Le modèle de concurrence Ravenscar: principes et mise en œuvre
(Ravenscar: Principles and Implementation)

José F. Ruiz <ruiz@adacore.com>
Senior Software Engineer

Ecole d’été Temps Réel (ETR’09)
Paris, 3 Septembre, 2009
Outline of the presentation

• Short introduction to Ada 2005

• Ravenscar for embedded high-integrity real-time systems
  – Why
  – What

• Bare board run time for Ravenscar
  – Ideas
  – Architecture
  – Implementation details
  – Metrics
Ada for real-time systems

- **Concurrency**
  - Within the language
    - Avoid error-prone low-level constructions
  - Well-defined semantics for scheduling
  - Safe / efficient mutual exclusion
    - Avoidance of unbounded priority inversion
  - Ravenscar profile
    - Restricted set of tasking features amenable to schedulability analysis and certification

- **Asynchrony**
  - Asynchronous events / event handlers
    - Connection with interrupts
  - Asynchronous Transfer of Control
    - Timeout
    - Task termination
  - Preemptive task abortion
  - Asynchronous task control

- **Time**
  - Support for high-resolution monotonic clock and absolute and relative delays
New real-time features in Ada 2005

- The Ravenscar profile

- Task elaboration and finalization
  - Partition elaboration policy for high-integrity systems (atomic elaboration)
  - Task termination procedures

- Restriction pragmas
  - No_Relative_Delay, Max_Entry.Queue-Length,

- Time and clocks
  - Timing events
  - Execution time clocks
  - Execution time budgets
    - For task groups also

- Scheduling
  - New dispatching policies
    - Non-preemptive, round robin, Earliest Deadline First (EDF)
  - Dynamic ceiling priorities
  - Priority specific dispatching
Subsetting the language

- For developing safety-critical software a reliable subset of the language is chosen
  - Such as SPARK, MISRA C,...

- Ada allows you to:
  - Restrict individual features
    - `pragma Restrictions (No_Dynamic_Allocation, No_Dynamic_Priorities);`
  - Select a set of restrictions
    - `pragma Profile (Ravenscar);`

- The Ada advantage
  - The compiler ensures compliance against the coding standard
  - The compiler may select a simpler run-time system
  - It ensures portability among different compilers
Motivation behind Ravenscar

- **Use concurrency in embedded real-time systems**
  - Verifiable
  - Simple
    - Implemented reliably and efficiently

- **Scheduling theory for accurate analysis of real-time behavior**
  - Preemptive fixed priority scheduling
    - Rate Monotonic Analysis (RMA)
    - Response Time Analysis (RTA)
  - Priority Ceiling Protocol (PCP)
    - Avoids unbounded priority inversion and deadlocks

- **Tool support**
  - RMA and RTA
  - Static simulation of concurrent real-time programs
Tasks in Ravenscar

Time-triggered tasks

```
task body Cyclic is
  Period : constant Time_Span := Seconds (1);
  Next_Activation : Time := Clock;
begin
  loop
    delay until Next_Activation;
    -- Do something
    Next_Activation := Next_Activation + Period;
  end loop;
end Cyclic;
```

Event-triggered task

```
task body Sporadic is
begin
  loop
    Monitor.Wait_Event; -- protected entry
    -- Do something
  end loop;
end Sporadic;
```
Synchronization in Ravenscar

**task body** Consumer is  
Element : Item;  
begin  
loop  
Buffer.Get (Element);  
Consume (Element);  
end loop;  
end Consumer;

**task body** Producer is  
Period : constant Time_Span := Seconds (1);  
Next_Activation : Time := Clock;  
Element : Item;  
begin  
loop  
delay until Next_Activation;  
Element := Produce_Item;  
Buffer.Put (Element);  
Next_Activation := Next_Activation + Period;  
end loop;  
end Producer;

**protected body** Buffer is  
procedure Put (Element : Item) is  
begin  
Container := Element;  
Received := True;  
end Put;  
entry Get (Element : out Item) when Received is  
begin  
Element := Container;  
Received := False;  
end Get;  
end Buffer;
Interrupt support in Ravenscar

```
protected Monitor is
  pragma Priority (Interrupt_Priority'Last);

procedure Handler;
pragma Attach_Handler (Handler, Interrupt_ID);

entry Wait;
private
  Signaled : Boolean := False;
end Monitor;

task body Interrupt_Task is
begin
  loop
    Monitor.Wait;
    Handle_Interrupt;
  end loop;
end Interrupt_Task;
```

```
protected body Monitor is
  procedure Handler is
  begin
    Received := True;
  end Handler;

entry Wait when Signaled is
begin
  Signaled := False;
end Wait;
end Monitor;
```
What is the Ravenscar Profile?

