CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

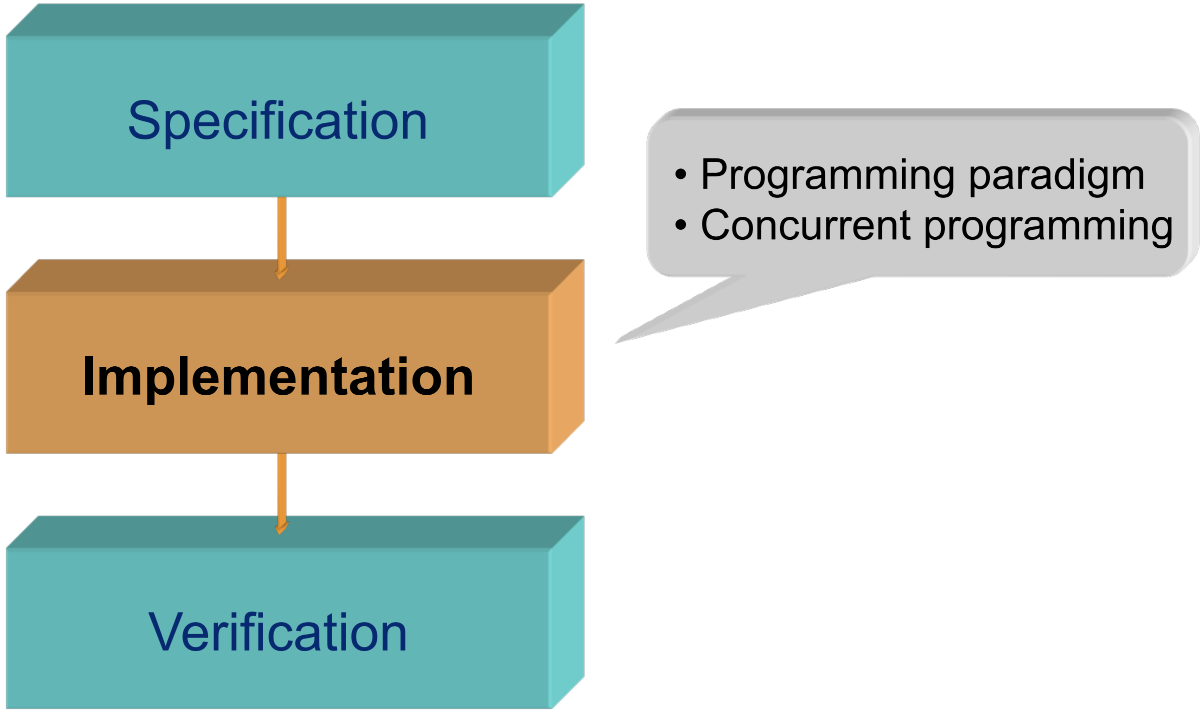
Real-Time Systems

Lecture #2

Professor Jan Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology

Real-time systems



Real-time programming

Recommended programming paradigm:

- Concurrent programming
  - Reduces unnecessary dependencies between tasks
  - Enables a composable schedulability analysis
- Reactive programming
  - Certifies that tasks are activated only when work should be done; tasks are kept idle otherwise
  - Maps directly to the task model used in schedulability analysis
- Timing-aware programming
  - Certifies that timing constraints are visible at the task level
  - Enables priority-based scheduling of tasks, which in turn facilitates schedulability analysis

Real-time programming

Desired properties of a real-time programming language:

- Support for partitioning software into units of concurrency
  - tasks or threads (Ada95, Java or POSIX C)
  - object methods (C/C++ using the TinyTimber kernel)
- Support for communication with the environment
  - access to I/O hardware (e.g. view I/O registers as variables)
  - machine-level data types (e.g. bit-field type, address pointers)
- Support for the schedulability analysis
  - notion of (high-resolution) time (⇒ timing-aware programming)
  - task priorities (reflects constraints ⇒ timing-aware programming)
  - task delays (idle while not doing useful work ⇒ reactive model)
  - hardware interrupt handlers (event generators ⇒ reactive model)

Real-time programming

What programming languages are suitable?

