An Introduction to Ada 95 for Programmers
(Part 2 – Parallel and Real-time Programming)

Dr. David A. Cook
DCook@AEGISTG.COM

Dr. Eugene W.P. Bingue
Dr.Bingue@ix.netcom.com
Ada-Engineering.com
Multitasking
What is Real-time?
Tasks Construction

- Task types
- Protected types
- Task communications
Task Structure

- Anonymous Tasks
- Task Types
- Task Attributes
- Delay, Delay Until
- Exceptions During Tasking
Task Execution Control

- Task States
- Aborting Task
- Synchronization Points
- Task access types
Real Time Issue

- Potential Overhead Factors.
- Improving Performance
- Special Tasking Issues
Parallel Processing (Tasking)

- The Ada parallel processing model is a useful model for the abstract description of many parallel processing problems. In addition, a more static monitor-like approach is available for shared data-access applications.

- Ada provides support for single and multiple processor parallel processing, and also includes support for time-critical real-time and distributed applications.
What a Task is

7 Concurrently Executing Program Unit
One processor (single thread of control)
Multi-programming (multiple threads)
Multi-processing (multiple threads)
Distributed Environment (sterile)
Distributed Environment

7 Always a Slave
Must have a master
Sometimes abortable
Can be aborted by ANYBODY (who has visibility)
Since a task must have a master, it can never be a library unit

7 What makes the master important?
The master may not terminate until all “children” are finished
Library packages acting as a master may have “rogue” tasks
Simple Task Syntax

```
task [type] task_simple_name [ is
    {entry declaration}
    {representation clause}
end [ task_simple_name ] ] ;

task body task_simple_name is
    [declarative part]
begin
    sequence_of_statements
    [exception
      exception handler ]
end [ task_simple_name ] ;
```
Examples

7 Using simple task types

```ada
task type EAT_UP_RESOURCES;

task body EAT_UP_RESOURCES is
begin
  loop
  null;
end loop;
end EAT_UP_RESOURCES;

EAT_UP_1 : EAT_UP_RESOURCES;
BIG_EAT : array (1..10) of EAT_UP_RESOURCES;
```

7 Using access types with task types

```ada
task type EAT_UP_RESOURCES;

task body EAT_UP_RESOURCES is
begin
  loop
    null;
  end loop;
end EAT_UP_RESOURCES;

type EATER is access EAT_UP_RESOURCES;

EAT_UP_1 : EATER;
EAT_UP_A_LOT : array (1..10) of EATER;
```
When does a task start?

A task starts after the elaboration of the declarative part that each task is declared in. Basically, after the “begin” statement, but before any other executable statement.

This allows TASKING_ERROR to be raised in the “master” in case of problems in the elaboration of a task.

NOTE - This is the ONLY time that a task will raise an asynchronous exception in the master. There may be only 1 TASKING_ERROR per master per declarative region.
Synchronization Calls

The "accept" synchronizes the caller and server, during SOS #1 and prepares the task to execute SOS #2.

SOS #1 occurs during rendezvous, and the caller is "blocked" while the receiver (server) executes the statements. SOS #1 should be only as long as absolutely necessary. SOS #1 may be null.

SOS #2 occurs after rendezvous, and multiple threads of control exist. Both the caller and server are executing in parallel.

```ada
task type DO_SOMETHING is
  entry SYNC_POINT;
end DO_SOMETHING;

task body DO_SOMETHING is
begin
  loop
    accept SYNC_POINT do
      <SOS #1>
      end SYNC_POINT;
      <SOS #2>
      end loop;
  end DO_SOMETHING;
```
Task Communications

7 An “ENTRY POINT” defines a point to rendezvous (synchronization or exchange data point) with a task. You can NEVER call a task, only rendezvous with it at an entry point. An entry point is like a “phone number” to the task

```ada
task type DO_SOMETHING is
  entry SYNC_POINT;
end DO_SOMETHING;

task body DO_SOMETHING is
  begin
    loop
      accept SYNC_POINT;
      <SOS> -- Sequence Of Statements
      end loop;
  end DO_SOMETHING;
```

