Asynchronism close to hardware

Interrupts

Interrupt control ...

- ... at the individual device level
- ... at the system interrupt controller level
- ... at the operating system level

References

[Dibble2009]
Dibble, Peter
Interrupts
Programming in Ada 2005

[BN2004]
Burns, A & Wehling, A

[AdaRM2012]

A/D, D/A & Interfaces: Examples

LM12L458
(National Semiconductors)

Interrupt control ...

- ... at the individual device level
- ... at the system interrupt controller level
- ... at the operating system level

Select interrupt source (interrupt enable register, 15 bits):

- Watchdog (INT8): limit conditions are fulfilled.
- Instruction (INT7): the instruction pointer equals a pre-programmed value (bits 8-10).
- Conversion (INT2): a specified number of conversions (bits 11-15) have been performed.
- Auto-Zero (INT3): short calibration has been performed.
- Full-Calibration (INT4): long calibration has been performed.
- Pause (INT5): Sequencer arrived at a pause state.
- Active (INT7): Controller returned from power-down to active mode.

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

- Indicates the currently active interrupts.
- Shows the actual number of conversions and the currently processed instruction.
- Reading this register resets all status bits (bits 6, 7).
Asynchronism close to hardware

Interrupts

Interrupt control ...
- ... at the individual device level
- ... at the system interrupt controller level
- ... at the operating system level

Interrupts come on three levels:
- Interrupt service routines: Does not involve processes.
  All activities under strict real-time constraints will be addressed here.
- Communicating interrupts to processes: Synchronize to events or data structures.
  After the interrupt part of the operation has been addressed, processes might need to be notified about the current state.
- Transforming interrupts to signals: Notify a process via the scheduler.
  For activities which are not under real-time constraints, the interrupt can be translated into a signal (scheduler-level event).

Interrupt service routines (available in real-time operating systems and languages only)

Purpose:
- Allow full access to the interrupt controller (interrupt vectors, priority, and device).
- Activate a routine in an addressable amount of time.

Cannot operate on the level of threads or tasks!

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

Initial response to interrupt (executed by CPU without any software involved):
- save (push) essential CPU registers (IP, condition flags, (SP))
- jump to the (vectorized) interrupt service routine
  - Allow full access to the interrupt controller (interrupt vectors, priority, and device).
  - Activate a routine in an addressable amount of time.
- Cannot operate on the level of threads or tasks!

Interrupt service routines (available in real-time operating systems and languages only)

Some VxWorks OS entries:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inConnect</td>
<td>Connect a routine to an interrupt vector</td>
</tr>
<tr>
<td>inLevelSet</td>
<td>Set the interrupt mask level</td>
</tr>
<tr>
<td>intLock</td>
<td>Disable interrupts (outside 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>inVecBaseGet</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>
Asynchronism close to hardware

Interrupt service routines

- Available in real-time operating systems and languages only
- Initial setup of interrupt service routines
  - Save sensitive CPU registers
  - Jump to vectorized interrupt service routines
- Interrupt service routines
  - Operate at the device or interrupt level
- Interrupts
  - Communicate with the operating system
  - Signal the operating system of a problem
- Signals
  - Communicate synchronization information
  - Used for system-wide coordination

Interrupt control

- At the device level
- At the system interrupt controller level
- At the operating system level

Interrupt service routine to task communication methods:
- Save (push) essential CPU registers
- Jump to the interrupt service routine
- Special CPU instructions (also added by the OS)
- Restore (pop) all remaining/touched CPU registers
- Interrupt service routine code

Interrupts:
- Pipes
- Signals
- Exceptions

Current implementation methods:
- Use an unusual return value and a global variable convention

Wish list for exception handling in real-time programming languages:
1. Exception facilities should not introduce significant overhead
2. Exception facilities should produce no or minimal runtime overhead
3. Exceptions should produce predictable runtime behavior
4. Exceptions should be handled consistently across different platforms
5. Exception handling should not interfere with program execution
6. Exception handling should support concurrent programming paradigms
7. Exception handling should support resource recovery
Asynchronism in general program flows

