MEDIS Project

Advanced Industrial Informatics Specialization Modules

Industrial Computers Module

Training

Saint Petersburg

Outline

1. The Problem
2. Project

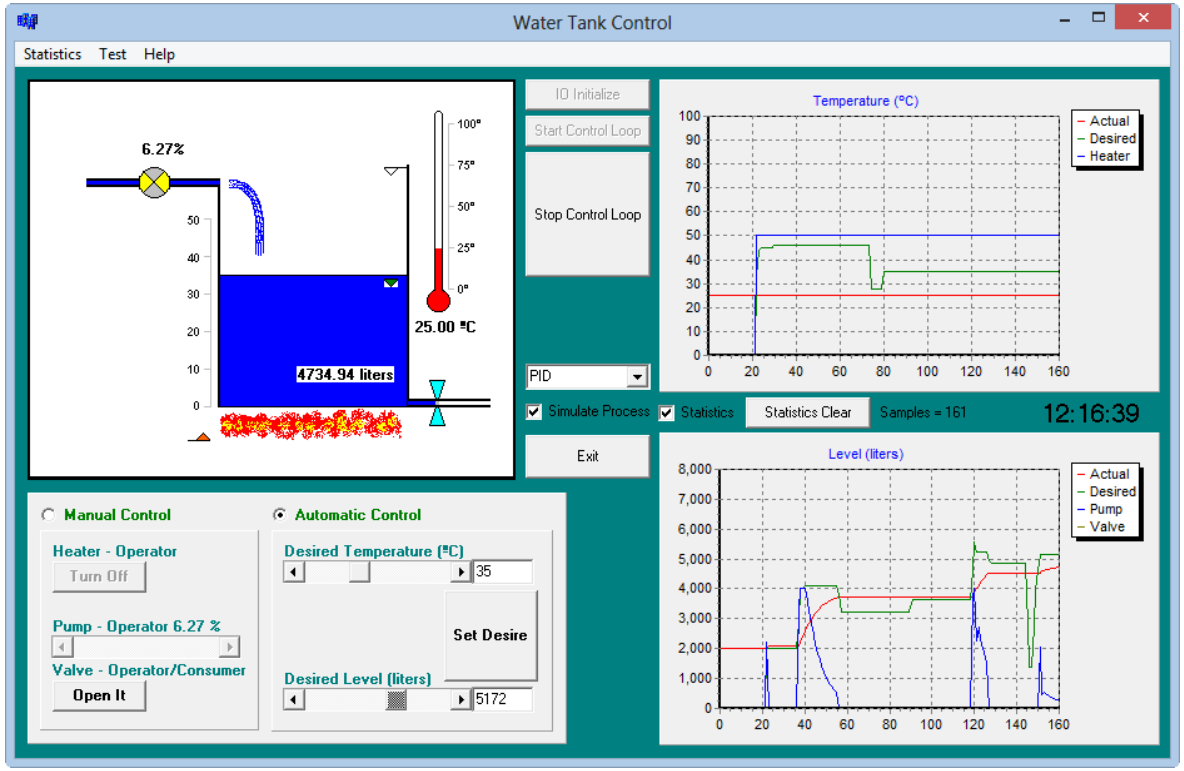
The Problem

The Problem – The Water Tank Controller

Teaching Philosophy – Problem Based Learning

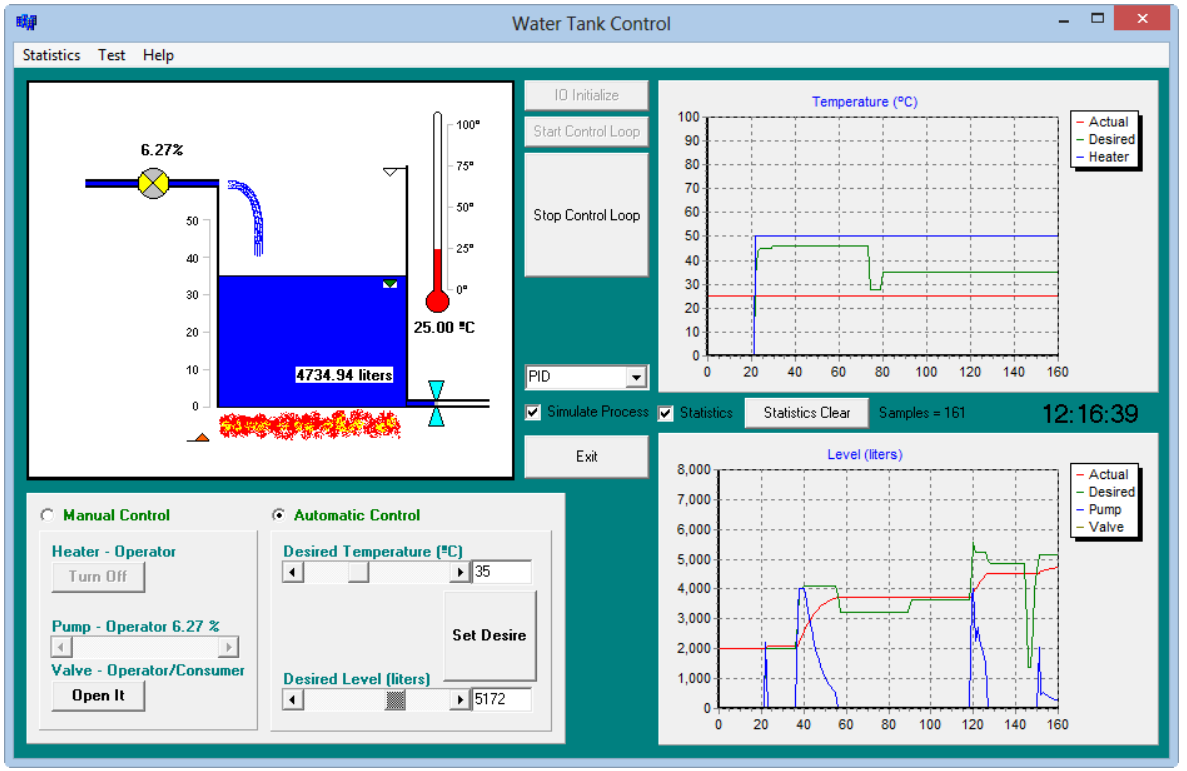
Water Tank

Water Tank Controller – Industrial Computer - based



Specification – Functionality

- Control of two variables: Water Temperature and Level
- Control Strategies: OnOff, PID, …
- Manual / Automatic Control
- Simulated / Real Process