This is the “WAIT FOREVER” model
Multiple Accept Statements

There is nothing “sacred” about “accept” statements.
There may be multiple accepts per entry point

```ada
task type DO_SOMETHING_ELSE is
  entry SYNC_POINT;
end DO_SOMETHING_ELSE

begin
  loop
    accept SYNC_POINT do
      <SOS #1>
      end SYNC_POINT;

      <SOS #2>
      accept SYNC_POINT;

    end loop;
  end DO_SOMETHING_ELSE;
```
Entry Call Parameters

An entry point may define parameters
(like a procedure or function definition)

```ada
task type DO_LITTLE is
  entry GET_DATA (PARAM1: in SOME_TYPE);
  entry PUT_DATA (PARAM2: out SOME_TYPE);
end DO_LITTLE;

TASK_DO_LITTLE : DO_LITTLE;

task body DO_LITTLE is
  HOLDER : SOME_TYPE;
begin
  loop
    accept GET_DATA (PARAM1: in SOME_TYPE) do
      HOLDER := PARAM1;
      end GET_DATA;

    accept PUT_DATA (PARAM2: out SOME_TYPE) do
      PARAM2 := HOLDER;
      end PUT_DATA;
  end loop;
end DO_LITTLE;
```
What the Previous Example Does

1. Enforces “server-client” relationship for a “critical” data item.
2. Requires a “new” item to be created before it can be “consumed”
3. Requires the current item to be “consumed” before a new item can be created.
4. Will allow multiple producers/consumers to interact by using the task as a “middleman”
Implicit Queues for Entry Points

**Queues**
- By definition of accept statement, only 1 caller may be in rendezvous per task.
- This means that calls for task entries are neither reentrant or recursive

There is a queue associated with each entry point. All callers to this entry stand in an ordered line.
Use “Wait Until I get Done” with Great Care!

Could be replaced with a simple procedure/function call except in special Cases!

Use entry points to pass data “one way” NOT

task type DO_PROCESSING is
    entry DO_WORK (DATA : in out SOME_TYPE);
end DO_PROCESSING;

WORKER : DO_PROCESSING;

task body DO_PROCESSING is
    begin
        loop
            accept DO_WORK (DATA : in out SOME_TYPE) do
                <LSOS> -- some long, involved processing here
            end DO_WORK;
            end loop;
        end DO_PROCESSING;
When You Need to Send and Receive Data From a Task

```
task DO_PROCESSING is
  entry GET_DATA (DATA : in SOME_TYPE);
  entry PUT_DATA (DATA : out SOME_TYPE);
end DO_PROCESSING;

task body DO_PROCESSING is
  HOLDER : SOME_TYPE;
begin
  loop
    accept GET_DATA(DATA: in SOME_TYPE) do
      HOLDER := DATA;
    end GET_DATA;
    <LSOS> -- some long, involved processing here
    accept PUT_DATA(DATA: out SOME_TYPE) do
      DATA := HOLDER;
    end PUT_DATA;
  end loop;
end DO_PROCESSING;
```
Exiting or Quitting a Task

Task “quits” under task control

```ada
begin
  loop
    accept GET_DATA (DATA : in SOME_TYPE) do
      HOLDER := DATA;
      end GET_DATA;
    -- some long processing here
    accept PUT_DATA (DATA : out SOME_TYPE) do
      DATA := HOLDER;
      end PUT_DATA;
    exit when <some condition>;
  end loop;
end DO_PROCESSING;
```

```ada
task type DO_PROCESSING is
  entry GET_DATA (DATA : in SOME_TYPE);
  entry PUT_DATA (DATA : out SOME_TYPE);
end DO_PROCESSING;
WORKER : DO_PROCESSING;
task body DO_PROCESSING is
  HOLDER : SOME_TYPE;
begin
  loop
    accept GET_DATA (DATA : in SOME_TYPE) do
      HOLDER := DATA;
      end GET_DATA;
    -- some long processing here
    accept PUT_DATA (DATA : out SOME_TYPE) do
      DATA := HOLDER;
      end PUT_DATA;
    exit when <some condition>;
  end loop;
end DO_PROCESSING;
```
Multiple Callers - the *Select*

Task TASK2 is
   entry ENTRY1;
   entry ENTRY2;
end TASK2;

Task body TASK2 is
begin
  loop
    select  --Waits for a call of ENTRY1 or ENTRY2
      accept ENTRY1 [do
        <SOS>
      end ENTRY1];
      [<SOS>]
    or
      accept ENTRY2 [do
        <SOS>
      end ENTRY2];
      [<SOS>]
    end select;
  end loop;
end TASK2;
The Select Concerns

7 The order of selection is not defined by the language!!!
   – It may be arbitrary, fair, consistent, inconsistent or predefined!!!
   – Any program that makes assumptions about the order of the selection of the open alternatives should be considered “erroneous”!!!
The Select (cont.)

