Generation of Structured Modula-2 Code From a Simnon System Description

Dahl, Ola

1989

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Dahl, O. (1989). *Generation of Structured Modula-2 Code From a Simnon System Description*. (Technical Reports TFRT-7416). Department of Automatic Control, Lund Institute of Technology (LTH).

Total number of authors:
1

CODEN: LUTFD2/(TFRT-7416)/1-33/(1989)

Generation of Structured Modula-2 Code From a Simnon System Description

Ola Dahl

Department of Automatic Control
Lund Institute of Technology
February 1989

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
Internal Report

Date of issue
February 1989

Document Number
CODEN: LUTFD2/(TFRT-7416)/1–33/(1988)

Author(s)
Ola Dahl

Supervisor
Karl Johan Åström, Lars Nielsen

Sponsoring organisation

Title and subtitle
Generation of Structured Modula-2 Code From a Simnon System Description

Abstract

Real time implementation of control algorithms is simplified by using automatic code generation. A compiler is used to translate the control algorithm from a high level design language to an implementation language. By using the same language for simulation and design, the control algorithm can easily be tested in simulation before the translation is done.

A compiler generating Modula-2 code from a Simnon system description has been developed. The compiler is written in Scheme, and translates a discrete Simnon system into a Modula-2 module. The generated code includes both the control algorithm and procedures for accessing variables. The variables are, as in Simnon, divided into different types, such as states, inputs, outputs, and parameters. The access procedures are used to read and write values, to get the number of variables of a certain type, and to get variable names. The chosen set of access procedures creates an environment for controller implementation, where it is possible to design a flexible program for real time control. The user interface can be made independent of the control algorithm, and the control algorithm can be changed without modifying the rest of the program.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
33

Recipient's notes

Security classification

1. Introduction

Simulation is often used for controller design. When the design is complete, a real time implementation of the control algorithm may be desirable. This can be more or less time consuming, depending on the tools available. The program described in this report is an attempt to simplify important parts of the implementation process by using automatic code generation. The control algorithm is written in a high level design language, and is automatically translated into the implementation language. This reduces the implementation effort, and eliminates the risk of introducing errors during the translation. By using the same language for simulation and design, the control algorithm can easily be tested in simulation before the translation is done.

An important part of the real time implementation is the user interface. When the controller is connected to the physical process, the user often wants to adjust the parameters in the control algorithm. The values of input and output variables may also be of interest. The user refers to the different variables, for example via their names, and the program reads or writes the associated values. This means that certain facts about the variables, such as parameter names, the number of inputs etc. must be embedded in the program. When a new control algorithm is implemented, new facts must be entered. When automatic code generation is used for the control algorithm, it is easy to also generate code for reading and writing parameters and other variables. By doing this, the control algorithm can easily be connected to the user interface of the real time system, and the control algorithm can be changed without modifying the rest of the real time program.

This report describes a compiler for translation of a discrete Simnon system into a Modula-2 module. The compiler can be used to simplify the implementation of control algorithms. The control algorithm is written in the Simnon language, and translated into a Modula-2 module. The compiler analyzes the source text, and errors are reported. The generated code contains the control algorithm and procedures for accessing the variables defined in the Simnon system. By using the Simnon language as design language, the control algorithm can either be tested in simulation, or directly translated into Modula-2. The modularization facilities of Modula-2 also gives advantages. The translated control algorithms are separated in modules, and it is for instance easy to build libraries of control algorithms.

The report is organized as follows. Section 2 contains an example of a translation, the compiler is described in Section 3, and the conclusions are given in Section 4. In Appendix a simple user program for single loop control is given. The program uses the generated code, and it is easy to change the control algorithm without modifying the rest of the program.

2. An Example

Some properties of the compiler will be demonstrated. The Simnon system to be translated is a PID controller (Åström, 1987). Here, only parts of the generated Modula-2 code will be shown. A complete listing together with a main program is given in Appendix.