Exceptions

(Assembler level)

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

```
for pc, PRINT_CHAR
jmp ID_ERROR
jmp DEVICE_NOT_ENABLED
# normal processing
```

Subroutine:

1. indicate an exception;
2. increment the return address on the stack;
3. by the exception routine to employ the caller-provided exception handling;
4. indicate normal operation;
5. increment the return address on the stack by the max. exception number + 1

```
% indicate an exception:
% indicate normal operation:
% indicate an exception:
% indicate normal operation:
```

Caller:

```
<exception-name>:

begin

<... statements ...>

[<long jump to continuation code>]

end
```

```
Jump label for unrecoverable exceptions.
Procedure variable for recoverable exceptions.
```

Exception handling routines are replaced by running over a new "0" code section for the same exception-name.

Exception handling is concluded by either:

- Running over the "end" statement and resume execution with the statement which follows "immediately after" the one which raised the exception.
- Executing a long jump to anywhere.
- Continuing in the code-scope of/after the exception handler was not considered.

Exception handling routines are "registered globally" by running over an "0" code section:

```
<exception-name>:

begin

( ... statements ... )

( <long jump to continuation code> )

end;
```

Exception handling is messy/hard/impossible in:

- Weakly/not scoped languages
- Macro-assembler level languages
- Some historical constructs in not-fully-scoped languages

Exception handling is messy/hard/impossible in:

- Strict functional languages
- Asynchronously

Exception handling is concluded by either:

- Running over the "end" statement and resume execution with the statement which follows "immediately after" the one which raised the exception.
- Executing a long jump to anywhere.
- Continuing in the code-scope of/after the exception handler was not considered.

Exception handling is "registered globally" by running over an "0" code section:

```
<exception-name>:

begin

( ... statements ... )

( <long jump to continuation code> )

end;
```

Exception handling routines are replaced by running over a new "0" code section for the same exception-name.

Exception handling is concluded by either:

- Running over the "end" statement and resume execution with the statement which follows "immediately after" the one which raised the exception.
- Executing a long jump to anywhere.
- Continuing in the code-scope of/after the exception handler was not considered.

Asynchronous exceptions

Real-time Java

```
try

// do something dangerous

catch (ExceptionType e) {
    // handle exception e
}
```

POSIX

```
try

// do something dangerous

catch (ExceptionType e) {
    // handle exception e
}
```

```
E: Constraint_Error

Handle a specific exception at the end of a defined code block
```

```
E: Constraint_Error

Handle a specific exception at the end of a defined code block
```

Exception forms

- Real-time Java
- POSIX

Exception granularity

(Blocks level)

```
try

// do something dangerous

catch (ExceptionType e) {
    // handle exception e
}
```

```
try

// do something dangerous

catch (ExceptionType e) {
    // handle exception e
}
```

```
E: Constraint_Error
```

```
E: Constraint_Error
```

```
E: Constraint_Error
```

```
E: Constraint_Error
```

```
E: Constraint_Error
```

```
E: Constraint_Error
```
Asynchronism in general program flows

### Exception granularity

**Block size**: large

```plaintext
declare
subtype Temperature is Integer range 0 .. 100;
subtype Pressure is Integer range 0 .. 1000;
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 temperatures, pressures and flow
-- according to requirements
dexception
-- handler for Constraint_Error
end;
```

### Exception attributes

Instead of catching exceptions after each statement:
- Exceptions can carry further attributes about their source.

### Exception propagation

**Uncaught** exception propagation

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

Common validity-time for each process
- Use a catch-all handler (possible in Ada and Real-time Java) at the highest task level.

### Exception recovery

**Resumption model** (offered in: Pearl, Mesa):
1. Find an exception handler.
2. Execute the exception handler (and potentially raise another exception).
3. After completion of exception handler (and possibly other handlers): return to the invoker and try resume processing "as if nothing happened".

**Block-Resumption model** (offered in: Eiffel):
1. Find an exception handler.
2. Execute the exception handler (and potentially raise another exception).
3. After completion of exception handler (and possibly other handlers):
   - return to the beginning of the code block in which the exception occurred.