Each accept statement in a “select” is called an ALTERNATIVE
- Each alternative is allowed to have an optional “guard” of the form
  \[\text{when } <\text{Boolean condition}> \implies \text{accept } ...\]
  - If the guard is true, then the alternative is “open” and the corresponding “accept” is considered
  - If the guard is false, the alternative is called “closed”, and not a possible alternative
  - If all alternatives are closed, a PROGRAM_ERROR is raised!!
  - In any “Wait case”, an alternative is evaluated only once per select!!
Quitting Under Caller Control

task type DO_PROCESSING is
  entry GET_DATA (DATA : in SOME_TYPE);
  entry PUT_DATA (DATA : out SOME_TYPE);
  entry SHUTDOWN;
end DO_PROCESSING;

WORKER : DO_PROCESSING;

begin
loop
select
  accept GET_DATA(DATA: in SOME_TYPE) do
    HOLDER := DATA;
    end GET_DATA;
  or
  accept PUT_DATA(DATA: out SOME_TYPE) do
    DATA := HOLDER;
    end PUT_DATA;
  or
  accept SHUTDOWN;
    --sync call only
  exit;
end select;
end loop;
end DO_PROCESSING;
--Question: What f callers still in queue?
Finite Wait - the *Delay*

This is the **WAIT FOR A FINITE AMOUNT OF TIME** option

The syntax is

\[
\text{delay } <\text{fixed-point DURATION}>; \\
\text{[ } <\text{SOS}> \text{ ]}
\]

The duration is expressed in seconds (X.X)

Since the delay may be dynamic (an expression), a negative value may be used (treated as 0)

Multiple delays are allowed (the shortest one “wins”)

the delay statement may also have a guard

After a time equal to the delay, no other open alternatives will be allowed

After a time >= the delay, the optional <SOS> after the delay is executed, and the select terminates
task FAST_FOOD is
  entry WALK_IN;
  entry DRIVE_UP;
end FAST_FOOD;

task body FAST_FOOD is
  begin
    loop
      select
        when WALK_IN_HOURS => accept WALK_IN do
          ..
        end WALK_IN;
      or
        when DRIVE_UP_HOURS => accept DRIVE_UP do
          ..
        end DRIVE_UP;
      or
        delay 60.0;  --if no customers after 1 minute, clean up
        CLEAN_UP_TABLES;
      end select;
    end loop;
  end FAST_FOOD;
Passive Quitting - *Terminate*

```plaintext
select
  accept ...
or
  accept ...
or
  terminate;
end select;
```

This says “If I have no callers in line, and my master is waiting to quit, and all of my children are ready to quit, then I may now terminate”

- This option is mutually exclusive with the *delay* Thus, you can only use the *terminate* option with a *wait forever* in a select
Don’t Wait at All - the *Else*

- This option is mutually exclusive with both the *delay* and the *terminate* alternative
  ```ada
  select
    accept ...
  or
    accept ...
  or
    accept ...
  else
    <SOS>;
  end select;
  ```

- If there is NOBODY in queue, then perform the sequence of statements

- This option must be used carefully. Depending upon the type of wait the
  caller will take, it can cause huge overhead and prevent “real” work from
  getting done!

- If a caller is using the “don’t wait” option also, what are the odds of
  achieving a rendezvous??
Never Code a *Busy Wait*

```ada
loop
    select
    accept SOME_ENTRY_CALL do ..
        ..
        ..
    end SOME_ENTRY_CALL;
    else
        null;
    end select;
end loop;
```

7 A “busy wait” consumes resources, but can tie up a non-time-slicing system!

7 Specifically, single processor systems are very sensitive to this.
Calling Task Entries

7 As we have seen, there are three ways to “receive” an entry call
   1. Wait forever
   2. Wait for a determinate time
   3. Don’t wait at all

7 There are three corresponding ways to “call” an entry point
NOTE: inside a task, you don’t know who was “placing” the call.
However, to call an entry, you MUST specify both the task name and the entry point.
Wait Forever Entry Call

7 Much like a procedure call. You simply specify the
TASK_NAME.ENTRY_NAME;

....