2.1 Translation of the Control Algorithm

The controller is implemented as the following Simnon system.

```
DISCRETE System PID

STATE i x
NEW ni nx
INPUT r y
OUTPUT u
TIME t
TSAMP ts

"Discrete time PID regulator with anti-windup

u = if ulimon then us else u1
us = if u1<umin then umin else if u1<umax then u1 else umax

u1 = K*b*r - K*y + i + x - bi*y

ni = i + K*h*e/Ti + aw
e = r - y
aw = if awon then h/Tr*(u - u1) else 0

nx = ai*x + bi*(1 - ai)*y
ai = Td/(Td + N*h)
bi = K*Td*N/(Td + N*h)

ts = t + h

ulimon : 0 "Switch for output saturation
awon : 0   "Switch for anti windup

umax : 1e3  "Output limits
umin : -1e3

K : 1      "Regulator gain
Ti : 1     "Integration time
Td : 0     "Derivative time
N : 10     "Derivative filter constant
b : 1      "Feedforward gain
```

```
h : 0.1   "Sampling interval
Tr : 1e5  "Anti windup time constant

END
```

Compiling the system gives two files as output, a definition module and an implementation module. The definition module contains declarations of the types and procedures accessible to the user of the module. The exported types are the hidden type SystemType, and the record types for the variables defined in the Simnon system. A variable of type SystemType represents a system, in this case a PID-regulator, containing the current values of the system variables. Each variable is regarded as having one of the six types: state, new, input, output, par, or auxvar. The system type and the parameters are declared as

```
TYPE

SystemType;

ParType = RECORD
  ulimon, awon, umax, umin, k, ti, td, n, b, h, tr : REAL;
END;
```

The procedures Init, UpDate, and IncrementTime operate on variables of type SystemType. These procedures are declared in the definition module as

```
PROCEDURE Init(VAR S : SystemType);

PROCEDURE UpDate(S : SystemType);

PROCEDURE IncrementTime(S : SystemType; VAR T : Time);
```

Initialization of a system is done by Init. The parameters are given the values defined in the Simnon system, and all other variables are given the value zero. The algorithm is executed once by calling UpDate. The equations from the Simnon system are then evaluated. A call to IncrementTime, with the current time as actual parameter to T, then gives the time for the next update. Note that in the execution of the algorithm, no reading of inputs from outside the system, for example from AD-converters, is done. A typical sequence in a user program is to read inputs, write the input values to the system, call UpDate, read outputs from the system, write the outputs, for example to DA-converters, and finally call IncrementTime to get the time for the next update. All variable types are accessible by procedures. These are, for each variable type, procedures for reading and writing variables, and procedures for associating variable names to variables. The reading and writing is done either by copying a complete variable record, or by accessing individual variables. Here, only the procedures for accessing the parameters will be shown. The other variable types are accessed by similar procedures.

```
PROCEDURE WritePar(S : SystemType; VAR Par : ParType);
```

```
PROCEDURE ReadPar(S : SystemType; VAR Par : ParType);

PROCEDURE WriteParVariable(
  VAR Par : ParType;
  Name : ARRAY OF CHAR;
  Value : REAL;
  VAR NameOk : BOOLEAN);

PROCEDURE ReadParVariable(
  VAR Par : ParType;
  Name : ARRAY OF CHAR;
  VAR Value : REAL;
  VAR NameOk : BOOLEAN);
```

Reading and writing parameters from a system is done by `WritePar` and `ReadPar`. The complete parameter record is read or written. Individual parameter variables are read and written into a parameter record by `WriteParVariable` and `ReadParVariable`. In a typical application, the user program reads parameter values from the keyboard, and writes the values to a parameter record. The complete record is then copied into the system by a call to `WritePar`. Accessing individual parameters requires the name of the parameter. The following two procedures makes the connection between parameters and their names.

```
PROCEDURE GetNumberOfPars() : CARDINAL;

PROCEDURE GetParName(
  Number : CARDINAL; VAR Name : ARRAY OF CHAR;
  VAR NumberOk : BOOLEAN);
```

Consider for example the problem of displaying the parameter names and their values on the terminal. This is easily done by using the procedures `GetNumberOfPars` and `GetParName`. A call to `GetNumberOfPars` gives the number of parameters. For each parameter the corresponding name is then given by `GetParName`. A call to `ReadParVariable` finally gives the value of the parameter. Note that the code for doing this can be made without knowledge of the parameter names. This means that a user interface that is independent of the control algorithm can be constructed.

The pieces of Modula-2 code shown here are only a small part of the generated code. A complete listing is given in Appendix.