### Exception scope and propagation

1. All procedures and functions declare every potentially raised exception (requested in CHILL, proposed in Real-time Java for user defined exceptions).
2. Exceptions are declared for whole modules — not specifically per method (Ada).
3. The environment automatically attaches additional information to exceptions, which are indicating the position of the exception occurrence and other observed conditions (implementation dependent,-flexible, useful for debugging).

### Exception handling

**Handling any exception at the end of a defined code block**

Can be combined: handle some specific exceptions (yet not all possible code for "other" exceptions).

### Block-level

```plaintext
begin
-- something dangerous
exception
when E => deal with it;
end;
```

### Exception scope

```plaintext
try
{
// do something dangerous
-- handler for Constraint_Error
// handle exception e

} catch (Exception e) {
// handle exception e
}
```

### Exception granularity

- Large (block size)
- Small (requested in CHILL, requested in Real-time Java for user defined exceptions)

### Exception can carry any number of user defined parameters.

### Problems

- Some errors cannot be 'repaired' (especially all timing related errors).
- Local variables must not be re-initialized (potentially confusing state of the system).
- Exceptions can be raised in the middle of evaluations, which will be hard to restore.

### Common Valitity-time for each process

- Use a catch-all handler (possible in Ada and Real-time Java) at the highest task level.
- Common reaction: PANIC!
Asynchronism

Asynchronism in general program flows

Exception recovery

Termination model

- By default in Ada, Real-time Java, offered in Pearl, Mesa.

1. Find an exception handler.
2. Execute the exception handler (and potentially raise another exception).
3. After completion of the exception handler (and possibly others handlers), return to the scope around the initial exception handler.

Features:
- The method of choice, i.e., exceptions imply that the operation was not successful and something else needs to be done now in a real-time system.
- There is no way to continue execution “as if nothing happened” (might be useful in case that the exception would be identified as of minor impact).

Problems:
- Exception recovery

Hybrid model

- Offered in Mesa.

1. Find an exception handler.
2. Execute the exception handler (and potentially raise another exception).
3. After completion of the exception handler (and possibly others handlers), the exception handler can decide whether to terminate or to resume.

Problems:
- Exception recovery

Exception handling issues

- Exceptions are objects in Real-time Java (Exceptions have a hierarchical relation).
- (Ada) Exceptions are declared at package level, i.e., it's unique, which functions may raise which exceptions!
- (Ada) Exceptions may be propagated outside the scope of their declaration, i.e., only those others can handle them.
- (Ada) Real-time Java: Exceptions in task bodies are not propagated to the parent task, i.e., if there could not be any handlers identified in the task, the task will die silently.
- (Ada) Real-time Java: Exception in task declarations are always propagated to the parent task.
- (Ada) Exceptions in task declarations, which are not handled in the accept statement, are propagated to both involved tasks.

Most expensive uncaught exception up to now: $\approx 10^{96}$

Possible workarounds by using POSIX long jumps or signals.

```
#include <signal.h>
#include <sys/signal.h>

void sig_handler(int sig) {
    if (sig == SIGINT)
        abort();
    else
        // Handle other signals
}

int main() {
    signal(SIGINT, sig_handler);
    // Other code...
    return 0;
}
```

Exception handling in Real-time Java

- Exceptions are objects in Real-time Java (Exceptions have a hierarchical relation).
- Exceptions handlers can catch:
  - (Ada) Exceptions may be propagated outside the scope of their declaration, i.e., they are handled in the calling block.
  - (Ada) Exceptions are declared at package level, i.e., it's unique, which functions may raise which exceptions!
  - (Ada) Exceptions may be propagated outside the scope of their declaration, i.e., only those others can handle them.

Problems:
- Exception recovery

Hiding asynchronism

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

Hiding asynchronism

Atomic actions

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).
- to reset to their initial state at any.