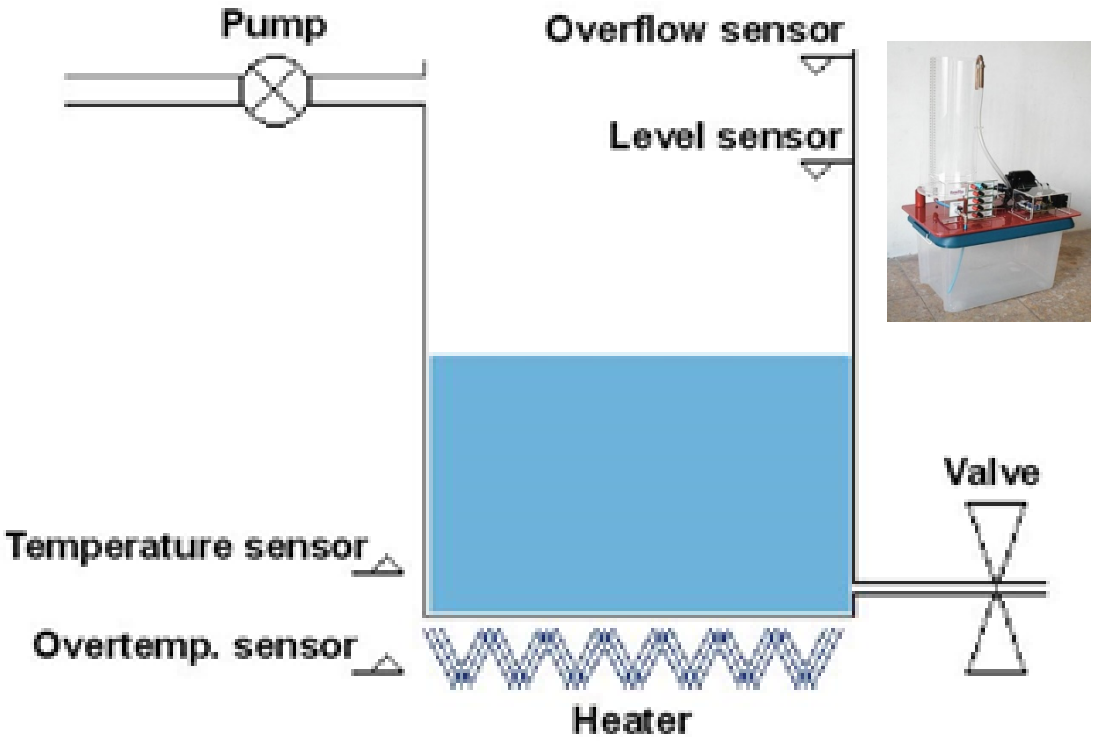
Specification – Process Interface

Sensors (Inputs):

Water Level (Analog)
Water Temperature (Analog)
Overflow Alarm (Digital)
Empty/Overheating Alarm (Digital)

Motors (Outputs):

Water Input Flow - Pump (Analog)
Water Output Flow - Valve (Digital)
Water Heater (Analog)



Specification – Event-based Control

Hot Water Service

After the initialization, when user orders a new service,

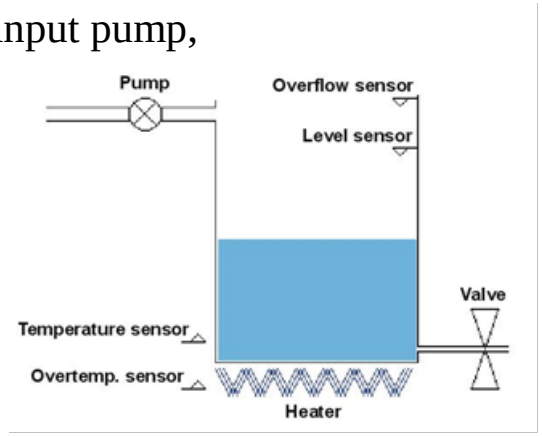
the tank is filled with cold water, using the input pump,
until a predefined level,

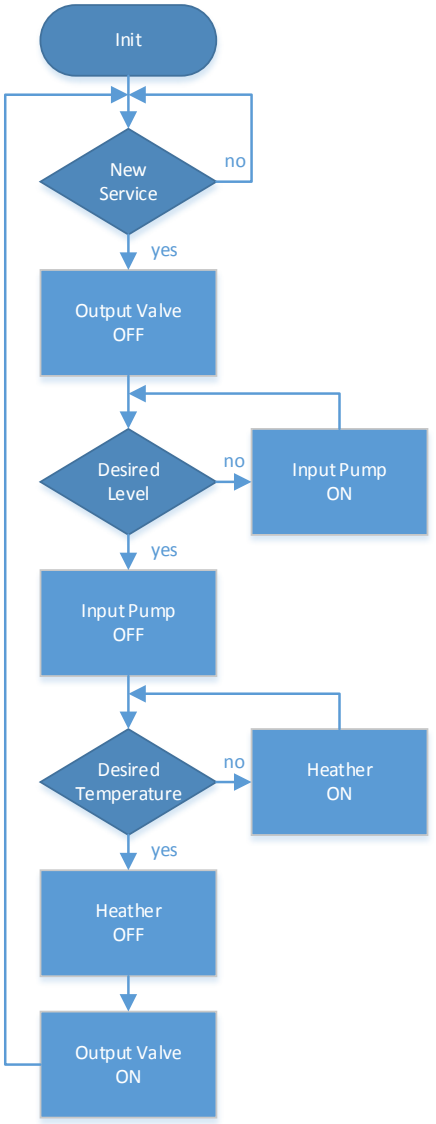
then

the water is heated, using the heather,
until a predefined temperature,

finally

the hot water is served through the output valve,
the controller waits until a new service request.





Specification – Continuous Control

Control Alternatives

Type of Control	Type of Driving Signal
On/Off	Digital (2 states)
	Analog (2 states)
P-discrete	Digital (N states)
	Analog (N states)
P	Digital PWM
	Analog
PID	Digital PWM
	Analog

Process Interface Resources

Data Acquisition Card – NI USB-6008

The National Instruments - NI USB-6008 (and USB-6009) are low-cost DAQ devices with easy screw connectivity and a small form factor.

With plug-and-play USB connectivity, these devices are simple enough for quick measurements but versatile enough for more complex measurement applications.