2.2 A Simple User Program

In Appendix, a simple user program is also listed. The program illustrates how the chosen format for the generated code facilitates the construction of a general single loop control program. The main module contains the code for the user interaction. The code is independent of the control algorithm, and the algorithm is changed simply by changing the module name in one of the import statements. No attempt has been made to make a complete user program. The interaction is simple and contains no graphics. The purpose

is to demonstrate how automatic code generation can be used to simplify the design of the real time control program. A more complete user program, including graphics, mouse interaction, and data logging, is under development.

3. The Compiler

The compiler can be divided into four parts. Three of them do analysis of the source text, and the forth generates the code. The source text is a subset of the Simnon language (Elmqvist, Åström and Schönthal, 1986), and the generated code is Modula-2, using library modules from Logitech (Logitech, 1986) and from the Department of Automatic Control (Andersson, 1987). The compiler can be modified to generate other types of code. This section briefly describes the different parts of the program. The internal data representation is illustrated by examples from the Simnon system used in the previous section. The compiler is written in Scheme (Dybvig, 1987).

3.1 Lexical Analysis

The lexical analysis groups the individual characters into tokens, such as identifiers, delimiters and numbers. The conversion is done according to the following syntax rules.

⟨unsigned_number⟩ :== ⟨unsigned_integer⟩ [⟨real_part⟩]
⟨real_part⟩ :== [. ⟨unsigned_integer⟩] ⟨exponent⟩
⟨exponent⟩ :== e [+ | -] ⟨unsigned_integer⟩
⟨unsigned_integer⟩ :== { ⟨digit⟩ }+
⟨identifier⟩ :== ⟨letter⟩ { ⟨letter⟩ | ⟨digit⟩ }*
⟨letter⟩ :== { a | b | ... | z | A | B | ... | Z }
⟨digit⟩ :== { 0 | 1 | ... | 9 }
⟨delimiter⟩ :== { + | - | *| / | = | > | < | (|) | : }

If s, s1 ... sn denotes character strings, occuring in the syntactic rules, the notations are defined by

⟨s⟩	A non terminal symbol, e.g. ⟨identifier⟩.
s	A terminal symbol, e.g. the character e.
[s]	The enclosed entity may be omitted.
{ s }*	Repetition, zero or more times.
{ s }+	Repetition, one or more times.
{ s1 | s2 | ... | sn }	Alternatives, of which one must be chosen.

As an example, the assignment

```
nx = ai*x + bi*(1 - ai)*y
```

from the Simnon system is, after the lexical analysis, represented as

```
(identifier "nx")
(space #\space)
(delimiter "=")
(space #\space)
(identifier "ai")
```

```
(delimiter "*")
(identifier "x")
(space #\space)
(delimiter "+")
(space #\space)
(identifier "bi")
(delimiter "*")
(delimiter "(")
(unsigned-number "1.0")
(space #\space)
(delimiter "-")
(space #\space)
(identifier "ai")
(delimiter ")")
(delimiter "*")
(identifier "y")
(newline #\newline)
```

3.2 Syntax Analysis

The syntax analysis is implemented as a recursive descent parser (Aho, Sethi and Ullman, 1986). The input format is a list of tokens, generated by the lexical analyzer, as shown in the example above. The syntax rules are

⟨syst_desc⟩ :== ⟨system_heading⟩ { ⟨declaration⟩ }* { ⟨var_assignment⟩
| ⟨par_assignment⟩ }* END

⟨system_heading⟩ :== DISCRETE SYSTEM ⟨identifier⟩

⟨declaration⟩ :== { INPUT | OUTPUT | TIME | STATE | NEW | TSAMP }
{ ⟨variable⟩ }+

⟨var_assignment⟩ :== ⟨variable⟩ = ⟨arithm_exp⟩

⟨par_assignment⟩ :== ⟨variable⟩ : ⟨number⟩

⟨arithm_exp⟩ :== { ⟨simple_arithm_exp⟩ | ⟨if_clause⟩ ⟨simple_arithm_exp⟩
ELSE ⟨arithm_exp⟩ }