• A subset of the Ada tasking model

• Defined to meet safety-critical real-time requirements
  – Determinism
  – Schedulability analysis
  – Memory-boundedness
  – Execution efficiency and small footprint
  – Suitability for certification

• State-of-the-art concurrency constructs
  – Adequate for most types of real-time software
Static existence model

- **Set of tasks / interrupts to be analyzed is fixed and has static properties**
  - Restrictions
    - `No_Task_Hierarchy`, `No_Task_Allocators`, `No_Task_Termination`, `No_Dynamic_Attachment`, `No_Dynamic_Priorities`, `No_Abort_Statement`

- **Tasks, protected objects only at library level**
  - No dynamic allocation of tasks or protected objects

- **Each task is infinite loop**
  - Single “triggering” action (delay or event)

- **Fixed priority**
  - Simple static analysis
    - Schedulability analysis

- **Tasks cannot be aborted**
  - No changes to the set of tasks
  - Reduces size and complexity of the run time system
  - Reduces non-determinacy
Static synchronization and communication model

- **Static set of protected objects**
  - Restrictions
    - No_Local_Protected_Objects, No.Protected_Types.Allocators

- **No rendezvous**
  - Restrictions
    - No.Select.Statements, Max.Task.Entries => 0
  - Synchronize using protected objects and suspension objects
    - Deterministic

- **Required for schedulability analysis**
Deterministic memory usage

- **Static set of tasks and protected objects**
  - Tasks descriptors and stacks are statically created at compile time

- **No implicit heap usage**
  - Restriction *No_Implicit_Heap_Allocations*
  - Explicit use of dynamic memory

- **No dynamically created task attributes**
  - Restriction *No_Task_Attributes_Package*
Deterministic execution model (I)

- **At most one entry per protected object**
  - Restriction $Max\_Protected\_Entries \Rightarrow 1$
  - Avoids two or more barriers becoming open simultaneously
    - Non-determinism of selecting which entry should be serviced first

- **At most one task can be suspended waiting on a closed entry barrier for each protected object**
  - Restriction $Max\_Entry\_Queue\_Length \Rightarrow 1$
  - Avoids the possibility of queues of tasks on an entry
    - Non-determinism depending on the length of the waiting time in the queue

- **Simple barriers**
  - Restriction $Simple\_Barriers$
  - The Boolean variable must be declared immediately within the protected object
  - Deterministic execution time
  - Absence of side effects for the evaluation of entry barriers

- **No requeue**
  - Restriction $No\_Requeue\_Statements$
  - Deterministic task release from protected entry barriers
Deterministic execution model (II)

• Task cannot be stopped from the outside
  – Restrictions
    – No_Asynchronous_Control, No_Abort_Statements
  – Time deterministic

• Deterministic time handling
  – Restrictions
    – No_Relative_Delay, No_Calendar
  – Delay until is deterministic and should be used for accurate release of time-triggered tasks
  – Use high precision timing

• Task creation and activation is very simple and deterministic
  – Tasks created at initialization, then activated and executed according to their priority
The Ravenscar Tasking Model

- A single processor
- A fixed number of tasks
- Single invocation event per task
  - Time-triggered or event-triggered
- Task interaction using shared data
  - Mutual exclusive access
- Remove constructions difficult to analyse
  - No asynchronous control, no abort, ...

Most violations detected at compile time
The Ravenscar Tasking Model (II)

- **Scheduling policy**
  - Preemptive fixed-priorities

- **Locking policy**
  - Ceiling priority for bounding priority inversion

- **Remove non-deterministic constructions**
  - No relative delays, no task termination, no abort, ...

Supports sound real-time development techniques, such as Rate Monotonic Analysis and Response Time Analysis
Tasks

- **Fixed set of tasks**
  - Only at library level
  - No dynamic allocation
  - No nested declaration of tasks
  - Statically created
    - Task descriptors, stacks, ...

- **Each task is infinite loop**
  - Single “triggering” action (delay or event)

- **Task creation and activation is very simple and deterministic**
  - All tasks are created at initialization
  - Then all activated and executed according to their priority
Protected objects and interrupt handling

• Simple protected operations
  – No queuing
  – Ceiling locking on monoprocessor
    – Bounded priority inversion
    – Efficient locking/unlocking by increasing/decreasing priority
  – “Proxy model” for protected entries
    – Avoid unneeded context switches

• Interrupt handling
  – Simple and efficient
    – Protected procedures as low level interrupt handlers
    – Masking hardware interrupts according to active priority
  – Low interrupt latency
    – Allow interrupt nesting
    – Dedicated interrupt stacks
Context

- **ESA/ESTEC project**
  - Ada cross development environment for ERC32 / LEON
  - Compliant with the Ravenscar profile
    - Reliable tasking
    - State-of-the-art high level abstract development methods