Web Site
http://sine.ni.com/nips/cds/view/p/lang/en/nid/201986

Operating System
Mac OS X, Windows 2000/XP, 7, 8, CE, Mobile, Vista

Driver
NI-DAQmx

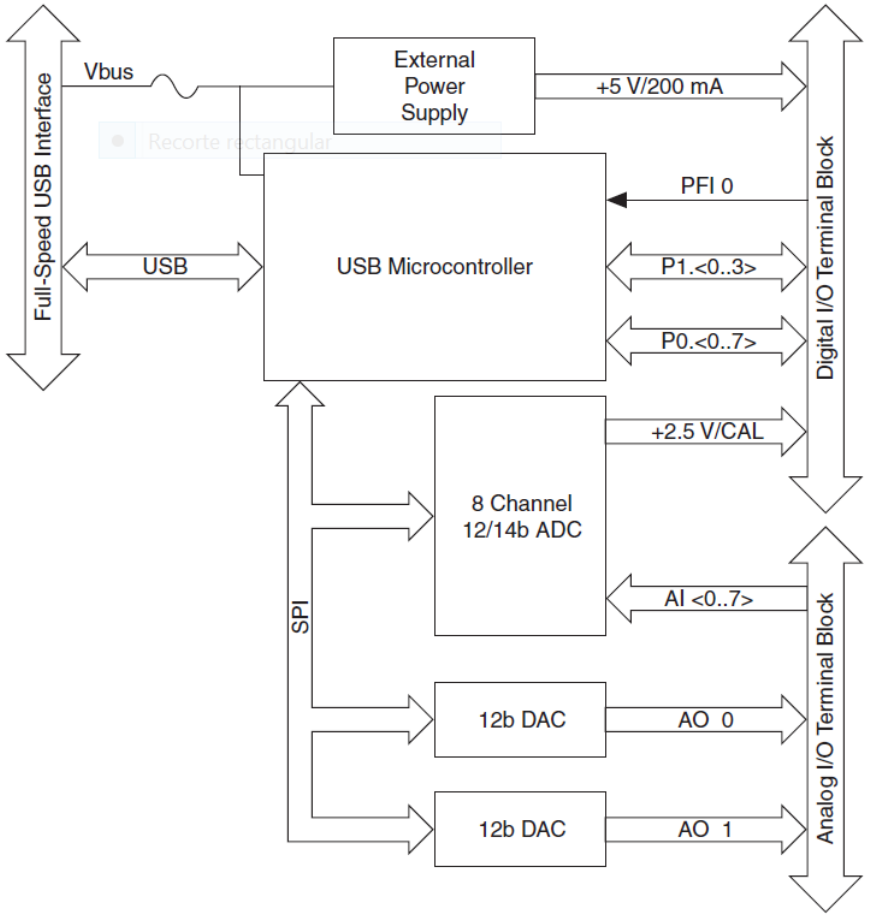
NI USB-6008 – Characteristics

8 analog inputs
- at 12 or 14 bits
- up to 48 KSamples/s

2 analog outputs
- at 12 bits
- software-timed

12 TTL/CMOS digital I/O lines

One 32-bit, 5 MHz counter

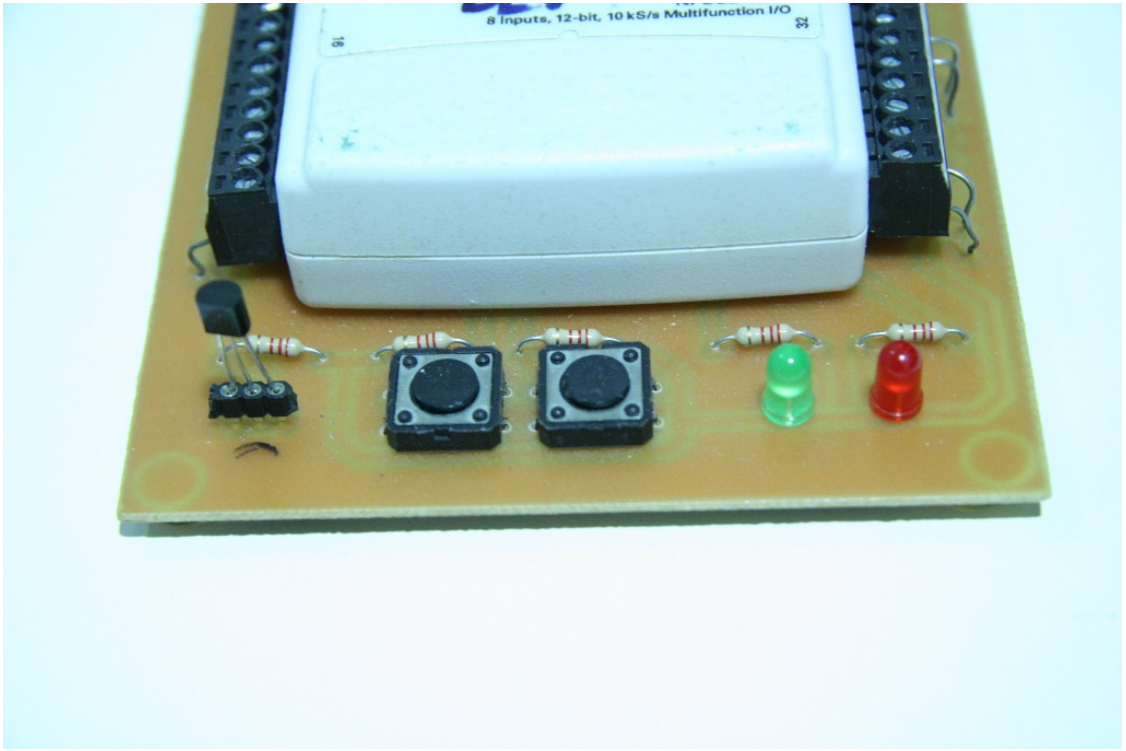


NI USB-6008 – Extensions – Simulators of Signals

2 buttons
simulate sensors – digital inputs

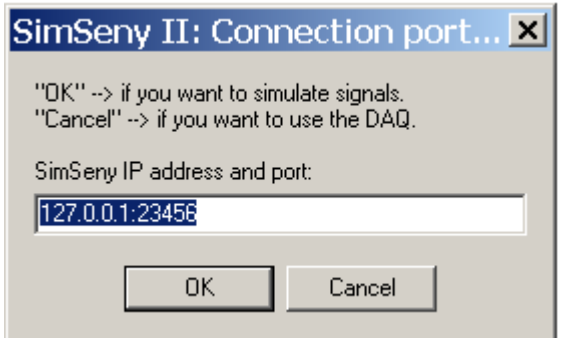
2 LEDs
simulate motors – digital outputs