- C, C++
  - Support for machine-level programming
  - Concurrent programming via run-time system (POSIX, TinyTimber)
  - Priorities and notion of time via run-time system (POSIX, TinyTimber)
- Java
  - Support for machine-level programming
  - Support for concurrent programming (threads)
  - Support for priorities and notion of time (Real-Time Java)
- Ada 95
  - Support for machine-level programming
  - Support for concurrent programming (tasks)
  - Support for priorities and notion of time

Why concurrent programming?

Most real-time applications are inherently parallel

- Events in the target system's environment often occur in parallel
- By viewing the application as consisting of multiple tasks, this parallel reality can be reflected
- While a task is waiting for an event (e.g., I/O or access to a shared resource) other tasks may execute

Enables a composable schedulability analysis

- First, the local timing properties of each task are derived
- Then, the interference between tasks are analyzed

System can obtain reliability properties

- Redundant copies of the same task makes system fault-tolerant

Issues with concurrent programming

Access to shared resources

- Many hardware and software resources can only be used by one task at a time (e.g., processor, data structures)
- Only pseudo-parallel access is possible in many cases

Synchronization and information exchange

- System modeling using concurrent tasks also introduces a need for synchronization and information exchange.

Concurrent programming must hence be supported by an advanced run-time system that handles the scheduling of shared resources and communication between tasks.

Support for concurrent programming

Support in the programming language:

- Program is easier to read and comprehend, which means simpler program maintenance
- Program code can be easily moved to another operating system
- For some embedded systems, a full-fledged operating system is unnecessarily expensive and complicated
- Examples: Ada 95, Java, Modula, Occam, ...

Example:

Ada 95 offers support via task, rendezvous & protected objects

Java offers support via threads & synchronized methods

Support for concurrent programming

Support in the run-time system:

- Simpler to combine programs written in different languages whose concurrent programming models are incompatible
- There may not exist a simple one-to-one mapping between the language's model and the run-time system's model
- Operating systems become more and more standardized, which makes program code more portable between OS's
(e.g., POSIX for UNIX, Linux, Mac OS X, and Windows)

