Real Time Programming with Ada

Part 2: Real time facilities

Real Time Programming: we need support for
- Concurrency (Ada tasking)
- Communication & synchronization (Ada Rendezvous)
- Consistency in data sharing (Ada protected data type)
- Real time facilities (Ada real time packages and delay statements)
  - accessing system time so that the passage of time can be measured
  - delaying processes until some future time
  - Timeouts: waiting for or running some action for a given time period

System Time

A timer circuit programmed to interrupt the processor at fixed rate.
- To approximate the universal time
- For distributed systems, we need clock synchronization
Each time interrupt is called a system tick (time resolution):
- Normally, the tick can vary 1-50ms, even microseconds in RTOS
  - Linux: 1ms
  - Linux 2.4, 10ms (100HZ), Linux 2.6, 1ms (1000HZ)
- The tick may be selected by the user
- All time parameters for tasks should be the multiple of the tick
- System time = 32 bits
  - One tick = 1ms: your system can run 50 days
  - One tick = 20ms: your system can run 1000 days = 2.5 years
  - One tick = 50ms: your system can run 2500 days = 7 years
- In Ada95, it is required that the system time should last at least 50 years

Real-Time Support in Ada

- Two pre-defined packages to access the system clock
  - Ada.Calendar and Ada.Real_Time
  - Both based on the same hardware clock
- There are two delay-statements
  - Delay time_expression (in seconds)
  - Delay until time_expression
- The delay statements can be used together with select to program timeouts, timed entry etc.

Package calendar in Ada: specification

```ada
package Ada.Calendar is
  type Time is private;
  function Clock return Time;
  function Year (Date : Time) return Year_Number;
  function Month (Date : Time) return Month_Number;
  function Day (Date : Time) return Day_Number;
  function Seconds (Date : Time) return Day_Duration;
  function Time_Of (Year : Year_Number; Month : Month_Number; Day : Day_Number; Seconds : Day_Duration := 0.0) return Time;
  function "+" (Left : Time; Right : Duration) return Time;
  function "*" (Left : Duration; Right : Time) return Time;
  function "-" (Left : Time; Right : Duration) return Time;
  function "-" (Left : Time; Right : Time) return Duration;
  function "<" (Left, Right : Time) return Boolean;
  function "<=" (Left, Right : Time) return Boolean;
  function ">" (Left, Right : Time) return Boolean;
  function ">=" (Left, Right : Time) return Boolean;
  function "exception" (Time_Error : exception := exception); private
    -- not specified by the language
    -- implementation dependent
  end Ada.Calendar;
```

Package calendar in Ada: specification (ctn.)

```ada
function "*" (Left : Time; Right : Duration) return Time;
function "*" (Left : Duration; Right : Time) return Time;
function "-" (Left : Time; Right : Duration) return Time;
function "-" (Left : Time; Right : Time) return Duration;
function "-" (Left : Time; Right : Time) return Boolean;
function "<" (Left, Right : Time) return Boolean;
function ">" (Left, Right : Time) return Boolean;
function "=" (Left, Right : Time) return Boolean;
Time_Error : exception := exception; private
  -- not specified by the language
  -- implementation dependent
  end Ada.Calendar;
```
### Package Real_Time in Ada: specification

```ada
package Ada.Real_Time is
  type Time is private;
  constant Time_First : Time;
  constant Time_Last : Time;
  constant Time_Unit : real;
  type Time_Span is private;
  constant Time_Span_First : Time_Span;
  constant Time_Span_Last : Time_Span;
  constant Time_Span_Zero : Time_Span;
  constant Tick : Time_Span;
  function Clock return Time;
  function + (Left, Right : Time) return Time;
  function + (Left : Time, Right : Time_Span) return Time;
  function - (Left, Right : Time) return Time_Span;
  function - (Left : Time, Right : Time_Span) return Time;
  function < (Left, Right : Time) return Boolean;
  function <= (Left, Right : Time) return Boolean;
  function > (Left, Right : Time) return Boolean;
  function >= (Left, Right : Time) return Boolean;
  function To_Duration (TS : Time_Span) return Duration;
  function To_Time_Span (D : Duration) return Time_Span;
  function Nanoseconds (NS : Integer) return Time_Span;
  function Microseconds (US : Integer) return Time_Span;
  function Milliseconds (MS : Integer) return Time_Span;
  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;
```

### Relative Delays

- **Delay the execution of a task for a given period**

  ```ada```
  ```
  Type: use Clock;
  loop
  exit when (Clock - Start) > 10.0; -- busy waiting
  end loop;
  ```

- **To avoid busy-waiting, most languages and OS provide some form of delay primitive**
  - In Ada, this is a delay statement `delay 10.0`
  - In UNIX, `sleep(10)`

### Absolute Delays

- **To delay the execution of a task to a specified time point (using clock access):**

  ```ada```
  ```
  START := Clock;
  FIRST_ACTION;
  loop
  exit when Clock >= Start+10.0; -- busy waiting
  end loop;
  SECOND_ACTION;
  ```

- **To avoid busy-wait (access “clock” all time every tick):**

  ```ada```
  ```
  START := Clock;
  FIRST_ACTION;
  delay until START + 10.0; -- this is by interrupt
  SECOND_ACTION;
  ```

- **As with delay, delay until is accurate only in its lower bound**
Absolute Delays: Example

```
task Ticket_Agent is
  entry Registration(...);
end Ticket_Agent;
task body Ticket_Agent is
  -- declarations
  Shop_Open : Boolean := True;
  begin
    while Shop_Open loop
      select
        accept Registration(...) do
          -- log details
          end Registration;
      or
        delay until Closing_Time;
        Shop_Open := False;
      end select;
    end loop;
end Ticket_Agent;
```

Periodic Task

