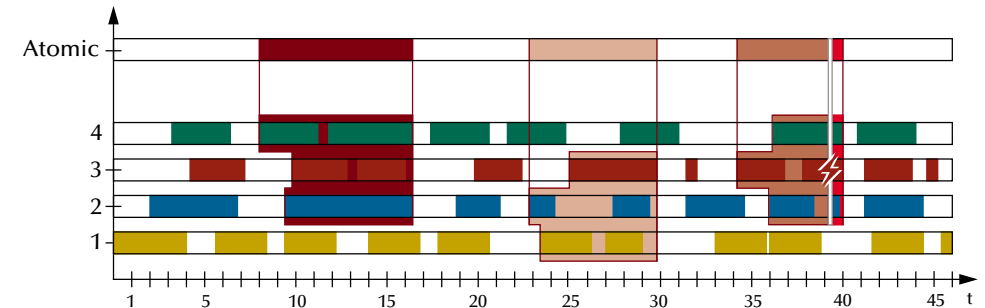
Failure in one action part:



Asynchronism

Atomic actions

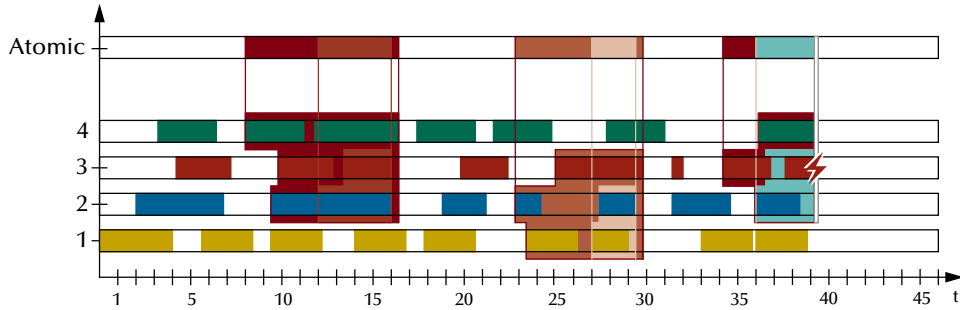
Failure in one action part: ☞ 'clean up' (restore the initial states)



Asynchronism

Nested atomic actions

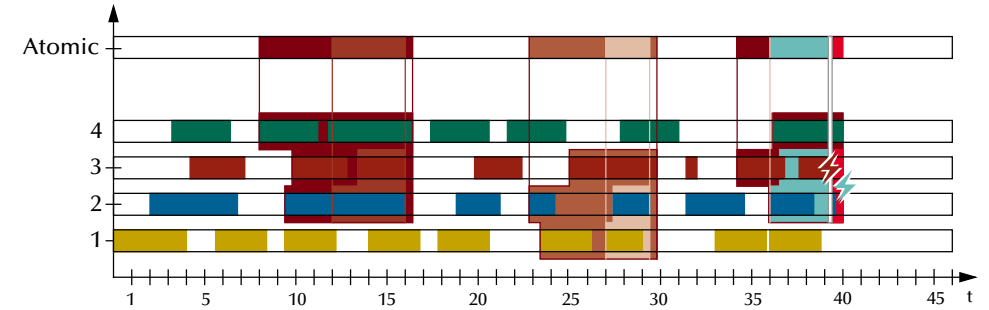
Failure in one part of a nested atomic action:



Asynchronism

Atomic actions

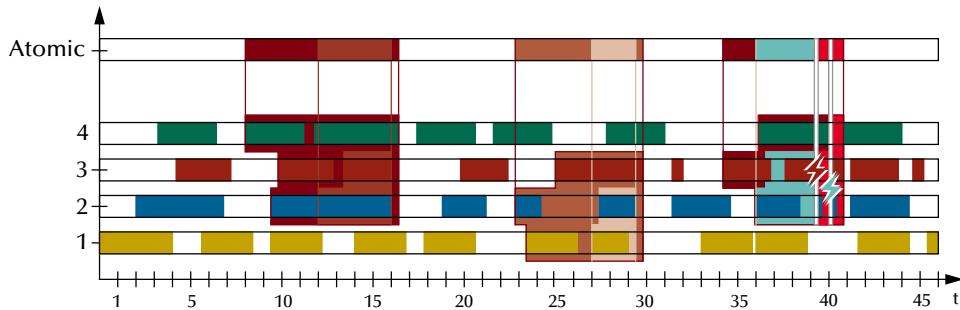
Failure in one part of a nested atomic action (iterative failure propagation):