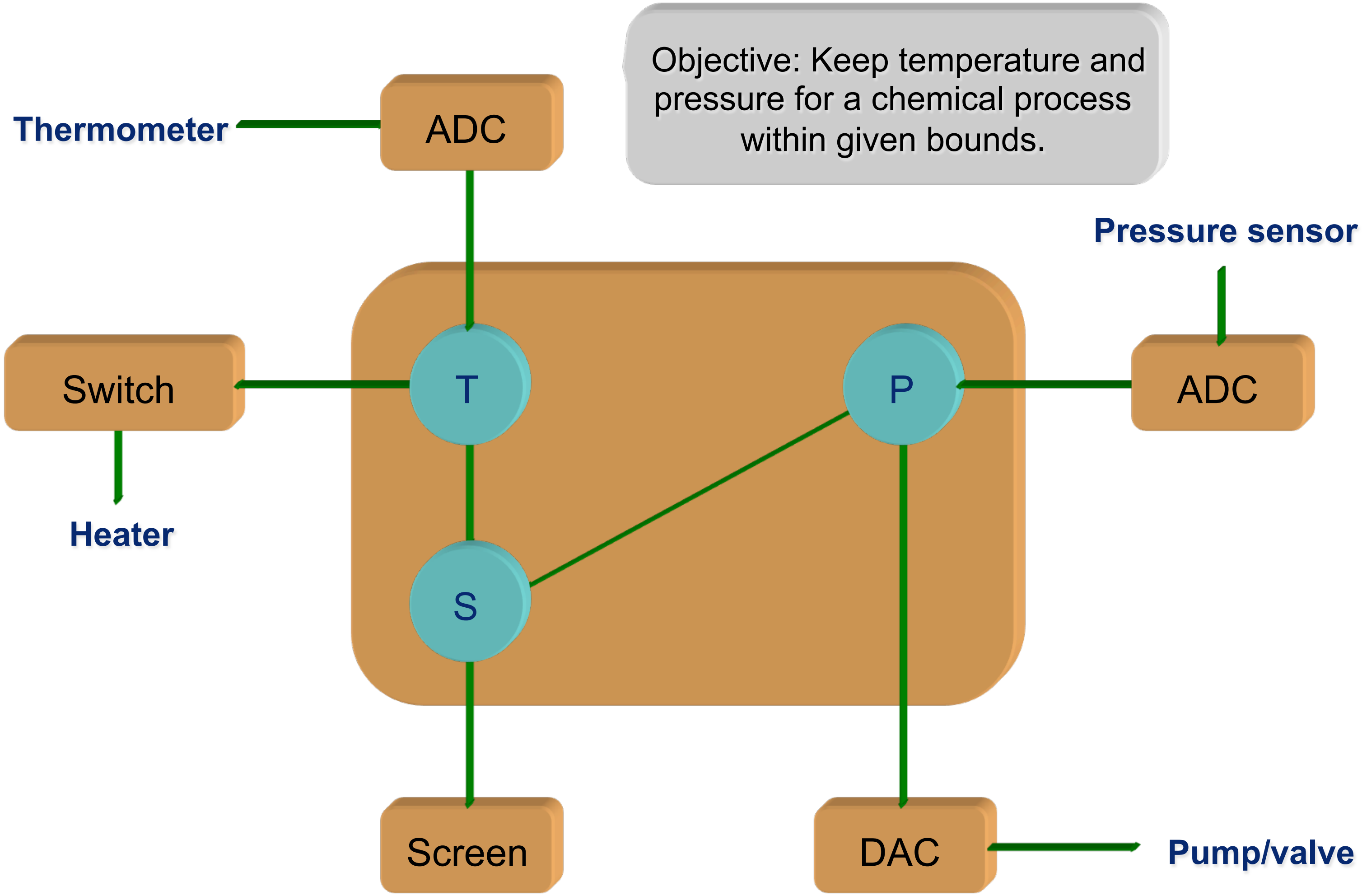
Example:

UNIX, Linux, etc offer support via fork, semctl & msgctl

POSIX offers support via threads & mutex methods

TinyTimber offers support via reactive objects & mutex methods

Example: a simple control system



Sequential solution (Ada95)

```
procedure Controller is
  TR : Temp_Reading;
  PR : Pressure_Reading;
  HS : Heater_Setting;
  PS : Pressure_Setting;
begin
  loop
    T_Read(TR);                     -- read temperature
    Temp_Convert(TR,HS);            -- convert to temperature setting
    T_Write(HS);                    -- to temperature switch
    PrintLine("Temperature: ", TR); -- to screen

    P_Read(PR);                     -- read pressure
    Pressure_Convert(PR,PS);        -- convert to pressure setting
    P_Write(PS);                    -- to pressure control
    PrintLine("Pressure: ", PR);    -- to screen
  end loop;
end Controller;
```

Sequential solution (C)

```
void Controller() {
  Temp_Reading TR;
  Pressure_Reading PR;
  Heater_Setting HS;
  Pressure_Setting PS;

  while (1) {
    T_Read(&TR);                       -- read temperature
    Temp_Convert(TR,&HS);              -- convert to heater setting
    T_Write(HS);                       -- set temperature switch
    PrintLine("Temperature: ", TR);    -- write to screen

    P_Read(&PR);                       -- read pressure
    Pressure_Convert(PR,&PS);          -- convert to pressure setting
    P_Write(PS);                       -- set pressure control
    PrintLine("Pressure: ", PR);       -- write to screen
  }
}
```

Sequential solution

Drawback:

- the inherent parallelism of the application is not exploited
  - Procedures `T_Read` and `P_Read` block the execution until a new temperature or pressure sample is available from the sensor
  - while waiting to read the temperature, no attention can be given to the pressure (and vice versa)
  - if the call for reading the temperature does not return because of a fault, it is no longer possible to read the pressure
- the independence of the control functions are not considered
  - temperature and pressure must be read with the same interval
  - the iteration frequency of the loop is mainly determined by the blocking time of the calls to `T_Read` and `P_Read`.