1 LM-335
temperature sensor – analog input



SimSeny

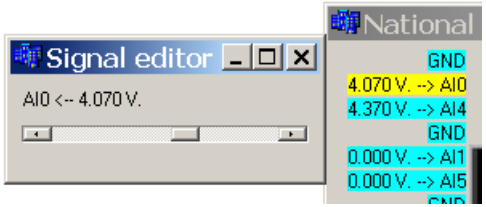
- SimSeny simulates at signal level
- Re-implements the API of a data acquisition card (e.g. NI USB-6008)
- It can be executed anyplace in the net (on a personal computer)

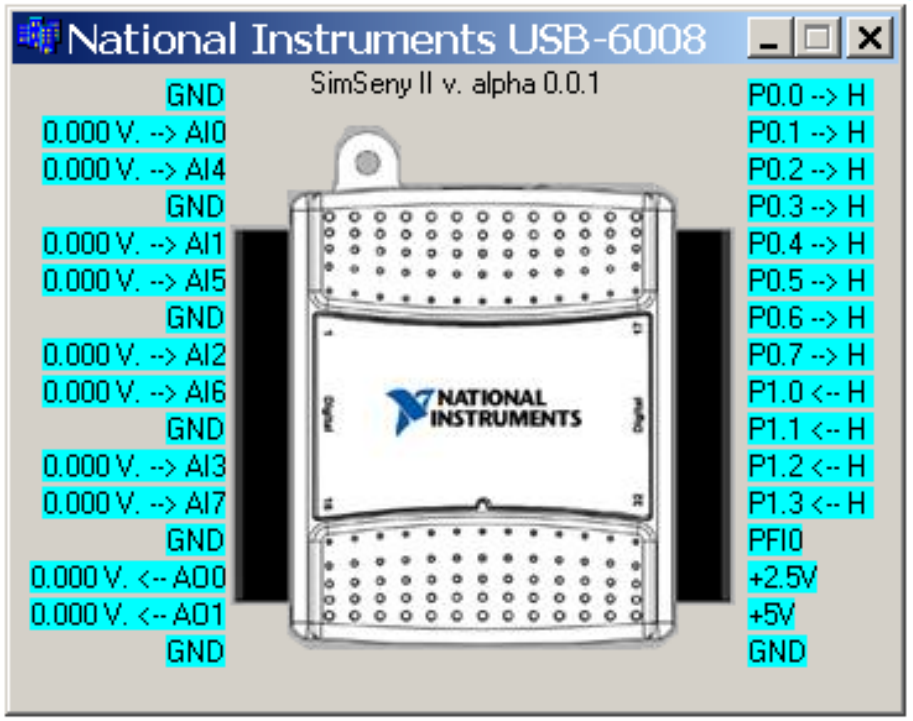


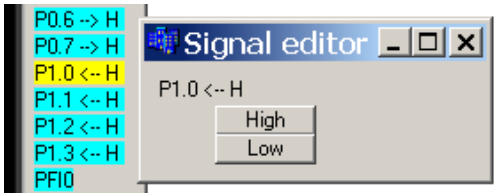
Web site:

http://www.disca.upv.es/aperles/simseny2/us_simseny2.html

SimSeny – Interface







Project

Program

Chapter 1 – Introduction to Industrial Informatics
Chapter 2 – Computer

Chapter 3 – Project Planning

Chapter 4 – Common Variables
Chapter 5 – Process Interface
Chapter 6 – User Interface
Chapter 7 – Tasks
Chapter 8 – Control

Chapter 9 – Project Integration

Project Phases

Software Engineering Techniques – *General Problem Solving Techniques applied to Software Development.*

Phases

recursively…

- Plan
- Design
- Develop

During Planning – Set Goals and Resources (*Requirements and Specifications)*

During Design – Improve the Solution (*Models)*

During Development – Apply and Validate the Solution (*Programs)*

Design Strategies

Two Simultaneous Design Strategies

- Top-Down
- Bottom-Up

Top-Down – Problem Decomposition (Modular Programming, Structured Programming)

Modular Decomposition
Criteria…

- Group Components by Functionality
- Try to Minimize the Number of Interfaces
- Decompose the Problems till a Tractable Level

Bottom-Up – Integrating Components

During Labs Exercises – Work at Components Level
During Project Exercises – Work at System Level (Integrate the Components)

Proposed Modules

Common Variables	• Defines a data structure that represents the state of the process under control. • It is implemented as a “blackboard”, where different processes, knowledge’s producers and knowledge’s consumers, share global information. • Helps to minimize the number of interfaces of the set of modules.
User Interface	• Defines the interface with the user of the application. • Implements the inputs of the parameters and the goals. • Implements a readable output of the state of the process.
Process Interface	• Defines the interface with the process under control. • Performs the inputs from the sensors. • Performs the outputs to the motors.
Simulated Process Interface	• Defines dynamic models of the physical process, which are used to simulate the process and test the controller.
Control	• Makes the decisions about the control actions. • Applies different control strategies. • Defines a machine of states. • Defines continuous regulators.
Tasks	• Synchronizes the execution of the different tasks of the controller. • Implements a tasks dispatcher.

Implementation of the Modules

Each module is implemented as two files:

- Header File (Module Interface) – File Name Extension: H
- Implementation File – File Name Extension: CPP

Students Apply

Laboratories about Modular Decomposition of the Project

Design of the Modules – Sequence

1. Common Variables
2. User Interface
3. Process Interface
4. Tasks
5. Control

Common Variables Module

- The Common Variables Module is a central module, because it minimizes the number of interfaces in the system (the number of interdependences between the other modules)
- This simplifies the design, and helps the maintenance of the program.
- Once the variables that describe the problem are defined, then it is easier to define the functions that generate these variables.