....

Some_Task.Some_Entry(Some_Parameters);

....

7 Once the “call” is placed, you have ABSOLUTELY
NO CONTROL over how long you wait. Also, you
can’t even determine how many people are in line
ahead of you!!
Timed Entry Call

This allows you to wait for a maximum time in queue, then “jump out of the queue”.

```
select
  TASK_NAME.ENTRY_NAME
  (optional_data);
  <optional SOS>
  or
  delay 60.0;
  <optional SOS>
end select;
```

The select statement is used for BOTH the “selective waits” in receiving an entry call in the task, and for placing calls to a task entry. This orthogonality is very confusing to beginning Ada code readers.
Only On Task at a Time

You can only call one task at a time.

```ada
select
  TASK_ONE.ENTRY_NAME;
or
  TASK_TWO.ENTRY_NAME;  -- ILLEGAL
end select;
```
Don’t Wait at All Entry Call

```ada
select
    TASK_NAME.ENTRY_NAME;
    <optional SOS>
else
    <SOS>
end select;
```

NEVER use this type of call if there is ANY chance that the task you are calling is also using the “else” option.

(translation - don’t use this option except in very special circumstances.)
Task Attributes

Task_Type’Callable;  -- is Task in a callable state.
                     -- Boolean returned.

Task_Type’Terminated;  -- is Task Terminated.
                        -- Boolean returned.

E’Count;  -- number of calls waiting in queue on an Entry.
            -- return Universal_Integer;

T’Identify;  -- Yields a value of Task_ID   (Annex C)
              -- Only allowed inside an entry_body or
              -- accept  statement.
Asynchronous Transfer of Control (then abort)

7 Allows a sequence of statements to be interrupted and then abandoned upon some event.

7 Event is either completion of an entry call, or expiration of a delay.

7 Used for a mode change, time bounded computations, user-initiated interruption, etc..
Asynchronous Transfer of Control
(Creating a *Timeout*)

```ada
select
    delay 5.0;
    Ada.Text_IO.Put_Line(“Calculation does not converge”);
    Some_Default_Action;
then abort
    Horribly_Complicated_Recursive_Function(X,Y);
end select;
-- After 5 seconds (plus a little), I will reach here
```
User-initiated Interrupt

loop
  select
    Terminal.Wait_for_Interrupt;
    Put_Line ("Process Interrupted..");
  then abort
  
    Put_Line ("-> ");
    Get_Line (Command, Last);
    Process_Command (Command (1..Last));

  end select;
end loop;

This process will be abandon by terminal interrupt
Time Bounded Situation

select -- Time Critical Data Processing
    delay 5.0;
    Set_Display_Object_Color (Yellow);
    Put_Line ("Target lock aborted data too old.");
then abort -- Data not received in 5.0 seconds
    Position_Object;
    Set_Display_Object_Color (Green);
end select;
Mode Change

select          -- Mode Change
               Confirmed_Air_Threat;
               Sound_ Tone;
               Crash_Avoidance;

then abort
               Land_Aircraft;
end select;
Protected Types

Protected types provide a low-level, lightweight synchronization mechanism whose key features are:

- Protected types are used to control access to data shared among multiple processes.
- Operations of the protected type synchronize access to the data.
- Protected types have three kinds of operations: protected functions, protected procedures, and entries.
Protected Units & Protected Objects

- Protected procedures provide mutually exclusive read-write access to the data of a protected object.
- Protected functions provide concurrent read-only access to the data.
- Protected entries also provide exclusive read-write access to the data.
- Protected entries have a specified barrier (a Boolean expression). This barrier must be true prior to the entry call allowing access to the data.
package Mailbox_Pkg is
  type Parcels_Count is range 0 .. Mbox_Size;
  type Parcels_Index is range 1 .. Mbox_Size;
  type Parcels_Array is array (Parcel_Index) of Parcels
protected type Mailbox is
  -- put a data element into the buffer
  entry Send (Item : Parcels);
  -- retrieve a data element from the buffer
  entry Receive (Item : out Parcels);
  procedure Clear;
  function Number_In_Box return Integer;
private
  Count : Parcels_count := 0;
  Out_Index : Parcels_Index := 1;
  In_Index : Parcels_Index := 1;
  Data : Parcels_Array;
end Mailbox;
end Mailbox_Pkg;
package body Mailbox_Pkg is