Atomic Actions

- Define clear entry and exit points for all processes involved in the atomic action.
- Separately the involved processes from the rest of the system ('side boundary')
- Prohibit or restrict communications to outside processes and resources.

Atomic actions may be nested, if they form a true enclosing relation.

Hiding asynchronism

Nested atomic actions

Atomic actions

Atomic actions failure

- Failure in outer atomic action.
- Preserve full atomicity.

Atomic actions failure

- Failure in outer atomic action.
- Break atomicity and propagate failure immediately.
- Preserve full atomicity.
Asynchronism

Hiding asynchronism

Atomic actions

- Failure in inner atomic action.
- Break atomicity and propagate failure immediately.

Atomic actions in Ada

with Atomic_Action_Types; use Atomic_Action_Types;
generic
Actions : Action_Parts (Tasks_Index) :=
package Generic_Atomic_Action is
exception
when ...
end

Atomic actions in Ada (implementation)

task body Action_Task is
begin
Monitor.Check_In (Task_Id);
select
Monitor.Failed (Task_Id);
Actions (Task_Id).Clean-up all;
end
begin
select
delay To_Duration (Actions (Task_Id).Scope.Start_Delay_Max);
raise Late_Activation_State;
then short
end
begin
delay To_Duration (Actions (Task_Id).Scope.Start_Delay_Min);
end

Atomic actions in Ada (specification)

type Action_Part_Time_Scope is record
  with
  Action
  Start_Delay_Min : Time_Span := Time_Span_Zero;
  Start_Delay_Max : Time_Span := Time_Span_Last;
  Deadline       : Time      := Time_Last;
end record;
type Action_Part_Procs is array (Positive range <>) of Action_Part_Procs;

Atomic actions implementations

- Implemented by dedicated tasks.
- "Client task" is blocked during the atomic action.

Atomic actions in Ada (usage)

Actions : Action_Parts (Tasks_Index) :=
package atomic.Action is new Generic_Atomic_Action (Actions);
procedure Perform is
begin
Atomic_Action.Perform;
exception
when ...
end

Scope mechanism is employed to limit communication possibilities of atomic action parts. The performance is atomic for the caller and invisible for others. Failures in one part are automatically propagated in the whole atomic action.
Atomic actions implementations

- Implemented allows for tasks to dedicate themselves.
- Nested actions are possible, client task not blocked.

Atomic actions failure recovery

- 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 real-time environments, failed atomic actions often imply irreversible failures
    - Tracking back and replaying the same atomic action with modified parameters or methods is rarely feasible, considering timing constraints.
    - Backtracking is often physically impossible.

Asynchronous events and transfer of control

- Error recovery — supporting atomic actions and forward recovery.
- Mode changes — sudden changes from normal operations to emergency measures.
- Partial / impure computations — whenever deadlines are more important than precision.
- Operator intervention — User triggered mode changes.

Real-time Java: Asynchronous events

Supplies three basic features:
1. Binding of handlers to internal and external asynchronous events.
2. Asynchronous transfer of control.
3. Asynchronous termination of threads.

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 fire ();
    public int getAndClearPendingFireCount ();
    public void addHandler (AsyncEventHandler handler);
    public void removeHandler (AsyncEventHandler handler);
    public void bindTo (java.lang.String happening);
    public void removeFromFeasibility ();
    protected int getFireCount ();
    protected synchronized void addFire ();
    public void addAsyncEvent (AsyncEvent event);
    public void removeAsyncEvent (AsyncEvent event);
    public void setAsyncEvent (AsyncEvent event);
    public void setAsyncEventExpiry (AsyncEvent event);
    public void setFireCount (int fireCount);
    public int getFireCountExpiry ();
    public int getFireCount ();
    public void run ();
}
```

Real-time Java: Asynchronous events

```
public class AsyncEventHandler extends AsyncEventHandler {
    public void run ();
    public void addFire ();
    public void addAsyncEvent (AsyncEvent event);
    public void removeAsyncEvent (AsyncEvent event);
    public void setAsyncEvent (AsyncEvent event);
    public void setAsyncEventExpiry (AsyncEvent event);
    public void setFireCount (int fireCount);
    public int getFireCountExpiry ();
    public int getFireCount ();
    public void run ();
}
```

Real-time Java: Asynchronous events