Students Apply

Laboratories about Basic Programming in C++

Common Variables Module

Design Steps

- Analysis of the Problem
- Identification of the Key Variables that Describe the Problem
- Definition of the Variables using appropriate C++ Data Types

  The Data Structure is hidden into the Implementation of the Module

  Groups of variables:

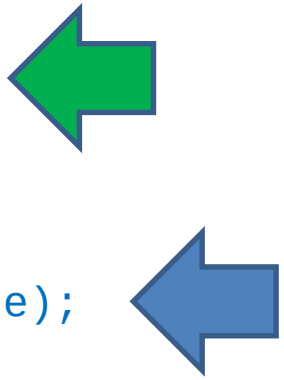
  - Observations (from the Sensors)
  - Actions (to the Motors)
  - Desires (goals)
  - Decisions
  - Parameters of the controller (modes, control strategies, control state)
- Definition of pairs of Access Functions (Read and Write) for each of the variables

  Defined in the implementation file

  Declared (published) in the header file

UnitCommonVariables.h

```
// Overflow Sensor (Digital 2-States)

enum EnumOverflow {OVERFLOW_NO, OVERFLOW_YES};

EnumOverflow ReadOverflow(void);

void         WriteOverflow(EnumOverflow new_value);
```



UnitCommonVariables.cpp

```
#include "UnitCommonVariables.h"

// Overflow Sensor (Digital 2-States)

static EnumOverflow Overflow = OVERFLOW_NO;

EnumOverflow ReadOverflow(void)
    {
    return Overflow;
    }

void WriteOverflow(EnumOverflow new_value)
    {
    Overflow = new_value;
    }
```

UnitCommonVariables.h

```
// Temperature Sensor (Analog)
double ReadTemperature (void);
void   WriteTemperature(double new_value);
```

UnitCommonVariables.cpp

```
static double Temperature;
double ReadTemperature(void){return Temperature;}

void WriteTemperature(double new_value) {
   if( (new_value >= TEMP_MIN) && (new_value <= TEMP_MAX) )
      {
      Temperature = new_value;
      }
   else
      {
      emit TempERROR(); //Raises a signal
      }
```

Common Variables – List of Variables

UnitCommonVariables.cpp

```
// SENSORS //

// Overflow Sensor (Digital 2-States)
static EnumOverflow Overflow = OVERFLOW_NO;

// Overheating Sensor (Digital 2-States)
static EnumOverheating Overheating = OVERHEATING_NO;

// Stop Sensor (Digital 2-States)
static EnumStop Stop = STOP_NO;

// Level Sensor (Analog)
static double Level;

// Temperature Sensor (Analog)
static double Temperature;
```

UnitCommonVariables.cpp

```
// MOTORS //

// Heater Motor (Digital 2-States)
static EnumHeater Heater = HEATER_OFF;

// Valve Motor (Digital 2-States)
static EnumValve Valve = VALVE_CLOSED;

// Pump Motor (Analog)
static double Pump;
```

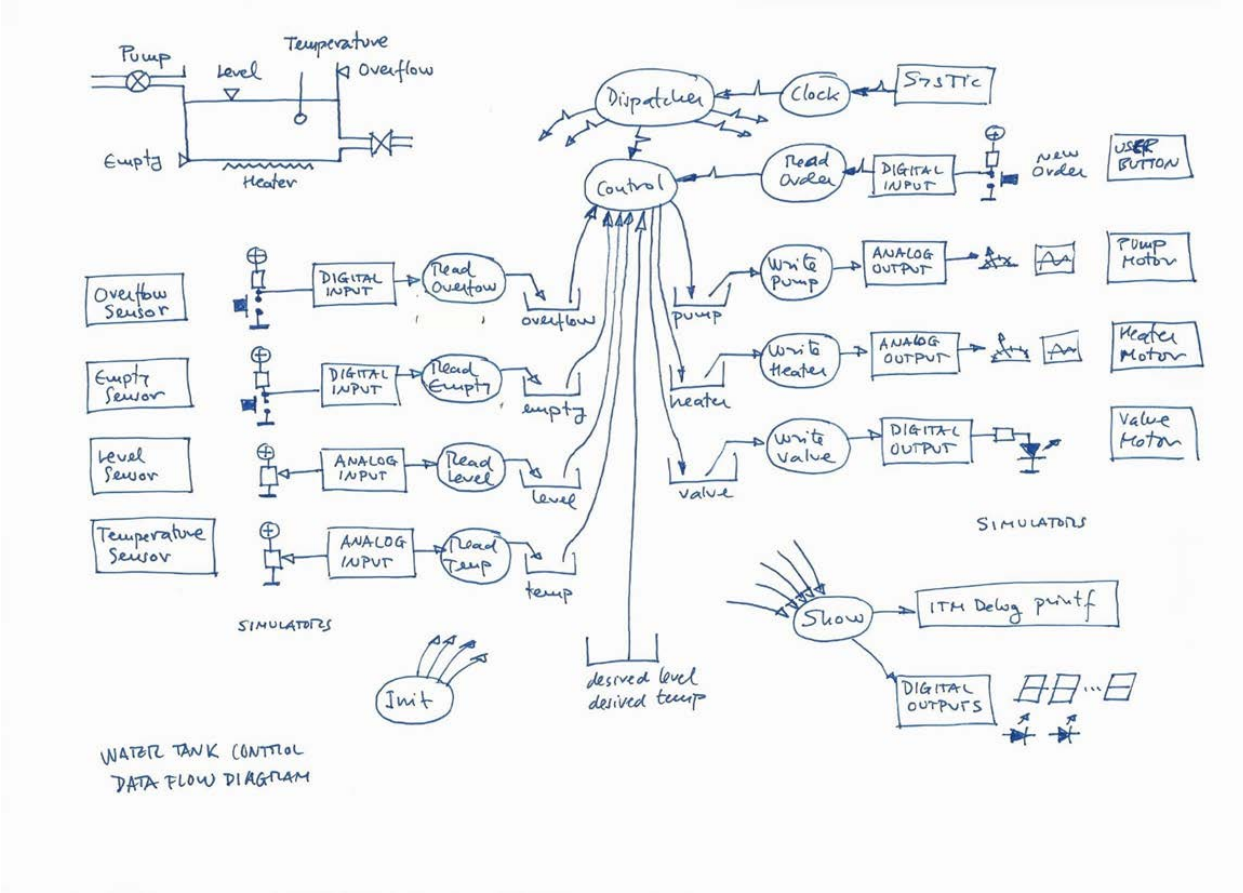
UnitCommonVariables.cpp

```
// DESIRES (GOALS) AND PARAMETERS //

// Level Desire (Analog)
static double LevelDesired;

// Temperature Desire (Analog)
static double TemperatureDesired;

// ControlMode (Digital N-States 2-States Now)
static EnumControlMode ControlMode = CONTROLMODE_MANUAL;

// ControlStrategy (Digital N-States 5-States Now)
static EnumControlStrategy ControlStrategy = CONTROLSTRATEGY_PID;

// ControlTimePeriod
static double ControlTimePeriod = DEFAULT_PERIOD;
```