protected body Mailbox is

entry Send (Item : Parcels) when Count < Mbox_Size is
    -- block until room
    begin
    Data (In_Index) := Item;
    In_Index := In_Index mod Mbox_size + 1;
    Count := Count + 1;
end Send;

entry Receive (Item : out Parcels) when Count > 0 is
    -- block until non-empty
    begin
    Item := Data(Out_Index);
    Out_Index := Out_Index mod Mbox_Size + 1;
    Count := Count -1;
end Receive;
Protected Types Example (cont)

procedure Clear is    --only one user in Clear at a time
begin
    Count := 0;
    Out_Index := 1;
    In_Index := 1;
end Clear;

function Number_In_Box return Integer is
    -- many users can check # in
Box
begin
    return Count;
end Number_In_Box;

end Mailbox;

end Mailbox_Pkg;
The Requeue Statement allows a call to an entry to be placed back in the queue for later processing.

Without the `with abort` option, the requeued entry is protected against cancellation.

```ada
requeue Entry_Name [with abort];
```
protected Event is
    entry Wait;
    entry Signal;
private
    entry Reset;
    Occurred : Boolean := False;
end Event;
protected body Event is
    entry Wait when Occurred is
        begin
            null; -- note null body
        end Wait;
    entry Signal when True is
        -- barrier is always true
        begin
            if Wait’Count > 0 then
                Occurred := True;
                requeue Reset;
            end if;
        end Signal;
    entry Reset when Wait’Count = 0 is
        begin
            Occurred := False;
        end Reset;
end Event;
Delay and Until Statements

delay Next_Time - Calendar.Now;
-- suspended for at least the duration specified

delay until Next_time;
-- specifies an absolute time rather than a time interval

The *until* does not provide a guaranteed delay interval, but it does prevent inaccuracies due to swapping out between the “delay interval calculation” and the delay statement.
task body Poll_Device is

    Poll_Time : Real_Time.Time := time_to_start_polling;
    Period : constant Real_Time.Interval := 10 * Milliseconds;

begin
    loop
        delay until Poll_Time;
        -- sequence of statements
        .
        -- to
        .
        -- Poll the device
        Poll_Time := Poll_Time + Period;
    end loop;
end Poll_Device;
Killing a Task
Abort a task

7 The “ABORT” statement can not only kill a task, but can have catastrophic effects upon the entire system.

7 Any program unit that has “visibility” to a task object can “abort” the task thru the abort statement.

abort TASK_NAME;
Aborting a task

7 This causes the task to become “abnormal”

7 If the task is “blocked” or “ready”, it just becomes complete

7 If not, it must become completed prior to any action affecting another task
Abort the Task

7 A task may “complete” in the middle of IO, updating a record, an assignment, etc.

7 Any entry in the tasks’ queues (or a “client” that was in rendezvous) now have a TASKING_ERROR raised

7 A task may kill itself to quickly terminate execution cleanly!!
Abort a Task

7 “An abort statement should be used only in extremely severe situations requiring unconditional termination”

7 Any abort statement (other than a task aborting itself) should only be used as a last resort if the task is non-responsive or a “rogue” task!! Steps must be taken to ensure data and file integrity and recovery!!
Real-time Features Required

for low-level, real-time, embedded, and distributed systems

Systems Programming Annex  Annex C
Real-Time Annex  Annex D

The Real-Time Annex requires the Systems Programming Annex for support
Systems Programming Annex
Annex C
Capabilities (Systems Programming)

• Access to Machine Operations (machine dependent)
  Must have assembler (if available)
  Memory addressing and offsets must be specified
  Overhead with inline vs. subprogram calls documented
  Pragmas for interfacing assembler and Ada must be supplied

