Übung 4

Aufgabe 1 *Prozesssteuerung mit Zeitbedingungen in Ada*

Das "einfache eingebettete" System aus der Vorlesung (vgl. S. 2-41 ff, Übung 2/Aufgabe 4 und Übung 3/Aufgabe 1) soll als Echtzeitsystem realisiert werden und die folgenden Zeitbedingungen erfüllen:

1. Die Steuerung der Heizung soll als periodischer Prozess mit der Periode 1s realisiert werden. D.h. innerhalb einer Sekunde muss ein Messwert gelesen, eine Stellwert berechnet und ausgegeben sein.
2. Bei der Eingabe der Temperatur ist ein Eingabeflattern von höchstens 5ms zu berücksichtigen. Bei der Ausgabe des Heizungs-Stellwerts ist ein Ausgabeflattern von 10ms zu berücksichtigen.
3. Die Steuerung des Drucks soll als periodischer Prozess mit einer Periode von 2s realisiert werden.
4. Ein-/Ausgabeflattern beim Druckprozess muss wie beim Temperaturprozess berücksichtigt werden (5ms Eingabe, 10ms Ausgabe).
5. Für die Ausgaben der gemessenen Werte (Druck und Temperatur) gibt es keine Zeitbedingungen. Die Konsolenausgabe darf aber die Steuerungsprozesse nicht "ausbremsen", d.h. wenn durch die Konsolenausgabe eine Deadline in einem Steuerungsprozess (Temperatur und Druck) nicht eingehalten werden kann, soll die Konsolen-Ausgabe unterbleiben.

Lsg.1.: (Erweiterung der Lösung zu Aufgabe 3-1)

Das Package IO ist unverändert gegenüber 3-1 (keine Write-Prozeduren mehr für die Ausgabe auf Console, stattessen Konsole-Tasks mit entsprechenden Eintrittspunkten).

Da für die Konsolen-Ausgabe der Temperatur andere Zeitbedingungen gelten, als für die Konsolenausgabe des Drucks, muss die Konsole-Task in zwei nebenläufige Tasks gesplittet werden: `T_Console` und `P_Console` mit Periode 1s bzw. 2s

Da in den beiden Steuerungs-Tasks Messwerteingabe – Stellwertberechnung und Stellwertausgabe sequentiell erfolgt, spielt das Ein-Ausgabeflattern hier keine Rolle und wird deshalb nicht berücksichtigt (es ist z.B. unerheblich, ob man nach Messwerteingabe wartet, bis die 5s "Flatterzeit" um sind oder nicht, da die Stellwertberechnung sowieso erst nach der Eingabe beginnt).

In einer Lösung mit drei unabhängigen Tasks für Eingabe, Berechnung und Ausgabe wäre das anders: die Berechnungs-Task dürfte erst 5ms nach Periodenbeginn mit der Verarbeitung des Messwerts beginnen (siehe zweite Lösung unten).

```
with Data_Types; use Data_Types;
package IO is
  -- procedures for data exchange with the environment
  procedure Read(TR : out Temp_Reading); -- from DAC
  procedure Read(PR : out Pressure_Reading); -- from DAC
  procedure Write(HS : Heater_Setting);   -- to switch.
  procedure Write(PS : Pressure_Setting); -- to DAC
end IO;
```