UnitCommonVariables.cpp

```
// CONTROL AND SIMULATION //

// ControlState
static enum EnumControlState ControlState = CONTROLSTATE_OFF;

// Simulate Process
static bool SimulateProcess = true;

// Statistics
static bool Statistics = true;

// Samples
static unsigned long int Samples = 0;
```

Common Variables - Data Flow Diagram Example



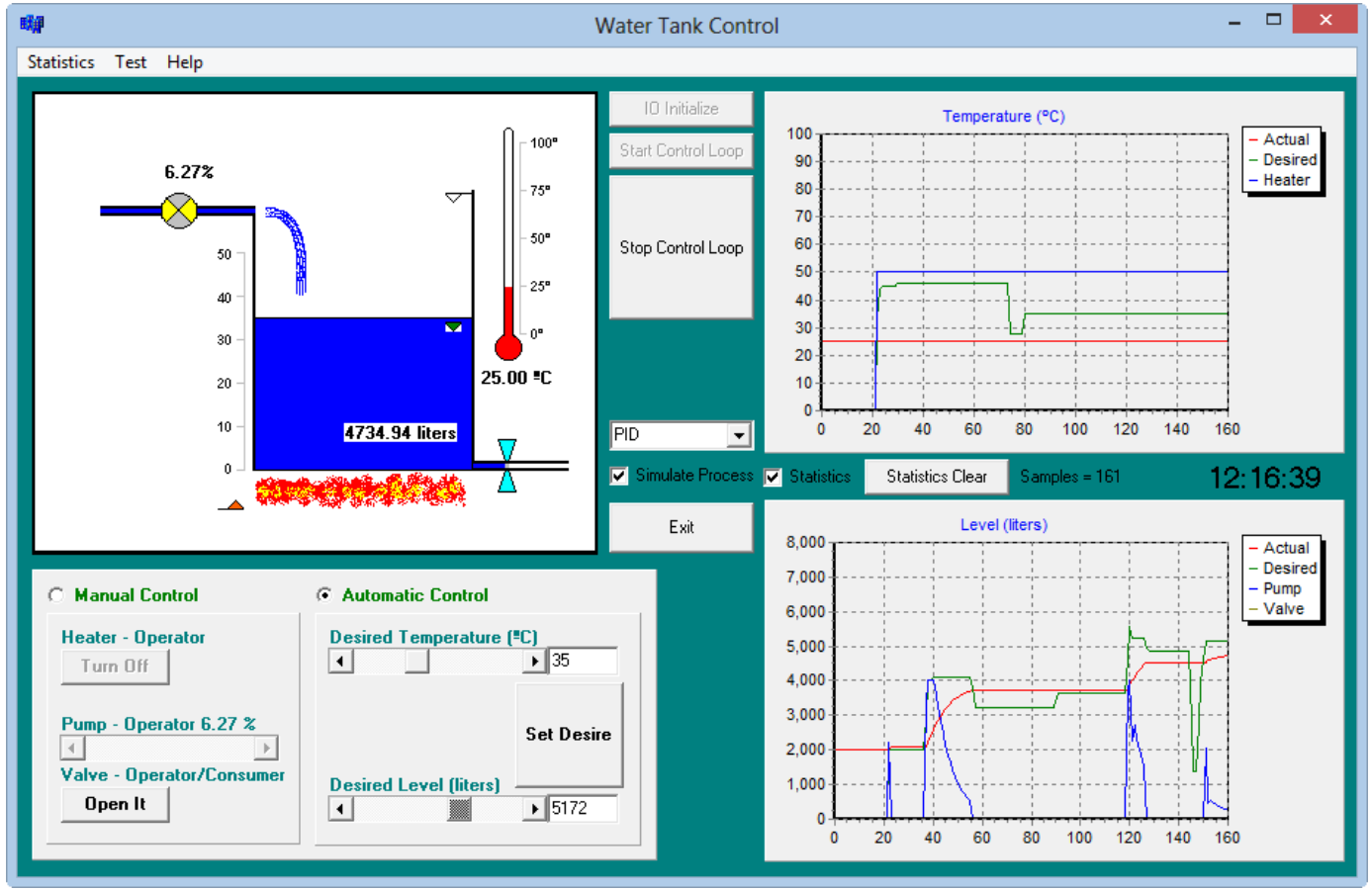
User Interface Module

- After the Common Variables Module, we continue with the User Interface, because the control elements of the user interface help to describe the specification of the controller (as a visual reminder of the actions to be performed)
- At the beginning of the User Interface Module implementation, most of the events from the user interface are not handled yet. As additional parts of the program are developed, the user interface becomes active.

Students Apply

Laboratories about Windows Events Loop, and about Design of Graphical User Interfaces.

User Interface Module - Example



Process Interface Module

- After the User Interface Module we continue with the Process Interface, implementing the sensors inputs and the motors outputs.
- The module defines Configuration functions and Operation (Input or Output) function.
- The functions of this module use the NIDAQmx library.

Students Apply

Laboratories about C bitwise operators and masks, and about the Data Acquisition Card, Digital Inputs, Digital Outputs, Analog Inputs, and Analog Outputs.

Simulated Process Interface Module

- To help validating the controller, students develop simulators of the physical process.

Students Apply

Laboratories about C++ programming, and about Timers.

UnitProcessInterface.h

```
                        // IO SYSTEM CONTROL //

void IO_Initialize (void);

bool IO_Initialized(void);
void IO_Finalize   (void);
```

UnitProcessInterface.h

```
// SENSORS (INPUTS) //

// Overflow Sensor (Digital Input 2-States)
EnumOverflow GetOverflow(void);

// Overheating Sensor (Digital Input 2-States)
EnumOverheating GetOverheating(void);

// Stop Sensor (Digital Input 2-States)
EnumStop GetStop(void);

// Level Sensor (Analog Input)
double GetLevel(void);

// Temperature Sensor (Analog Input)
double GetTemperature(void);
```

UnitProcessInterface.h

```
// MOTORS //

// Heater Motor (Digital Output 2-States)
void SetHeater(EnumHeater new_value);

// Valve Motor (Digital Output 2-States)
void SetValve(EnumValve new_value);

// Pump Motor (Analog Output)
void SetPump(double new_value);

// StopState
void SetStopState(void);
```

UnitSimulatedProcessInterface.h

```
void Simulate(void);

EnumOverflow    GetSimualtedOverflow   (void);
EnumOverheating GetSimulatedOverheating(void);

double GetSimulatedLevel      (void);
double GetSimulatedTemperature(void);
```