• Access to Interrupt Support
  pragma Interrupt-Handler (defines parameterless procedure
  that can be later attached to an interrupts)
  pragma Attach_Handler (can be used to specify attachment
  of parameterless procedure to a specific interrupt at
  initialization time). This pragma can be replaced by a
  dynamic procedure call to Attach_Handler that
  accomplish the same thing
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 : out Parameterless_Handler;
                                New_Handler : Parameterless_Handler;
                                Interrupt   : Interrupt_Id);

    procedure Detach_Handler (Interrupt : Interrupt_Id);

    function Reference (Interrupt: Interrupt_Id)  return Address;

    private
        ... -- not specified by the language
    end Ada.Interrupts;
package Ada.Interrupts.Names is
    implementation_defined : constant Interrupt_Id :=
        implementation_defined;

    ... -- not specified by the language
private
    end Ada.Interrupts.Names;
Shared Variable Control

- **Pragma Atomic** (applies to objects, components, or types)
- **Pragma Atomic_Components** (applies to arrays)
- **Pragma Volatile** (applies to objects, components, or types)
- **Pragma Volatile_Components** (applies to arrays)

The Atomic pragmas force indivisible read/write operations.

The Volatile pragmas force direct read/writes to memory.
package Ada.Task_Identification is
    type Task_Id is private;
    Null_Task_Id : constant Task_Id;
    function "=" (Left, Right: Task_Id) return Boolean;
    function Image (T: Task_Id) return String;
    function Current_Task return Task_Id;
    procedure Abort_Task (T : in out Task_Id);

    function Is_Terminated(T : Task_ID) return Boolean;
    function Is_Callable (T : Task_ID) return Boolean;

private
    ... -- not specified by the language
end Ada.Task_Identification;

Image returns an implementation-defined string that identifies a task
Current_Task returns a value that identifies the task
Task Attributes

with Ada.Task_Identification;
generic
  type Attribute is private;
  Initial_Value : Attribute;
package Ada.Task_Attributes is
  type Attribute_Handle is access all Attribute;

  function Value
      (T: Task_Identification.Task_Id :=
       Task_Identification.Current_Task)
      return Attribute;

  function Reference
      (T : Task_Identification.Task_Id :=
       Task_Identification.Current_Task)
      return Attribute_Handle;

  procedure Set_Value   (Val : Attribute;
                        T : Task_Identification.Task_Id :=
                        Task_Identification.Current_Task);

  procedure Reinitialize
      (T : Task_Identification.Task_Id :=
       Task_Identification.Current_Task);

end Ada.Task_Attributes;
Real-Time Annex

Specifies additional characteristics of Ada implementations intended for real-time systems software.

To conform to this annex, an implementation must also conform to the Systems Programming Annex.

Most of this annex consists of documentation requirements. An implementation must document the values of the annex-defined metrics for at least one hardware/system configuration.
Task Priorities

pragma Priority (expression);

pragma Interrupt_Priority (optional expression);

The range of System.Interrupt_Priority shall include at least one value.

The range of System.Priority must have at least 30 values.

Interrupt_Priority is defined as being greater than Priority.

The following declarations exist in package System

define

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;

Default_Interrupt_Priority : constant Interrupt_Priority := Interrupt_Priority'last;
Priority Scheduling

pragma Task_Dispatching_Policy (policy_identifier);

where FIFO_Within_Priorities is the only required policy. Other implementation-dependent policies may be defined.

An implementation must document:

- the maximum priority inversion a user task can experience;
- whether execution of a task can be preempted by the implementation processing of delay expirations for lower priority tasks (and, if so, for how long).
Priority Scheduling

The Ceiling_Locking policy (which specifies interactions between priority task scheduling and protected object ceilings) must be in effect for FIFO_Within_Priorities.

pragma Locking_Policy (policy_identifier);

where Ceiling_Locking is a predefined policy. Other policies may be implementation-defined.
Priority Ceiling Locking
An example WITHOUT Ceiling Locking

Three tasks

- P of priority 5
- Q of priority 3
- R of priority 1

Also, there is a protected object (O).

Task R is executing a procedure in O. P later requires access to the same procedure in O, but R must finish first. Q can preempt R.

P starts waiting

At this point, Q has blocked R and P

Q preempts R, priority 3

(R must complete exclusive write access to O before P can preempt R)
Priority Ceiling Locking

Solution - Have the protected object $O$ automatically execute at a “ceiling”.

Every protected object has a ceiling priority (set by either Priority or Interrupt_Priority pragma).

When a task executes a protected operation, it inherits the ceiling priority of the corresponding protected object.