Improved sequential solution (C)

The Boolean function **Ready_Temp** indicates whether a sample from the sensor is available

```
void Controller() {
  ...;

  while (1) {
    if (Ready_Temp()) {
      T_Read(&TR);                   -- read temperature
      Temp_Convert(TR,&HS);          -- convert to heater setting
      T_Write(HS);                   -- set temperature switch
      PrintLine("Temperature: ", TR);
    }

    if (Ready_Pres()) {
      P_Read(&PR);                   -- read pressure
      Pressure_Convert(PR,&PS);      -- convert to pressure setting
      P_Write(PS);                   -- set pressure control
      PrintLine("Pressure: ", PR);
    }
  }
}
```

Improved sequential solution

Advantages:

- the inherent parallelism of the application is exploited
  - pressure and temperature control do not block each other

Drawbacks:

- the program spends a large amount of time in “busy wait” loops
  - processor capacity is unnecessarily wasted
  - schedulability analysis is made complicated/impossible
- the independence of the control functions is not considered
  - if the call for reading the temperature does not return because of a fault, it is no longer possible to read the pressure

Concurrent solution

Step 1: Make concurrent:

- Partition the software into units of concurrency

Ada95:
Create two units of type **task**, `T_Controller` and `P_Controller`, each containing the code for handling the data from respective sensor.

TinyTimber: First create two objects, `T_Obj` and `P_Obj`, each with one method (`T_Controller` and `P_Controller`) containing the code for handling the data from respective sensor. Then create two interrupt handlers, one for each sensor, that calls the respective object method when data becomes available.

Concurrent solution

Step 2: Make reactive:

- Tasks should be idle if there is no work to be done

  Ada95: Call the blocking procedures `T_Read` and `P_Read` to idle.

  TinyTimber: Since methods `T_Controller` and `P_Controller` must be called to be activated they are by default idle.

- Activate task as a reaction to an incoming event

  Ada95: A call to procedure `T_Read` or `P_Read` unblocks when data becomes available at a sensor, thus activating the calling task.

  TinyTimber: An interrupt handler calls (activates) its corresponding method when data becomes available at a sensor.

Concurrent solution (Ada95)

```
procedure Controller is
task T_Controller;
task P_Controller;

task body T_Controller is
begin
  loop
     T_Read(TR);
     Temp_Convert(TR,HS);
     T_Write(HS);
     PrintLine("Temperature: ", TR);
  end loop;
end T_Controller;

task body P_Controller is
begin
  loop
     P_Read(PR);
     Pressure_Convert(PR,PS);
     P_Write(PS);
     PrintLine("Pressure: ", PR);
  end loop;
end P_Controller;

begin
  null;        -- begin parallel execution
end Controller;
```

Concurrent solution (TinyTimber)

```
// Define two new objects of TinyTimber basic class Object

Object T_Obj = InitObject();
Object P_Obj = InitObject();

// Declare the methods for each new object

void T_Controller(Object*, int);
void P_Controller(Object*, int);

// Define two new objects of class Sensor (definition not shown here),
// representing the sensors

Sensor sensor_t = initSensor(SENSOR_PORT0, &T_Obj, T_Controller);
Sensor sensor_p = initSensor(SENSOR_PORT1, &P_Obj, P_Controller);

...
```

Concurrent solution (TinyTimber)

```
// Define the methods for handling the input data. Each method is
// called with the data from the sensor as parameter.

void T_Controller(Object *self, int data) {
  Heater_Setting HS;

  Temp_Convert(data, &HS);          -- convert to heater setting
  T_Write(HS);                      -- set temperature switch
  PrintLine("Temperature: ", data);
}

void P_Controller(Object *self, int data) {
  Pressure_Setting PS;

  Temp_Convert(data, &PS);          -- convert to pressure setting
  P_Write(PS);                      -- set pressure control
  PrintLine("Pressure: ", data);
}

...
```