UnitProcessInterface.cpp

```
#define SS_FALSE_DAQ    //SimSeny: to avoid the needing of drivers

#include "ss_nidaqmx.h" //SimSeny: replaces "nidaqmx.h"
//#include "nidaqmx.h"  //NiDAQ real card

#include <stdio.h>
#include <stdlib.h>
#include <vcl.h>

#include "UnitProcessInterface.h"
```

UnitProcessInterface.cpp

```
#define IO_CARD_NAME                "InfiDAQ"

#define DIGITAL_INPUT_PORT          "port0"
#define BIT_SENSOR_OVERFLOW         0
#define BIT_SENSOR_OVERHEATING      1
#define BIT_SENSOR_STOP             2

#define DIGITAL_OUTPUT_PORT         "port1"
#define BIT_MOTOR_VALVE             0
#define BIT_MOTOR_HEATER            1

#define CHANNEL_MOTOR_PUMP          "ao0"
#define CHANNEL_SENSOR_TEMPERATURE "ai0"
#define CHANNEL_SENSOR_LEVEL        "ai1"
```

UnitProcessInterface.cpp

```
static TaskHandle TaskDigitalOutputs; //Heater, Valve
static TaskHandle TaskDigitalInputs;  //Overflow, Overheating, Stop
static TaskHandle TaskAnalogOutput_Pump;
static TaskHandle TaskAnalogInput_Level;
static TaskHandle TaskAnalogInput_Temperature;

static bool Initialized = false;

static int LastDigitalOutput;
```

UnitProcessInterface.cpp

```
void IO_Error(char *message)
   {
   ShowMessage(message);
   exit(1);
   }

void IO_Test(int error_code, int line)
   {
   char message[2000];
   char buffer [1000];
   if(error_code != 0)
      {
      DAQmxGetErrorString(error_code, buffer, 1000);
      sprintf(message,
              "NiDAQmx Error\nLine: %d\nReason: %s\n",
              line, buffer);
      IO_Error(message); }}
```

UnitProcessInterface.cpp

```
void IO_Initialize(void)
   {
   IO_Test( DAQmxCreateTask("Digital Inputs Task",
                            &TaskDigitalInputs),
            __LINE__);

   IO_Test( DAQmxCreateDIChan(TaskDigitalInputs,
                              IO_CARD_NAME/**/"/"/**/DIGITAL_INPUT_PORT,
                              "",
                              DAQmx_Val_ChanForAllLines),
            __LINE__);
```

Continues…

UnitProcessInterface.cpp

```
    IO_Test( DAQmxCreateTask("Digital Outputs Task ",
                             &TaskDigitalOutputs),
             __LINE__);

    IO_Test( DAQmxCreateDOChan(TaskDigitalOutputs,
                               IO_CARD_NAME/**/"/"/**/DIGITAL_OUTPUT_PORT,
                               "",
                               DAQmx_Val_ChanForAllLines),
             __LINE__);

    LastDigitalOutput = 0x0000;
```

Continues…

UnitProcessInterface.cpp

```
IO_Test( DAQmxCreateTask("Pump Task",
                         &TaskAnalogOutput_Pump), __LINE__);

IO_Test( DAQmxCreateAOVoltageChan(TaskAnalogOutput_Pump,
                           IO_CARD_NAME/**/"/"/**/CHANNEL_MOTOR_PUMP,
                                  "Pump Channel",
                                  0.0,
                                  5.0,
                                  DAQmx_Val_Volts,
                                  NULL),
          __LINE__);
```

Continues…

UnitProcessInterface.cpp

```
IO_Test( DAQmxCreateTask("Temperature Task",
                         &TaskAnalogInput_Temperature),
         __LINE__);

IO_Test( DAQmxCreateAIVoltageChan(TaskAnalogInput_Temperature,
                  IO_CARD_NAME/**/"/"/**/CHANNEL_SENSOR_TEMPERATURE,
                                  "",
                                  DAQmx_Val_RSE,
                                  0.0,
                                  10.0,
                                  DAQmx_Val_Volts,
                                  NULL),
          __LINE__);
```

Continues…

UnitProcessInterface.cpp

```
IO_Test( DAQmxCreateTask("Level Task",
                         &TaskAnalogInput_Level),
         __LINE__);

IO_Test( DAQmxCreateAIVoltageChan(TaskAnalogInput_Level,
                       IO_CARD_NAME/**/"/"/**/CHANNEL_SENSOR_LEVEL,
                                  "",
                                  DAQmx_Val_RSE,
                                  0.0,
                                  6.0,
                                  DAQmx_Val_Volts,
                                  NULL),
         __LINE__);

Initialized = true;
}
```

UnitProcessInterface.cpp

```
bool IO_Initialized(void)
   {
   return Initialized;
   }

void IO_Finalize(void)
   {
   if(Initialized)
      {
      SetPump  (0.0);
      SetValve (VALVE_CLOSED);
      SetHeater(HEATER_OFF);
      }
   }
```

UnitProcessInterface.cpp

```
EnumOverflow GetOverflow(void)   {
   uInt32 data;

   IO_Test( DAQmxStartTask(TaskDigitalInputs),
            __LINE__);
   IO_Test( DAQmxReadDigitalScalarU32(TaskDigitalInputs,
                                      0.0,
                                      &data,
                                      NULL),
            __LINE__);
   IO_Test( DAQmxStopTask(TaskDigitalInputs),
                          __LINE__);

   if((data & (1 << BIT_SENSOR_OVERFLOW)) == 0)  //0=YES, 1=NO
      return(OVERFLOW_YES);
   else
      return(OVERFLOW_NO);   }
```

UnitProcessInterface.cpp

```
EnumOverheating GetOverheating(void){…}
EnumStop GetStop(void){…}
```

UnitProcessInterface.cpp

```
double GetLevel(void)   {
   float64 volts;
   double liters;

   IO_Test( DAQmxStartTask(TaskAnalogInput_Level),
            __LINE__);
   IO_Test( DAQmxReadAnalogScalarF64(TaskAnalogInput_Level,
            1.0,
            &volts,
            NULL),
            __LINE__);
   IO_Test( DAQmxStopTask(TaskAnalogInput_Level),
            __LINE__);

   liters = (6.0 - volts) * (3800.0 / 6.0); // 6V=0liters, 0V=3800liters
   return(liters);   }
```