Die folgende Prozedur Controller definiert die drei Tasks `Temp_Controler`, `Pressure_Controler` und zwei Konsolen-Tasks `T_Console` und `P_Console` und startet diese nebenläufig. `Temp_Controler` und `Pressure_Controler` kommunizieren mit `T_Console` bzw. `P_Console` über ein Rendezvous. Die Realisierung der Zeitbedingungen ist blau dargestellt.

```
with Data_Types; use Data_Types;
with io; use io;
with Control_Procedures; use Contol_Procedures;
with Ada.Real_Time; use Ada.Real_Time;
```

```
procedure Contoller is

  task Temp_Contoller;
  task Pressure_Contoller;

  -- Da für die Konsolenausgabe von Druck- und Temperaturwerten unterschiedliche
  -- Zeitbedinungen gegeben sind, müssen zwei unabhängige Tasks definiert werden
  task T_Console is -- Temperaturwerte auf der Konsole ausgeben
    entry Write_Temp(TR_temp : Temp_Reading); -- von Task Temp_Controller zu rufen
  end T_Console;

  task P_Console is -- Druckwerte auf der Konsole ausgeben
    entry Write_Pressure(PR_temp : Pressure_Reading);
                                          -- vom Task Pressure_Controller zu rufen
  end P_Console;


  task body Temp_Contoller is
    TR : Temp_Reading;
    HS : Heater_Setting;

    Next_Release : Time;
    Release_Interval : constant Time_Span := Milliseconds(1000); -- Periode 1s

  begin
    NextRelease := Clock + Release_Interval;
    loop
      Read(TR);
      Temp_Convert(TR,HS);
      Write(HS);
      T_Console.Call_Temp(TR); -- sendet den Temperaturwert an Task T_Console
      delay until Next_Release;
      Next_Release := Next_Release + Release_Interval;
    end loop;
  end Temp_Controller;

  task body Pressure_Controller is
    PR : Pressure_Reading;
    PS : Pressure_Setting;

    Next_Release : Time;
    Release_Interval : constant Time_Span := Milliseconds(2000); -- Periode 2s

  begin
    NextRelease := Clock + Release_Interval;
    loop
      Read(PR);
      Pressure_Convert(PR,PS);
      Write(PS);
      P_Console.Call_pressure(PR_temp); -- sendet den Druckwert an Task P_Console
      Next_Release := Next_Release + Release_Interval;
    end loop;
  end Pressure_Controller;

  task body T_Console is
    TR : Temp_Reading;

    Next_Release : Time;
    Release_Interval : constant Time_Span := Milliseconds(1000); -- Periode 1s

  begin

    NextRelease := Clock + Release_Interval;
```

```
    loop
      select
        delay Release_Interval; -- nach 1s nicht länger warten!
      then abort
        accept Call_Temp(TR : Temp_Reading) do
        -- TR ausgeben
        end Call_Temp;
      end select;
      Next_Release := Next_Release + Release_Interval;
    end loop;
  end Console;

  task body P_Console is
    PR : Pressure_Reading;

    Next_Release : Time;
    Release_Interval : constant Time_Span := Milliseconds(2000); -- Periode 2s

  begin
    NextRelease := Clock + Release_Interval;
    loop
      select
        delay Release_Interval; -- nach 2s nicht länger warten!
      then abort
        accept Call_Pressure(PR : Pressure_Reading) do
        -- TR ausgeben
        end Call_Temp;
      end select;
      Next_Release := Next_Release + Release_Interval;
    end loop;
  end Console;


begin
  null;       -- Die vier Tasks werden gestartet
end Controller;
```

Lsg.2: (Von Simon Bausch)

Hier werden das Lesen der Messwerte und das Schreiben der Stellwerte in regelmäßigen Abständen (entsprechend den Perioden und unter Berücksichtigung des möglichen E-/A-Flatterns) durch Zeitereignisse gesteuert (Analog zum Beispiel auf S. 5-47ff). Die Zeitsteuerung ist im Rahmen von geschützten Objekten zu realisieren: `pressure_reader` bzw. `pressure_writer` und `temp_reader` bzw. `temp_writer`.

Die Lese- und Schreib-Vorgänge müssen von der Steuerungstask durch Aufruf der Prozedur `start` gestartet werden und laufen dann zeitgesteuert automatisch ab. In periodischen Abständen werden dann Messwerte gelesen bzw. Stellwerte geschrieben. Der Zugriff auf gelesene Werte durch die Steuerungstask erfolgt durch einen Eintrittspunkt (`read_from`), der nur offen ist, wenn ein neuer Messwert vorliegt. Das ablegen eines neuen Stellwerts im geschützten Objekt ist ohne Einschränkung durch die Prozedur `write_to` möglich.

Hier Beispielhaft der Code für die Druck-Ein-/und Ausgabe. – Man beachte, dass nicht nur die Geschützte Typen definiert ist, sondern auch gleich zwei geschützte Objekte (**`pressure_sr`**, **`pressure_aw`**).

```
with io; use io;
with system; use system;
with data_types; use data_types;
with ada.real_time; use ada.real_time;
with ada.real_time.timing_events; use ada.real_time.timing_events;
```

```
package pressure_io is

      protected type pressure_reader is
            pragma interrupt_priority(interrupt_priority'last);
            procedure start;
            entry read_from(data: out pressure_reading);
            procedure timer(event: in out timing_event);
      private
            reading: pressure_reading;
            data_available: boolean := false;
            next_time: time;
      end pressure_reader;

      protected type pressure_writer is
            pragma interrupt_priority(interrupt_priority'last);
            procedure start;
            procedure write_to(data: pressure_setting);
            procedure timer(event: in out timing_event);
      private
            value: pressure_setting;
            data_available: boolean := false;
            next_time: time;
      end pressure_writer;

      pressure_sr: pressure_reader;
      pressure_aw: pressure_writer;

      pressure_period: constant time_span := milliseconds(2000);

private

with ...   use ...

package body pressure_io is

      protected body pressure_reader is
            procedure start is
            begin
                  read(reading);
                  next_time := clock + pressure_period;
                  data_available := true;
                  set_handler(pressure_input_jitter_control, next_time,
                              pressure_sr.timer'access);
            end start;
            entry read_from(data: out pressure_reading) when data_available is
            begin
                  data := reading;
                  data_available := false;
            end read_from;
            procedure timer(event: in out timing_event) is
            begin
                  read(reading);
                  next_time := next_time + pressure_period;
                  set_handler(pressure_input_jitter_control, next_time,
                              pressure_sr.timer'access);
            end timer;
      end pressure_reader;
```

```
            protected body pressure_writer is
            procedure start is
            begin
                  next_time := clock + pressure_period;
                  set_handler(pressure_output_jitter_control, next_time,
                              pressure_aw.timer'access);
            end start;
            procedure write_to(data: pressure_setting) is
            begin
                  value := data;
                  data_available := true;
            end write_to;
            procedure timer(event: in out timing_event) is
            begin
                  write(value);
                  next_time := next_time + pressure_period;
                  set_handler(pressure_output_jitter_control, next_time,
                              pressure_aw.timer'access);
            end timer;
      end pressure_writer;

end pressure_io;
```