⟨simple_arithm_exp⟩ :== [+ | -] ⟨term⟩ { { + | - } ⟨term⟩ }*

⟨term⟩ :== ⟨factor⟩ { { *| / } ⟨factor⟩ }*

⟨factor⟩ :== ⟨primary⟩

⟨primary⟩ :== {⟨unsigned_number⟩ | ⟨variable⟩ |
(⟨simple_arithm_exp⟩)}

⟨if_clause⟩ :== IF ⟨boolean_exp⟩ THEN

⟨boolean_exp⟩ :== ⟨boolean_term⟩

⟨boolean_term⟩ :== ⟨boolean_factor⟩ { OR ⟨boolean_factor⟩ }*

⟨boolean_factor⟩ :== ⟨boolean_secondary⟩ { AND ⟨boolean_secondary⟩ }*

⟨boolean_secondary⟩ :== [NOT] ⟨boolean_primary⟩

⟨boolean_primary⟩ :== { ⟨variable⟩ | ⟨relation⟩ | (⟨boolean_exp⟩) }

⟨relation⟩ :== ⟨simple_arithm_exp⟩ { < | > } ⟨simple_arithm_exp⟩

⟨number⟩ :== [+ | -] ⟨unsigned_number⟩

⟨variable⟩ :== ⟨identifier⟩

The Simnon statement

```
nx = ai*x + bi*(1 - ai)*y
```

is, after the syntax analysis, represented as

```
(variable-assignment
   ((identifier "nx")
    (delimiter "=")
    (simple-expression
       ((((identifier "ai"))
         (delimiter "*") ((identifier "x")))
        (delimiter "+")
        (((identifier "bi"))
         (delimiter "*")
         (((delimiter "(")
           (((unsigned-number "1.0"))
            (delimiter "-")
            ((identifier "ai")))
           (delimiter ")"))
          (delimiter "*")
          ((identifier "y"))))))
    (newline #\newline)))
```

3.3 Semantic Analysis

The semantic analyzer checks declarations and assignments. Multiple declarations and illegal assignments, e.g. assignments of input variables, are detected. The parameter assignments and the variable assignments are then sorted. The output from the semantic analysis is stored in a number of global variables. These are

`System-name`	The system name.
`Declared-identifiers`	A list of all identifiers.
`Declared-states`	A list of the state variables.
`Declared-news`	A list of the corresponding variables for updating the states.
`Declared-inputs`	A list of the input variables.
`Declared-outputs`	A list of the output variables.
`Declared-parameters`	A list of the parameters.
`Declared-auxvars`	A list of the auxiliary variables.
`Declared-time`	The variable declared as `time`.
`Declared-tsamp`	The variable declared as `tsamp`.
`Tsamp-parameter`	The parameter used in the assignment of the variable declared as `tsamp`. The assigned value must be the sum of a parameter and the variable declared as `time`.

`Parameter-Assignments`	A list of the parameter assignments.
`Variable-Assignments`	A list of the variable assignments.
`Sorted-Variable-Assignments`	A list of the sorted variable assignments, represented as shown in the example on syntax analysis.

3.4 Code Generation

The generated Modula-2 code contains the control algorithm and the procedures for accessing the variables defined in the Simnon system. Given the global variables from the semantic analysis, the access procedures are easily generated. The control algorithm, i.e. the sorted variable assignments, is generated by translating the assignment expressions from their internal representation into Modula-2 expressions. As an example, the Simnon statement

```
aw = if awon then h/Tr*(u - u1) else 0
```

is translated to

```
IF awon > 0.5 THEN
  aw := h / tr * (u - u1);
ELSE
  aw := 0.0;
END;
```

where the conditional Simnon assignment has been translated to a Modula-2 if-statement. The arithmetic expressions are also translated.

3.5 Error handling

In each of the analysis phases, the compiler checks for errors in the source text. When an error is found, the compilation stops, and an error message is printed. In most cases, the source line on which the error was found is also printed.

3.6 Possible Extensions of the Compiler

The compiler can be modified in several ways. One possible extension of the analysis part of the compiler is to extend the source language to cover the full Simnon language, for example to include standard library functions, sin, cos, etc. The code generation can also be modified. Other types of code may be of interest. One example is generation of C-code for the scheduler described in (Brück, 1988). The generated code could be of the same structure as in the Modula-2 case, including both the control algorithm, and access procedures.

3.7 Controller Implementation Efficiency

The efficiency of the compiler when used as a tool for controller implementation has been measured in terms of code size. The size of the generated code is compared with the size of the Simnon system. A number of control algorithms, including PI, PID, polynomial controller, and controller with feedback from estimated states, was translated from Simnon to Modula-2. The average quotient between the number of lines in the Modula-2 implementation module, and the Simnon system, was 10.8. This is of course only a crude measure, but it illustrates how the implementation efficiency is improved compared to hand translation.