```
public class BoundAsyncEventHandler extends AsyncEventHandler {
    public BoundAsyncEventHandler (SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, java.lang.String name);
    public void run ();
    public void addFire ();
    public void addAsyncEvent (AsyncEvent event);
    public void removeAsyncEvent (AsyncEvent event);
    public void setAsyncEvent (AsyncEvent event);
    public void setAsyncEventExpiry (AsyncEvent event);
    public void setFireCount (int fireCount);
    public int getFireCountExpiry ();
    public int getFireCount ();
    public void run ();
}
```
Asynchronism

Java: Interrupting exception

While an event 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 InterruptedFlag to find out, whether it is supposed to be interrupted:

- If the thread is currently executing and ignoring the flag
- 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.

Asynchronously Interrupting Exception

Real-time Java: Asynchronously interrupting exception

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

Real-time Java:

The asynchronously Interrupting Exception might intercept a control flow directly, while:

- The code region, which are interruptible need to be indicated.
- Synchronized blocks, task creations and finalizations are not interruptible.
- The response time of asynchronous call within (bounded number of) bytecode (which itself need to be documented).
- If the interruptible thread is currently blocked in a Java (in a operation)
  then the thread is either released or the block definition of an allowable blocking time or situation is implementation dependent.

Asynchronism

Real-time Java: Asynchronously interrupting exception

Cases of operations if a synchronized InterruptedException occurred:

1. If control is in a synchronized section (within an interruptible section):
   - The AIE is put into a waiting state.
2. If control is in an interruptible section:
   - Control is transferred to the nearest dynamically enclosing catch clause, which is in an non-interruptible section.
3. If control is in an interruptible section:
   - Nothing happens until an interruptible section is reached.
4. If an interruptible section is reached while propagating an exception:
   - Original exception is discarded or replaced by the AIE.

Real-time Java: Asynchronously interrupting exception

public class AsynchronouslyInterruptedException extends java.lang.InterruptedException {
    public synchronized void doInterruptible (InterruptibleLogic)
        implements NonInterruptibleServices*
    {
        AIE stopNow = AIE.getGeneric();
        try {
            // code interrupted with calls to NonInterruptibleServices
            AIE.AI AI = doInterruptible (InterruptibleLogic);
            if (stopNow.happened (false)) {
                // handle the AIE
                AIE AI = AIE.getGeneric();
            }
        } catch (AIE AI) {
            // handle the AIE
        }
    }
}

Real-time Java: Asynchronously interrupting exception

import NonInterruptibleServices *
public class InterruptibleService {
    public AIE stopNow = AIE.getGeneric();
    public boolean Service () throws AIE {
        try {
            // code is propagate anything else
            AIE AI = AIE.getGeneric();
            if (stopNow.happened (false)) {
                // handle the AIE
            }
        } catch (AIE AI) {
            // handle the AIE
        }
    }
}

Asynchronism

Real-time Java: Interruptible interface

public interface Interruptible {
    public void interruptible (AIE InterruptibleException)
    public void run (AIE InterruptibleException exception)
        throws AIE InterruptibleException;
}

An object may declare an interruptible method explicitly by implementing the above interface.

- sole 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

Form of asynchronism

**Ada**

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

Set of different methods to handle different kinds of events!

---

Ada Interrupts

**Ada: Interrupt handlers**

```ada
package Ada.Interrupts is
  type Interrupt_ID is implementation-defined;
  type Parameterless_Handler is access protected procedure;
  function IsReserved (Interrupt : Interrupt_ID) return Boolean;
  function IsAttached (Interrupt : Interrupt_ID) return Boolean;
  function Current_Handler (Interrupt : Interrupt_ID) return Parameterless_Handler;
  procedure Attach_Handler (New_Handler : in Parameterless_Handler; Interrupt : Interrupt_ID);
  procedure Exchange_Handler (Old_Handler : out Parameterless_Handler; New_Handler : in Parameterless_Handler; Interrupt : Interrupt_ID);
  procedure Detach_Handler (Interrupt : in Interrupt_ID);
  function Reference (Interrupt : Interrupt_ID) return System_Address;
end Ada.Interrupts;
```

---

Asynchronism

Notions of time in **Ada**