If the active priority of the task is higher than the ceiling of the protected operation, a Program_Error is raised.
Expiration of Time Delay and elective Accepts

If two or more selective accepts are present with the different priorities, then the highest priority is executed.

If two or more expired delays or selective accepts are present with the same priority, the first in textual order is executed / selected.
Entry Queuing Policies

This specifies how the calls to a single entry point are queued up.

    pragma Queuing_Policy (policy_identifier);

where FIFO_Queuing and Priority_Queuing are predefined. Other implementation-defined policies may exist.

FIFO_Queuing is the default.
Dynamic Priorities

Allow the priority of a task to be modified or queried at run time

```ada
with System;
with Ada.Task_Identification; -- See G.6.1
package Ada.Dynamic_Priorities is

   procedure Set_Priority(Priority : System.Any_Priority;
                          T : Ada.Task_Identification.Task_Id :=
                              Ada.Task_Identification.Current_Task);

   function Get_Priority (T : Ada.Task_Identification.Task_Id :=
                           return System.Any_Priority;

private
   ... -- not specified by the language
end Ada.Dynamic_Priorities;
```
Preemptive Abort

Implementations must document

Execution time (in processor clock cycles) that it takes for an abort_statement to cause completion

• On multiprocessors, the upper bound (in seconds) on the time that the completion of an aborted task can be delayed beyond the point that is required for a single processor

• An upper bound on the execution time of an asynchronous_select
Tasking Restrictions

- No_Task_Hierarchy
- No_Nested_Finalization
- No_Abort_Statement
- No_Terminate_Alternatives
- No_Task_Allocators
- No_Implicit_Heap_Allocation
- No_Dynamic_Priorities
- No_Asynchronous_Control
- Max_Select_Alternatives
- Max_Task_Entries
- Max_Protected_Entries
- Max_Storage_At_Blocking
- Max_Asynchronous_Select_Nesting
- Max_Tasks

The above are restrictions that are language-defined for use with the pragma Restrictions
Monotonic Time

This clause specifies a high-resolution, monotonic clock package

package Ada.Real_Time is

  type Time is private;
  Time_First: constant Time;
  Time_Last: constant Time;
  Time_Unit: constant := implementation_defined_real_number;

  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;
  function Clock return Time;
type Seconds_Count is range implementation-defined;

procedure Split (T : in Time; SC: out Seconds_Count;
    TS : out Time_Span);

function Time_Of (SC: Seconds_Count; TS: Time_Span)
    return Time;

private
... -- not specified by the language
end Ada.Real_Time;
Monotonic Time Limits

The range of Time shall be sufficient to represent real ranges up to 50 years later.

Tick shall be no greater than 1 millisecond.

Time_Unit shall be less than or equal to 20 microseconds.

Time_Span_First shall be no Greater than -3600 seconds and Time_Span_Last no less than 3600 seconds.

The actual values of Time_First, Time_Last, Time_Span_First, Time_Span_Last, Time_Span_Unit and Tick shall be documented.
Delay Accuracy

An implementation shall document the following

• An upper bound on the execution time (in processor clock cycles) of a delay_relative_statement whose requested values is less than or equal to zero.

• An upper bound of the execution time of a delay_until_statement whose requested value of the delay expression is less than or equal to the value of the Real_Time.Clock and Calendar.Clock.

• An upper bound on the lateness of a delay_relative_statement for a positive values of the delay (and delay_until_statement), in a situation where the task has sufficient priority to preempt the processor as soon as it becomes ready.
Synchronous Task Control

Describes a language-defined private semaphore (suspension object)

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;

• An object of type Suspension_Object has two states: True and False
• Set_True and Set_False are atomic with respect to each other
• Suspend_Until_True blocks the calling task until the state is True,
  Program_Error is raised if another task is already waiting
• Current_State returns the current state of the object.
This clause introduces a language-defined package to do asynchronous suspend/resume on tasks.

```ada
with Ada.Task_Identification;
package Ada.Asynchronous_Task_Control is
  procedure Hold(T : Ada.Task_Identification.Task_Id);
  procedure Continue(T : Ada.Task_Identification.Task_Id);
  function Is_Held(T : Ada.Task_Identification.Task_Id)
    return Boolean;
private
  ... -- not specified by the language
end Ada.Asynchronous_Task_Control;
```
Asynchronous Task Control

- After the Hold operation, the task becomes “held”. There is a conceptual “idle task” whose priority is below System.Any_Priority’First. The held task is set to a “held priority” below the “idle task”.