Concurrent solution (TinyTimber)

```
...

// Initialize the two sensor objects

void kickoff(Object *self, int unused) {
  SENSOR_INIT(&sensor_t);
  SENSOR_INIT(&sensor_p);
}

// Install interrupt handlers for the sensors, and then kick off
// the TinyTimber run-time system

int main() {
    INSTALL(&sensor_t, sensor_interrupt, SENSOR_INT0);
    INSTALL(&sensor_p, sensor_interrupt, SENSOR_INT1);
    TINYTIMBER(&P_Obj, kickoff, 0);
    return 0;
}
```

Concurrent solution

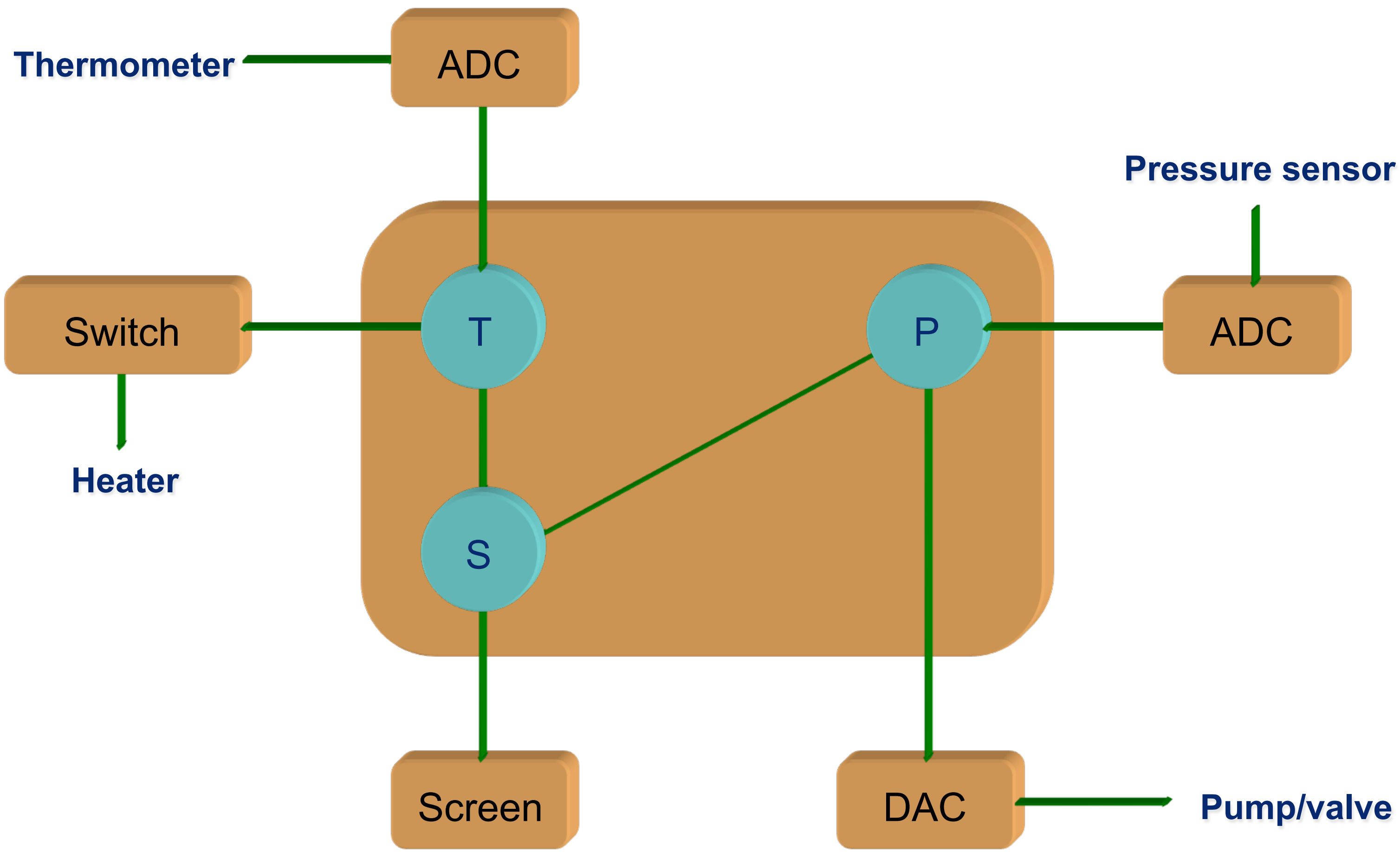
Advantages:

- the inherent parallelism of the application is fully exploited
  - pressure and temperature control do not block each other
  - the control functions can work at different frequencies
  - no processor capacity are unnecessarily consumed
  - the application becomes more reliable

Drawbacks:

- the parallel tasks share a common resource
  - the screen can only be used by one task at a time
  - a resource handler must be implemented, for controlling the access to the screen (to avoid garbled text)
  - the resource handler must guarantee *mutual exclusion* (*mutex*)

Example: control system



Solid concurrent solution (Ada95)

```
-- Protected objects in Ada95 guarantee mutual exclusion for their
   declared procedures: a calling task will be blocked if any of the
   procedures in the object are already being used.

protected type Screen_Controller is
  procedure T_Printline(data: in Temp_Reading);
  procedure P_Printline(data: in Pressure_Reading);
end Screen_Controller;

protected body Screen_Controller is
begin
  procedure T_Printline(data : in Temp_Reading) is
  begin
    Println("Temperature: ", data);
  end T_Printline;

  procedure P_Printline(data : in Pressure_Reading) is
  begin
    Println("Pressure: ", data);
  end P_Printline;
end Screen_Controller;
```

Solid concurrent solution (Ada95)

```
procedure Controller is
task T_Controller;
task P_Controller;
task body T_Controller is
begin
  loop
     T_Read(TR);
     Temp_Convert(TR,HS);
     T_Write(HS);
     Screen_Controller.T_PrintLine(TR);
  end loop;
end T_Controller;
task body P_Controller is
begin
  loop
     P_Read(PR);
     Pressure_Convert(PR,PS);
     P_Write(PS);
     Screen_Controller.P_PrintLine(PR);
  end loop;
end P_Controller;
begin
  null;        -- begin parallel execution
end Controller;
```

Solid concurrent solution (TinyTimber)

```
/*
 * TinyTimber objects guarantee mutual exclusion for their declared
 * methods: a call to the method will be blocked if any of the methods
 * in the object are already being used.
 */

// Define a new object of TinyTimber basic class Object

Object Screen_Controller = InitObject();

// Define mutex methods for the new object

void T_Println(Object *self, int data) {
    PrintLine("Temperature: ", data);
}

void P_Println(Object *self, int data) {
    PrintLine("Pressure: ", data);
}
```

Solid concurrent solution (TinyTimber)

```
/*
 * TinyTimber supports synchronous calls: the caller will be blocked
 * if any of the methods in the object are already being used.
 */

void T_Controller(Object *self, int data) {
  Heater_Setting HS;

  Temp_Convert(data, &HS);          -- convert to heater setting
  T_Write(HS);                      -- set temperature switch
  SYNC(&Screen_Controller, T_PrintLine, data);
}

void P_Controller(Object *self, int data) {
  Pressure_Setting PS;

  Temp_Convert(data, &PS);          -- convert to pressure setting
  P_Write(PS);                      -- set pressure control
  SYNC(&Screen_Controller, P_PrintLine, data);
}
```

Solid concurrent solution (TinyTimber)

```
/*
 * TinyTimber also supports asynchronous calls: the caller can continue
 * immediately after posting the method call, regardless of whether any
 * of the methods in the object are already being used or not.
 */

void T_Controller(Object *self, int data) {
  Heater_Setting HS;

  Temp_Convert(data, &HS);          -- convert to heater setting
  T_Write(HS);                      -- set temperature switch
  ASYNC(&Screen_Controller, T_PrintLine, data);
}

void P_Controller(Object *self, int data) {
  Pressure_Setting PS;

  Temp_Convert(data, &PS);          -- convert to pressure setting
  P_Write(PS);                      -- set pressure control
  ASYNC(&Screen_Controller, P_PrintLine, data);
}
```