Die Steuerungsprozedur startet nebenläufig die Druck-Steuerung (task `pressure_controller`) und Temperatursteuerung (Task `temp_controler`) Beide startet zunächst die Zeitsteuerung der Lese- und Schreibvorgänge in den geschützten Objekten zum jeweiligen Periodenbeginn, und führt dann unter Berücksichtigung des jeweiligen Ein-/Ausgabeflatterns den Zyklus Messwert lesen (aus dem geschützten Objekt) – Stellwert berechnen – Stellwert schreiben (in das geschützte Objekt) durch. – Hier ein Codeauszug, der die Task `pressure_controller` zeigt. Nebenläufig dazu wird die Task `temp_controler` gestartet, die analog strukturiert ist.

```
with ... use ...
procedure controller is

      task temp_controller;
      task body temp_controller is ...

      task pressure_controller;

      task body pressure_controller is
            pr: pressure_reading;
            ps: pressure_setting;
            period_end: time;
      begin
            --loop
            --    read(pr);
            --    pressure_convert(pr, ps);
            --    write(ps);
            --    write(pr);
            --end loop;
            pressure_sr.start;
            delay 2.0 - 0.01;  -- period - output jitter
            pressure_aw.start;
            period_end := clock + pressure_period;
            loop
                  -- If reading does not finish within 5 ms,
                  -- skip this read, thus using old value for
                  -- this iteration.
                  select
                        delay 0.005;
                  then abort
                        pressure_sr.read_from(pr);
                  end select;
```

```
                -- If computation of actuator setting does
                -- not finish within 1985 ms, abort it and
                -- use old setting for this iteration.
                select
                      -- period - input jitter - output jutter
                      delay 2.0 - 0.005 - 0.01;
                then abort
                      pressure_convert(pr, ps);
                end select;
                -- Writing to actuator and console must
                -- finish within 10 ms, starting with the
                -- actuator write (the more important one).
                --   * If we miss the output time window,
                --     pressure_aw.write_to() is aborted and
                --     pressure_aw will write the old setting
                --     for this iteration.
                --   * If we make actuator output, but
                --     there is no time left for console
                --     output (as the next iteration must
                --     now start), write() is aborted.
                select
                      delay 0.01;
                then abort
                      pressure_aw.write_to(ps);
                      write(pr);
                end select;
                -- If the read was aborted for waiting on
                -- the next sensor input, this iteration
                -- will finish in a few milliseconds, so
                -- wait for next iteration to start.
                delay until period_end;
                period_end := period_end + pressure_period;
          end loop;
      end pressure_controller;

begin
      null;
end controller;
```

Der vollständige Code ist zu finden als GPS-Projekt im Archiv `A4-1.zip` auf moodle.

Aufgabe 2 *Prozesssteuerung mit Zeitbedingungen in C/Real-Time POSIX*

Entwerfen Sie eine Lösung für die Prozesssteuerung als Echtzeitsystem mit den Bedingungen von Aufgabe 1 in C/Real-Time POSIX.

Aufgabe 3 *Zufahrtskontrolle zu einem Parkplatz*

Die Lösung zu Aufgabe 2, Übung 3, beinhaltet die Task Einfahrt für die Steuerung der Einfahrtschranke folgende Codesequenz, mit der festgestellt wird, ob das Fahrzeug nach Öffnen der Schranke den Schrankenbereich verlassen hat:

```
ED: EDurchfahrt;
...
ED = True;       -- Durchfahrt beginnt
while (ED = True) loop Read(ED); endloop; -- Durchfahren lassen
```

Analog beinhaltet in der Task Ausfahrt für die Steuerung der Ausfahrtschranke eine ähnliche Codesequenz:

```
   AD: ADurchfahrt;
   ...
   AD = True;      -- Durchfahrt beginnt
   while (AD = True) loop Read(AD); endloop; -- Durchfahren lassen
```

Verändern Sie die Lösung so, dass diese beiden Codesequenzen mit einer Deadline von 10s ausgeführt werden müssen. Wenn nach 10s das Fahrzeug den Schrankenbereich nicht verlassen hat (d.h. die while-Schleife beendet ist), soll das System in einen "Notfall-Modus" gehen.