- **Consortium**
  - AdaCore as prime contractor
  - DIT/UPM as external advisor

- **Building on previous Open Ravenscar Real-Time Kernel (ORK) project**
  - Proof-of-concept
Run-time architecture

• Use a safe Ada subset

• Complete run time
  – Layered architecture
  – Including multitasking core
    – No underlying kernel or operating system

• Isolate target dependencies
  – Allow easy retargeting

• Configuration support
  – Hardware and run-time parameters
Design issues

- Code in Ada as far as possible

- Take advantage of Ravenscar restrictions to simplify the implementation

- Careful implementation of kernel primitives in order to provide effective timing analysis

- Use a safe Ada subset to provide an adequate level of safety

- Isolate target dependencies to allow easy retargeting

- Include configuration support for hardware and kernel parameters
Functionality of the multitasking core

- Task management
- Task synchronisation
- Time-keeping and delays
- Ada interrupt support

 ERC32 / LEON

<table>
<thead>
<tr>
<th>Multitasking Core</th>
<th>Task management</th>
<th>Synchronisation</th>
<th>Time</th>
<th>Ada interrupt support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restricted GNARL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Restricted GNNULL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Architecture of the multitasking executive

- Kernel
- Parameters
  - CPU_Primitives
  - Peripherals
  - Serial_Output

- Thread Management
- Synchronization
- Scheduling
- Interrupt Handling
- Time Keeping and Delays
High Integrity Ravenscar

- **Ravenscar Ada tasking**
  - Superset of the Zero FootPrint run time
  - No other sequential Ada feature added

- **Simple**
  - Footprint for simple tasking program is 10KB

- **Static tasking model**
  - Tasks descriptors and stacks are statically created at compile time
  - Task creation and activation is very simple
    - All tasks are created at initialization
    - Then all activated and executed according to their priority

- **Simple protected operations**
  - No queuing
  - Locking/unlocking by increasing/decreasing priority

- **Complex features removed**
  - Such as exception handling and propagation
Multitasking core for Ravenscar

• **Mostly in Ada**
  - Reduced, simple, and safe subset of Ada (ISO 15942)
  - Some assembly
    - Context switches and trap handling

• **BSP for increasing portability**

• **Implements**
  - Preemptive fixed priority scheduling
  - Synchronization
  - Interrupt handling
  - Clock and timers
  - Target console
    - Using UART port (another UART port for the debugger)
High Integrity Ravenscar: Metrics

- **Linux kernel 2.6.28**
  - 10 millions lines of code
  - Average cyclomatic complexity: 6

Line metrics summed over 90 units
- code lines: 4186

Element metrics summed over 90 units
- all statements: 737
- all declarations: 2036
- logical SLOC: 2773

134 public types in 30 units including
  - 16 private types

166 type declarations in 38 units

246 public subprograms in 41 units

181 subprogram bodies in 32 units

Average cyclomatic complexity: 1.98
Static tasking model

• Tasks descriptors and stacks are statically created at compile time
  – No usage of dynamic memory

• Task creation and activation is very simple and deterministic
  – All tasks are created at initialization
  – Then all activated and executed according to their priority
Context switches and time management

- **Context switching**
  - Save/restore only the windows used
  - Floating point context only when needed
    - Only floating point tasks may suffer this overhead

- **Time management**
  - Monotonic clock
    - clock ticks (64-bit) from system start-up
      - Use one 32-bit (or 24-bit) timer (time stamp counter)
      - Upper 32-bit (or 40-bit) in memory
  - Efficient absolute delays
    - alarm-clock model
    - Use the other timer
    - delayed queue ordered by wake-up time
    - no cancellation
type Timers_Counter is mod 2 ** 24;
for Timers_Counter'Size use 24; -- Timer counters are 24-bit

type Reserved_8 is array (0 .. 7) of Boolean;
for Reserved_8'Size use 8;
pragma Pack (Reserved_8);

Timer_1_Counter_Register_Address : constant System.Address := System'To_Address (16#80000040#);

type Timer_Register is
  record
    Timer_Value : Timers_Counter;
    Reserved    : Reserved_8;
  end record;
for Timer_Register use
  record
    Timer_Value at 0 range 8 .. 31;
    Reserved   at 0 range 0 .. 7;
  end record;
for Timer_Register'Size use 32;

Timer_1_Counter : Timer_Register;
pragma Atomic (Timer_1_Counter);
for Timer_1_Counter'Address use Timer_1_Counter_Register_Address;
Conclusions

• Ada 2005 Ravenscar Profile allows for safe tasking
  - High-level abstractions, but ...
    - Deterministic
    - Time analyzable

• It can be implemented with a compact and efficient run-time system
  - Simple
  - Portable
  - Mostly written in Ada

• Ravenscar for multiprocessors
  - Under discussion (Ravenscar on each processor)