- For a held task, it’s base priority no longer constitutes an inheritance source. Instead, the “held priority” is the new inheritance source.

- A Continue operation resets the state to not-held, and the priority is now reevaluated.
More than just the Source Code must be Certified

Ada System
(Program)

Source Code

Compiler

Run-time System

Linker

Ada Library and Runtime code

This is Certified and is Safety Critical
Lack of Experience

Lack of experience in Ada programming causes poor code performance.

Lack of experience in “C/C++” causes code errors.
Low Level Features

- Importing & Exporting other languages
- Callbacks
- Representation Clauses
Standard Interfaces

- **pragma Import**: used to import a foreign language into Ada.
- **pragma Export**: used to export an Ada entity to a foreign language.
- **pragma Convention**: use the convention of another language.
Standard Interfaces

The following packages are REQUIRED by the standard:

• package Interface.C  -- interface to C

• package Interface.COBOL  -- interface for COBOL

• package Interface.FORTRAN  -- interface for FORTRAN
“Nothing is more difficult to carry out, nor more doubtful of success, nor more dangerous to handle, than to initiate a new order of things. For the reformer has enemies in all those who profit by the old order, and only lukewarm defenders in all those who would profit by the new order, this luke-warmness arising partly from... the incredulity of mankind, who do not truly believe in anything new until they have had actual experience in it”.

-- Niccolo Machiavelli, from The Prince
How Not To Do Systems Engineering And The Sinking Of The Largest Offshore Oil Platform

March 2001

Disclaimer: Slides Received From Unknown Author
For those of you who may be involved in the engineering of systems
Please read this quote from a Petrobras executive,
extolling the benefits of cutting quality assurance and inspection costs,
on the project that sunk into the Atlantic Ocean off the coast of Brazil in March 2001.
"Petrobras has established new global benchmarks for the generation of exceptional shareholder wealth"
through an aggressive and innovative programme of cost cutting on its P36 production facility.
Conventional constraints have been successfully challenged
and replaced with new paradigms appropriate to the globalised corporate market place.
Through an integrated network of facilitated workshops,
the project successfully rejected the established constricting and negative influences of prescriptive engineering,
onerous quality requirements, and outdated concepts of inspection and client control.
Elimination of these unnecessary straitjackets has empowered the project's suppliers and contractors to propose highly economical solutions,
with the win-win bonus of enhanced profitability margins for themselves.
The P36 platform shows the shape of things to come
in unregulated global market economy of the 21st Century.”
And now you have seen the final result of this proud achievement by Petrobras.
Ada Engineered Products (1)
LAMPS SH-60R ASW Helicopter
Ada Engineered Products (2)
Boeing 777 Commercial Aviation

- Airbus 320
- Airbus 330
- Airbus 340
- Beechjet 400A
- Beech Starship I
- Beriev BE-200
- Boeing 737
- Boeing 747
- Boeing 757
- Boeing 767
- Boeing 777
- Canadair Regional Jet
- Embraer CBA-123
- Embraer CBA-145
- Fokker F-100
- Ilyushin 96M
- LM Hercules
- Saab 2000
- Tupolev TU-204
CANAL+ TECHNOLOGIES is the world's leading provider of digital broadcasting and interactive TV software solutions. Its field-proven systems are being used by more than 20 different digital operators and over 4.5 million set-top boxes based on its technologies are currently deployed.
Ada Engineered Products (4)
Hertz Neverlost
Ada Engineered Products (5)
70’Kingcat M270 Luxury Power Catamaran
Inertial Confinement Fusion
192 Lasers (510 Meters Path)
1.8 megajoules
Tiny Target – 600 µm diameter
At Lawrence Livermore National Laboratory
Common Characteristics of Ada Applications

1. Reliability is a real concern
2. Control safety or mission critical applications
3. Control hard real-time or near real-time application
4. Reliability is a real concern
5. Control highly distributed systems
6. Control systems with multiple interfaces
7. Reliability is a real concern

Achieved via a sound systems engineering approach

With the Ada Language as a Key Technology
If Architects used programming languages ...
Design and Implementation with SE & Ada 95
Design(?) and Implementation without SE
The End