UnitProcessInterface.cpp

```
double GetTemperature(void){…}
```

UnitProcessInterface.cpp

```
void SetHeater(EnumHeater new_value)
   {
   if(new_value == HEATER_ON)
      LastDigitalOutput = LastDigitalOutput | (1 << BIT_MOTOR_HEATER);
   else
      LastDigitalOutput = LastDigitalOutput & ~(1 << BIT_MOTOR_HEATER);

   IO_Test( DAQmxStartTask(TaskDigitalOutputs), __LINE__);
   IO_Test( DAQmxWriteDigitalScalarU32(TaskDigitalOutputs,
                                       false,
                                       0.0,
                                       LastDigitalOutput,
                                       NULL),
            __LINE__);
   IO_Test( DAQmxStopTask(TaskDigitalOutputs),
            __LINE__);
   }
```

UnitProcessInterface.cpp

```
void SetValve(EnumValve new_value) {…}
```

UnitProcessInterface.cpp

```
void SetPump(double new_value)   {
   double volts;

   volts = new_value * (5.0/100.0); //0V=0%, 5V=100%

   IO_Test( DAQmxStartTask(TaskAnalogOutput_Pump),
            __LINE__);
   IO_Test( DAQmxWriteAnalogScalarF64(TaskAnalogOutput_Pump,
                                      false,
                                      1.0,
                                      volts,
                                      NULL),
            __LINE__);
   IO_Test( DAQmxStopTask(TaskAnalogOutput_Pump),
            __LINE__);
   }
```

Tasks Module

- After the Process Interface Module, we continue with the Tasks Module, defining a Clock and synchronizing the tasks of the control loop.
- Since the dynamics of the process is slow, we use a Software Timer and handle its Ticks events to iterate the control loop.

Students Apply

Laboratories about the Handling Events and Software Timers.

UnitTasks.h

```
void ControlLoopStart  (void);
void ControlLoopStop   (void);
void ControlLoopIterate(void);
```

UnitTasks.cpp

```
void ControlLoopIterate(void)   //Task Sequence
   {
   if(ReadControlState() == CONTROLSTATE_ON)
      {
      if(ReadSimulateProcess() == true)
         {
         SimulatedObserve();
         }
      else
         {
         Observe();
         }
      Desire();
      Decide();
      Act();
      }
   Inform();
   }
```

UnitTasks.cpp

```
void ControlLoopStart(void)
   {
   //Initialize in Safe State
   WriteStopState();
   SetStopState();
   //Control State ON
   WriteControlState(CONTROLSTATE_ON);
   }

void ControlLoopStop(void)
   {
   //Keep in Safe State
   WriteStopState();
   SetStopState();
   //Control State OFF
   WriteControlState(CONTROLSTATE_OFF);
   }
```

UnitTasks.cpp

```
void Observe(void)
   {
   //The Observe function establishes the current state of the problem
   //It reads from the Sensors and writes on the Common Variables
   WriteStop       ( GetStop    () );
   WriteOverflow   ( GetOverflow() );
   WriteOverheating( GetOverheating() );
   WriteLevel      ( GetLevel      () );
   WriteTemperature( GetTemperature() );
   }
```

UnitTasks.cpp

```
void SimulatedObserve(void)
   {
   Simulate(); //Step of simulation

   //From Simulated Sensors to Common Variables
   WriteStop       ( GetStop()                ); //Not Simulated
   WriteOverflow   ( GetSimualtedOverflow()   );
   WriteOverheating( GetSimulatedOverheating() );
   WriteLevel      ( GetSimulatedLevel()      );
   WriteTemperature( GetSimulatedTemperature() );
   }
```

UnitTasks.cpp

```
void Desire(void)
   {
   //The Desire function establishes the control goals

   //In this implementation
   //   this function doesn’t do anything in a direct way
   //   because the operator inputs and selections on the user interface
   //   establish the desired levels
   //The event handlers of the user interface
   //   perform that storing process
   };
```

UnitTasks.cpp

```
void Decide(void)
   {
   //Applies Control Strategies

   switch( ReadControlMode() )
      {
      case CONTROLMODE_MANUAL: //The Control is Manual
         //Nothing is done from here directly
         //User operates the process through the user interface controls
         //We keep this case for conceptual convenience
         break;

      case CONTROLMODE_AUTOMATIC: //The Control is Automatic
         Control(); // Control function
         break;
      }
   }
```

UnitTasks.cpp

```
void Act(void)
   {
   //From Common Variables to Motors

   SetPump  (ReadPump()  );
   SetValve (ReadValve() );
   SetHeater(ReadHeater());
   }
```

UnitTasks.cpp

```
void Inform(void)
   {
   if(ReadStatistics() && (ReadControlState() == CONTROLSTATE_ON))
      {
      WriteSamples(ReadSamples() + 1);
      CollectStatistics();
      }
   ShowTank();
   }
```

Control Module

- After the Tasks Module, we complete the system with the Control Module, implementing different control strategies.

Students Apply

Laboratories about C++ programming, and about Timers.

UnitControl.h

```
void Control(void);
```

UnitControl.cpp

```
#include <stdlib.h>
#include "UnitCommonVariables.h"
#include "UnitControl.h"

void Control(void)
   {
   //Alarming Situations
   if((ReadOverflow()    == OVERFLOW_YES)||
      (ReadOverheating() == OVERHEATING_YES))
      {
      WriteHeater(HEATER_OFF);
      WriteValve (VALVE_CLOSED);
      WritePump  (0.0);
      emit LevelALARM();
      return;
      }

//Continues…
```

```
switch(ReadControlStrategy())
   {
   case CONTROLSTRATEGY_ONOFF:
      ControlOnOff();
      break;
   case CONTROLSTRATEGY_ONOFF_H:
      ControlOnOffHysteresis();
      break;
   case CONTROLSTRATEGY_P:
      ControlP();
      break;
   case CONTROLSTRATEGY_PI:
      ControlPI();
      break;
   case CONTROLSTRATEGY_PID:
      ControlPID();
      break;
   }
}
```

UnitControl.cpp

```
void ControlOnOff(void)
   {
   ControlOnOffLevel();
   ControlOnOffTemperature();
   }

void ControlOnOffHysteresis(void){…}
void ControlP(void){…}
void ControlPI(void){…}
void ControlPID(void){…}
```

UnitControl.cpp

```
void ControlOnOffLevel(void)
   {
   if( ReadLevel() < ReadLevelDesired() )
      {
      WritePump(100.0);
      }
   else
      {
      WritePump(0.0);
      }
   }
```