Asynchronism

Atomic actions

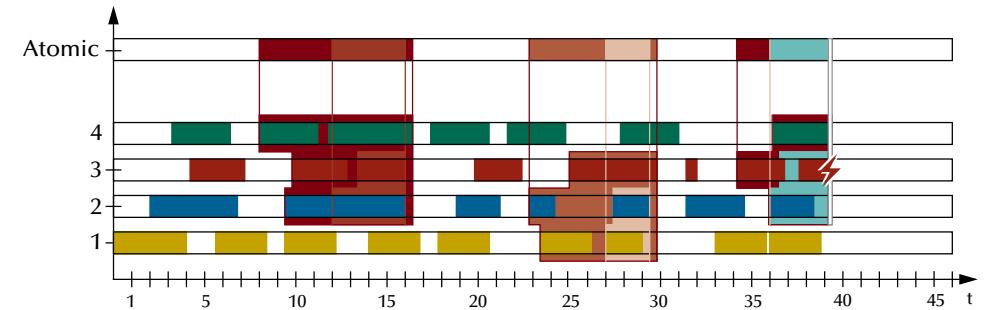
Failure in one part of a nested atomic action (iterative failure propagation):



Asynchronism

Nested atomic actions

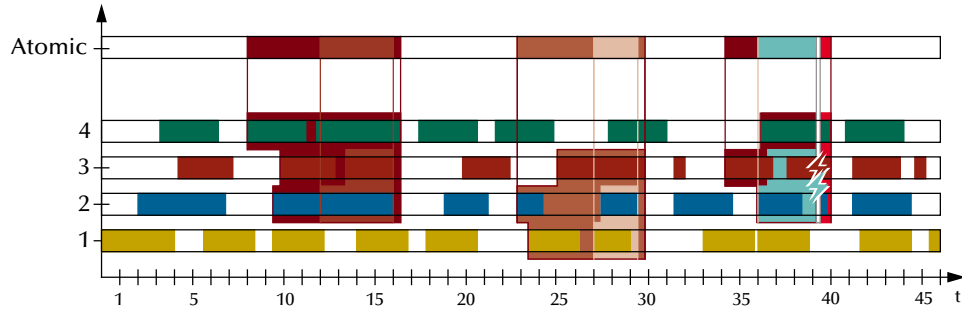
Failure in one part of a nested atomic action:



Asynchronism

Atomic actions

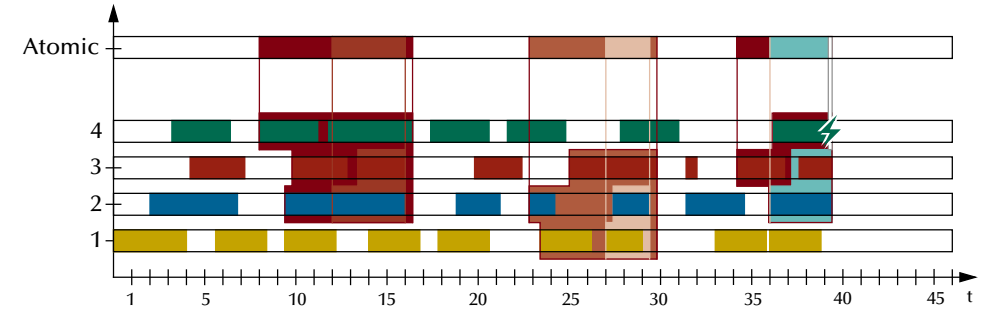
Failure in one part of a nested atomic action (immediate failure propagation):



Asynchronism

Atomic actions

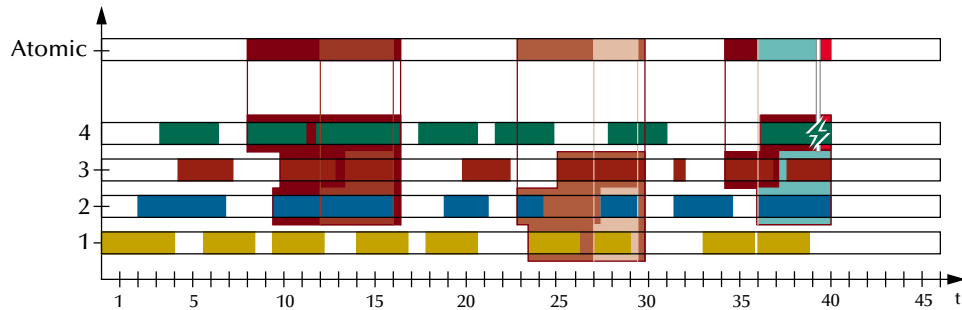
Failure in an enclosing atomic action:



Asynchronism

Atomic actions

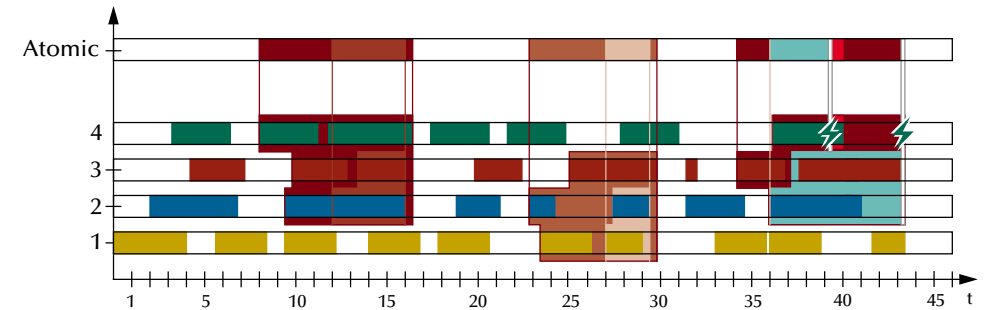
Failure in an enclosing atomic action (no communication with inner action):



Asynchronism

Atomic actions

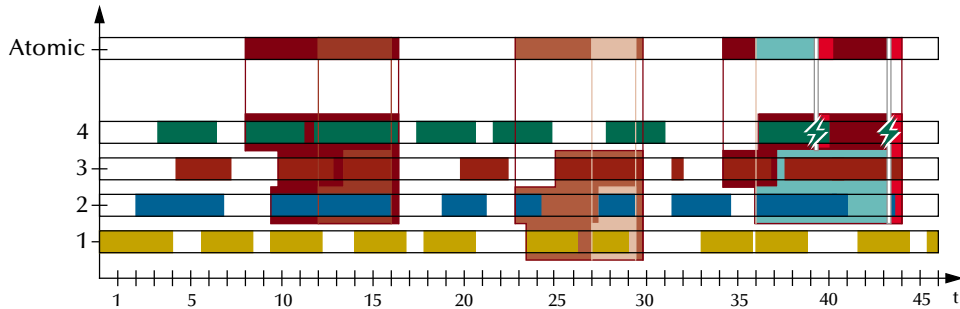
Failure in an enclosing atomic action (no communication with inner action):



Asynchronism

Atomic actions

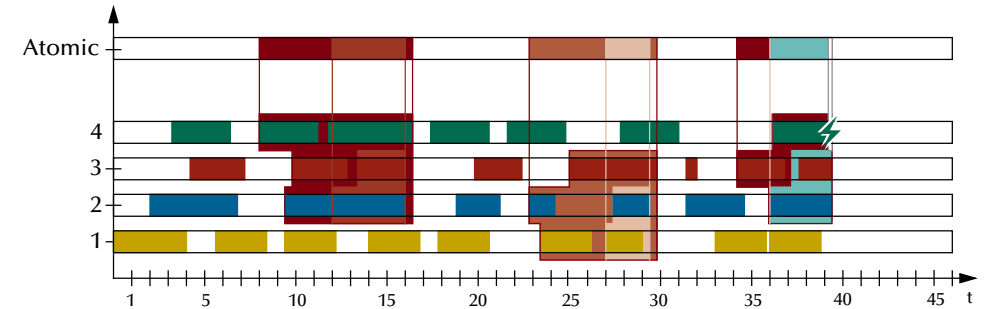
Failure in an enclosing atomic action (no communication with inner action):



Asynchronism

Atomic actions

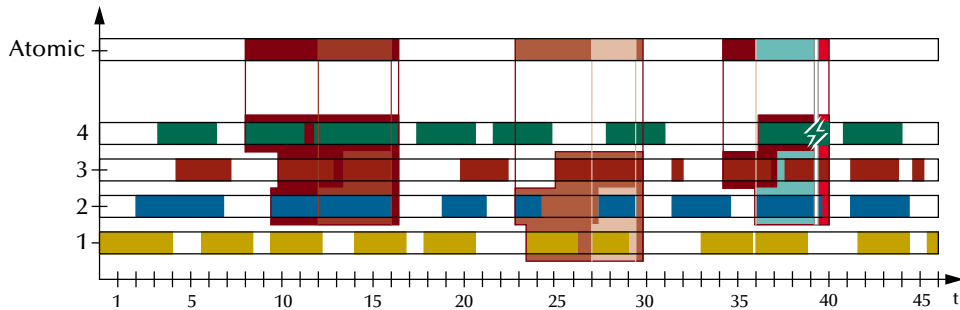
Failure in an enclosing atomic action:



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.

```

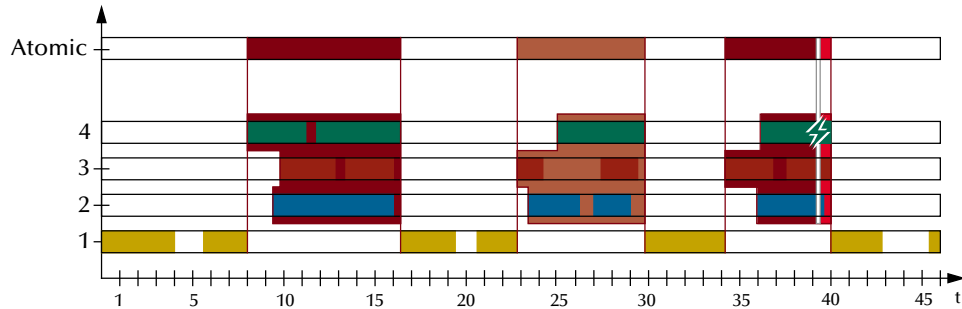
action A with (P2, P3, ...) do
    ...
    -- communication is restricted to P2, P3, ...
    -- exceptions and timing constraints violations are propagated
    -- to all involved processes
    ...
exception
    when exception_a => ... -- recover locally
    when exception_b => ... -- recover locally
    ...
    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:

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:

```
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_Procs is record
  Action,
  Cleanup : Action_Part_Proc;
  Scope   : Action_Part_Time_Scope;
end record;
```

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

☛ The Cleanup procedure is meant to restore the initial state! ('undoing' all effects of Action)



Asynchronism

Atomic actions in Ada95:

```
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;
```

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;
  end;
end;

```

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 select;
end Action_Task;

```

Atomic actions in Ada95:

```

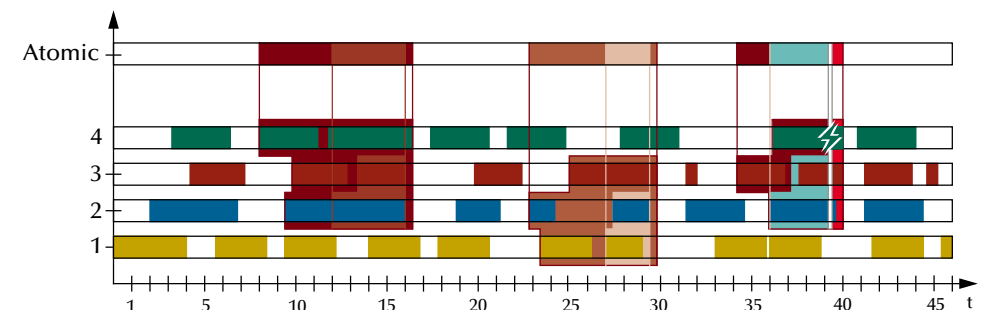
protected Monitor is
  entry Check_In (Task_Id : in Task_Ids);
  entry Fail (Condition : in Atomic_Condition);
  entry Failed (Task_Id : in Task_Ids);
  -- blocking until Fail is called
  entry Check_Out (Task_Id : in Task_Ids);
  -- blocking until all parts are completed or all are cleaned up
  entry 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

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.



Asynchronism



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.



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 AsyncEventHandlers 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



Real-time Java: Asynchronous events

```
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 ();
    ...
}
```



Real-time Java: Asynchronous events

```
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 ();
    ...
}
```



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,
                                   ProcessingGroupParameters group,
                                   boolean nonheap,
                                   java.lang.Runnable logic);
}
```

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

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!

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).

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

```
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();
}
```

code need to be part of a RealtimeThread

will propagate any other exception immediately

Real-time Java: Asynchronously interrupting exception

```
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

```
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

```
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 a AIE class.
 - ☞ The `AIE.doInterruptible` can only be called by one thread at a time!

Asynchronism

Real-time Java: Timeout on actions

```
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!

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 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 :
    function Current_Handler (Inter
    procedure Attach_Handler (New_H
        Interrupt : in Interrupt_ID);
    procedure Exchange_Handler (Old_H
        New_Handler : in Parameterless_Handler;
        Interrupt : in Interrupt_ID);
    procedure Detach_Handler (Inter
    function Reference (Interrupt : I
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.



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

  procedure Detach_Handler (Interrupt : 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



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

  procedure Detach_Handler (Interrupt : 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.



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

  procedure Detach_Handler (Interrupt : Interrupt_ID);

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

```

Metrics: The implementation shall document to worst case overhead for an interrupt handler invocation (in clock cycles).



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

  procedure Detach_Handler (Interrupt : 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.

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.

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 queued 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;
```

Ada95: Asynchronous Transfer of Control

Exception handling:

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

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

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

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;
```

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

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

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;
  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;
  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;

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;

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

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