```ada
package Ada.Real_Time.Timing_Events is
  type Timing_Event is tagged limited private;
  type Timing_Event_Handler is access protected procedure (Event : in out Timing_Event);
  procedure Set_Handler (Event : in out Timing_Event; At_Time : in Time; Handler : in Timing_Event_Handler);
  procedure Set_Handler (Event : in out Timing_Event; In_Time : in Time; Handler : in Timing_Event_Handler);
  function Current_Handler (Event : Timing_Event) return Timing_Event_Handler;
  procedure Cancel_Handler (Event : in out Timing_Event; Cancelled : out Boolean);
  function Time_Of_Event (Event : Timing_Event) return Time;
private...
end Ada.Real_Time.Timing_Events;
```

---

Asynchronism

**Ada: Asynchronous Transfer of Control**

Execute the trigger (entry-call or delay)

```ada
select
  entry-call | delay
  [ - statements - ]
then short
  - statements -
  and select.
```

---

Asynchronism

**Ada: Asynchronous Transfer of Control**

Execute the trigger (entry-call or delay) when:

1. The trigger can be completed (Thompsonian style)
2. Following the trigger are executed and the delayed statement is completed (the abortable part is never started).

```ada
select
  entry-call | delay
  [ - statements - ]
then short
  - statements -
  and select.
```
Asynchronism

Ada: Asynchronous Transfer of Control

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

1. If the trigger can be completed:
   a. The optional statements following the trigger are executed and the select statement is completed (the abortable part is never started).
   b. If the trigger is blocked or requeued to a blocked entry:
      - The statements in the abortable part are executed:
         - An attempt is made to revoke the triggering statement. The select statement is completed after the cancelled or completed triggering statement.
         - The abortable part is aborted, and the optional statements following the trigger are executed and the select statement is completed.

2. If the abortable part completes before the trigger is completed:
   a. The statements in the abortable part are executed.
   b. If the abortable part is completed before the abortable part is completed:
      - An attempt is made to revoke the triggering statement. The select statement is completed after the cancelled or completed triggering statement.
      - The abortable part is aborted, and the optional statements following the trigger are executed and the select statement is completed.

Note about exception handling:

All parts of a select-then-abort statement can raise exceptions, yet, in case of an interruption of the abortable part, the exceptions from the abortable part are lost!

Equivalent?
Asynchronism

Ada: Asynchronous Transfer of Control

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

Case 1a: ST completes before timeout

Case 1b: ST does not complete before timeout

Case 2: Rendezvous completes before ST

Case 2b: ST completes before rendezvous

Case 3: Timeout occurs before the rendezvous starts

Case 3a: ST completes before rendezvous starts

Case 3b: Rendezvous completes before ST starts

Case 3c: Rendezvous completes before ST completes
Asynchronism in Ada 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/synchronization/termination).
- 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 tasks.

(Differences)
- Ada
  - Mechanisms: Interrupts are interrupts and ATC is embedded in the 'select' scheme.
  - Assumes that all code which is called from within an ATC-enabled region is ATC-enabled.
  - Delivers an interrupt to one global handler and each ATC-enabled region has exactly one exit point.
- Real-time Java
  - Mechanisms: Asynchronism is embedded into the synchronous exception scheme.
  - Declares ATC-enabled regions per method and any asynchronous event is deferred until the next ATC-enabled method is executing.
  - Delivers asynchronous events to all enrolled handlers and propagates an asynchronous interrupting event through the closest handlers.

Summary
- Interrupts / Signals
  - Device/system/language/operating-system level interrupt control.
  - Characteristics of interrupts and signals.
- Exceptions
  - Exception classes/ granularity/parametrization/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 Ada.