```
task body Periodic_T is
  Next_Release : Time;
  ReleaseInterval : Duration := 10
  begin
    Next_Release := Clock + ReleaseInterval;
    loop
      -- Action
      delay until Next_Release;
      Next_Release := Next_Release + ReleaseInterval;
    end loop;
  end Periodic_T;
```

Control Example I

```
with Ada.Real_Time; use Ada.Real_Time;
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;

procedure Controller is
  task Temp_Controller;
  task Pressure_Controller;
```

Control Example II

```
task body Temp_Controller is
  TR : Temp_Reading; HS : Heater_Setting;
  Next : Time;
  Interval : Time_Span := Milliseconds(30);
  begin
    Next := Clock;
    loop
      Read(TR);
      Temp_Convert(TR,HS);
      Write(HS);
      Write(TR);
      Next := Next + Interval;
      delay until Next;
    end loop;
  end Temp_Controller;
```

Control Example III

```
task body Pressure_Controller is
  PR : Pressure_Reading; PS : Pressure_Setting;
  Next : Time;
  Interval : Time_Span := Milliseconds(70);
  begin
    Next := Clock;
    loop
      Read(PR);
      Pressure_Convert(PR,PS);
      Write(PS);
      Write(PR);
      Next := Next + Interval;
      delay until Next;
    end loop;
  end Pressure_Controller;
  begin
    null;
    end Controller;
```

Control Example III

```
task body Pressure_Controller is
  PR : Pressure_Reading; PS : Pressure_Setting;
  Next : Time;
  Interval : Time_Span := Milliseconds(70);
  begin
    Next := Clock;
    loop
      Read(PR);
      Pressure_Convert(PR,PS);
      Write(PS);
      Write(PR);
      Next := Next + Interval;
      delay until Next;
    end loop;
  end Pressure_Controller;
  begin
    null;
    end Controller;
```

Here Temp_Controller & Pressure_Controller start concurrently

Will run on average every 10 seconds
local drift only

If Action takes 11 seconds, the delay statement will have no effect
Programming Timeouts

Timeout and message passing

```
loop
  select
    accept Call(T : temperature) do
      New_temp:=T;
    end Call;
  or
    delay 10.0; --action for timeout
  end select;
  --other actions
end loop;
```

Timeout (by server)

```
task Server is
  entry Call(T : in Temperature);
  -- other entries
end Server;

task body Server is
  -- declarations
begin
  loop
    select
      accept Call(T : in Temperature) do
        New_Temp := T;
      end Call;
    or
      delay 10.0; -- action for timeout
    end select;
    -- other actions
  end loop;
end Server;
```

Timeout (by client)

```
loop
  -- get new temperature T
  Server.Call(T);
end loop;
```

```
loop
  -- get new temperature T
  select
    Server.Call(T);
  or
    delay 0.5; -- other actions
  end select;
end loop;
```

Timeouts on Entries

- The above examples have used timeouts on inter-task communication; it is also possible, within Ada, to do timed (and conditional) entry call on protected objects

```
select
  P.E ; -- E is an entry in protected object P
or
  delay 0.5;
end select;
```

Timeouts on Actions

```
select
  delay 0.1;
then abort
  -- action
end select;
```

- If the action takes too long, the triggering event will be taken and the action will be aborted
- This is clearly an effective way of catching runaway code --- Watchdog
SUMMARY: Language support for RT Programming

- Concurrency: multi-tasking
- Communication & synchronization
- Consistency in data sharing (protected data types)
- Real time facilities
  - Access to system clock/time
  - Delay constructs: Delay(10) and Delay until next time
  - Timely execution of tasks (run-time system)

The “core” of RT Programming Languages

- Primitive Types
  - Basic Types: e.g. integers, reals, lists, ...
  - Abstract data type: Semaphore
    - P(S)
    - V(S)
- Assignment: X := E
- Control Statements: if, While, ..., goto
- Sequential composition: P;P
- Concurrent composition: P || P
- Communication: a!e, a?x
- Choice: P or P
- Clock reading: Time
- Delays: Delay(n), Delay until n
- Exception: Loop P until B

RT Programming Languages

- “Classic” high-level languages with RT extensions e.g.
  - Ada
  - Real-Time Java, C + RTOS
- SDL, Soft RT language for telecommunication systems
- Synchronous Programming (from 1980’s)
  - Esterel (Gerard Berry)
  - Lustre (Caspi and Halbwachs)
  - Signal (le Guernic and Benveniste)
- Design, Modeling, Validation, and Code Generation
  (from models to code)
  - Giotto (Henzinger et al, not quite synchronous)
  - UPPAAL/TIMES (Uppsala)
  - Real-Time UML
  - Simulink

The History of Computer Science: Lifting the Level of Abstraction

Compilation: perhaps “the” success story of computer science

High-level languages: Programming with focus on the application

The “assembly age”: Programming with focus on the platform

Compilation: perhaps “the” success story of computer science

High-level languages: Programming with focus on the application

The “assembly age”: Programming with focus on the platform

Code generation from specifications: still mostly a dream

Compilation: perhaps “the” success story of computer science
Future Goal in Real-Time Software Development

- Efficient code (scheduled by RTOS)
  - Hardware
  - Esterel

- Code generation difficult
  - Giotto

- Code verification difficult
  - UML-based tools

- Mathematical models (e.g. Simulink in Matlab)
  - Hardware (different platforms)