Im Notfallmodus übernimmt die Signal-Steuerung die Kontrolle. D.h. das Signal wird auf Rot gesetzt, die anderen Tasks werden blockiert solange bis der Notfall-Modus wieder aufgehoben wird.

Der Notfallmodus wird kann "von außen" wieder aufgehoben werden. Sehen Sie hierfür einen geeigneten Eintrittspunkt der Signal-Steuerung vor, dem die aktuelle Anzahl der belegten Parkplätze übergeben wird. Wer diesen Eintrittspunkt aufruft, soll hier nicht berücksichtigt werden.

Nach Aufhebung des Notfallmodus sollen alle Schranken geschlossen werden und der Prozess läuft "normal" weiter (mit der übergebenen Anzahl von belegten Parkplätzen).

Lsg:

Die Packages `Data_Types` und `MyIO` sind dieselben wie in der Lösung zu Übung1, Aufgabe 5 (vgl. Übung1-lsg.pdf)

Erweiterungen gegenüber der Lösung von Aufgabe 3-2 sind blau gekennzeichnet.

```
with Data_Types; use Data_Types;
with MyIO; use MyIO;
procedure Main is

  task Signal is
    entry einfahrt();
    entry ausfahrt();
    entry setEmergency();
    entry releaseEmergency(pp:Integer);
  end;

  task Einfahrt;
  task body Einfahrt is
     EA : EAnfrage := False;
     EOpen : Einfahrt := Open;
     EClose : Einfahrt := Close;
     ED : EDurchfahrt := False;
   begin
     loop
       Read(EA);
       if (EA = True) then -- Wenn Anfrage Einfahrt
         Signal.einfahrt() ; -- Signal informieren, ggfs. warten
         Write(EOpen);   -- Einfahrt öffnen
         ED = True;      -- Durchfahrt beginnt
         select
           delay 10;
           Signal.setEmergency();
         then abort
           while (ED = True) loop Read(ED); endloop; -- Durchfahren lassen
           Write(EClose);  -- Einfahrt schließen
         end select;
       end if;
     end loop;
   end Einfahrt;
```

```
task Ausfahrt;
task body Ausfahrt is
  AA : AAnfrage  := False;
  AD : ADurchfahrt  := False;
  AOpen : Ausfahrt := Open;
  AClose : Ausfahrt := Close;
begin
  loop
    Read(AA);
    if (AA = true) then -- Wenn Anfrage Ausfahrt
      Signal.ausfahrt() - Signal informieren, ggfs. warten
      Write(AOpen);  -- Ausfahrt öffnen
      AD = True;     -- Durchfahrt beginnt
      select
        delay 10;
        Signal.setEmergency();
      then abort
        while (AD = True) loop Read(AD); endloop; -- Durchfahren lassen
        Write(AClose); -- Ausfahrt schließen
      end select;
    end if;
  end loop;
end Ausfahrt;

task body Signal is
  PP: Integer := 0;
  Emergency: Boolean := false;
  AClose : Ausfahrt := Close;
  EClose : Einfahrt := Close;
begin
  write(Free); -- Signal auf grün
  loop
    select
      when (PP < 50) and (not Emergency) =>
        accept einfahrt do
          PP := PP+1;
          if PP = 50 then write(Full); -- Signal auf Rot setzen
        end einfahrt
    or
      when not Emergency =>
        accept ausfahrt do
         if PP = 50 then write(Free); -- Signal auf Grün setzen
         PP := PP-1;
        end ausfahrt
    or
      when not Emergency =>
        accept setEmergency do
          write(Full); -- Signal auf rot
          Emergency = True;
        end setEmergency;
    or
      when Emergency =>
        accept releaseEmergency(PPnew) do
          PP := PPnew;
          if (PP < 50) write(Free);  -- ggfs. Signal auf grün
          write(AClose); write(EClose); -- Schranken schließen
        end releaseEmergency;
```

```
        or
           terminate; -- Falls Einfahrt und Ausfahrt nicht mehr aktiv sind
                      -- Wird auch die Signal-Task beendet.
       end select;
     end loop

  end Signal;

begin   null;  end Main; -- Einfahrt, Ausfahrt und Signal werden gestartet
```