4. Conclusions

It is demonstrated how a compiler can be used to simplify the real time implementation of control algorithms. The compiler translates the control algorithm from a design language to an implementation language. The algorithm is written as a discrete Simnon system, and translated by the compiler to a Modula-2 module. Besides translating the algorithm, the compiler also generates access procedures for the variables defined in the Simnon system. The access procedures are used to read and write values, and to make connections between variables and variable names. The chosen set of access procedures gives an environment for controller implementation, where it is possible to design a flexible program for real time control. The user interface can be made independent of the control algorithm, and the control algorithm can be changed without modifying the rest of the program.

5. Acknowledgements

The first version of the compiler was implemented as a project work in a course on AI programming, led by Dr. Wolfgang Kreutzer, University of Canterbury, Christchurch, New Zealand. Prof. Karl Johan Åström and Dr. Lars Nielsen have contributed with valuable discussions during the implementation of the present version.

The work was financially supported by the Swedish National Board for Technical Development.

6. References

AHO, A.V, SETHI, R. AND J.D. ULLMAN (1986): *Compilers – Principles, Techniques and Tools*, Addison-Wesley.

ANDERSSON, L. (1987): "Documentation of kernel and library modules,".

BRÜCK, D.M. (1988): "A Foreground/Background Real-Time Scheduler for the IBM AT," CODEN: LUTFD2/TFRT-7393, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

DYBVIG, R.K. (1987): *The SCHEME Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

ELMQVIST, H., K.J. ÅSTRÖM AND T. SCHÖNTHAL (1986): *SIMNON User's Guide for MS-DOS Computers*, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

LOGITECH, INC. (1986): *Logitech Modula-2, Software Development System, Modula-2 Version 2.0.*

ÅSTRÖM, K.J. (1987): "Implementation of PID Regulators," CODEN: LUTFD2/TFRT-7344, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Appendix - Listing of the Example

```
DISCRETE System PID

STATE i x
NEW ni nx
INPUT r y
OUTPUT u
TIME t
TSAMP ts

"Discrete time PID regulator with anti-windup

u = if ulimon then us else u1
us = if u1<umin then umin else if u1<umax then u1 else umax

u1 = K*b*r - K*y + i + x - bi*y

ni = i + K*h*e/Ti + aw
e = r - y
aw = if awon then h/Tr*(u - u1) else 0

nx = ai*x + bi*(1 - ai)*y
ai = Td/(Td + N*h)
bi = K*Td*N/(Td + N*h)

ts = t + h

ulimon : 0 "Switch for output saturation
awon : 0   "Switch for anti windup

umax : 1e3  "Output limits
umin : -1e3

K : 1     "Regulator gain
Ti : 1    "Integration time
Td : 0    "Derivative time
N : 10    "Derivative filter constant
b : 1     "Feedforward gain
h : 0.1   "Sampling interval
Tr : 1e5  "Anti windup time constant

END
```

```
DEFINITION MODULE pid;

FROM Kernel IMPORT Time;

EXPORT QUALIFIED

  SystemType, StateType, NewType, InputType, OutputType,
  ParType, AuxVarType,

  WriteState, WriteStateVariable,
  ReadState, ReadStateVariable,
  GetNumberOfStates, GetStateName,

  WriteNew, WriteNewVariable,
  ReadNew, ReadNewVariable,
  GetNumberOfNews, GetNewName,

  WriteInput, WriteInputVariable,
  ReadInput, ReadInputVariable,
  GetNumberOfInputs, GetInputName,

  WriteOutput, WriteOutputVariable,
  ReadOutput, ReadOutputVariable,
  GetNumberOfOutputs, GetOutputName,

  WritePar, WriteParVariable,
  ReadPar, ReadParVariable,
  GetNumberOfPars, GetParName,

  WriteAuxVar, WriteAuxVarVariable,
  ReadAuxVar, ReadAuxVarVariable,
  GetNumberOfAuxVars, GetAuxVarName,

  IncrementTime, Init, UpDate;

TYPE

SystemType;

StateType = RECORD
  i, x : REAL;
END;

NewType = RECORD
  ni, nx : REAL;
END;

InputType = RECORD
  r, y : REAL;
END;

OutputType = RECORD
  u : REAL;
END;

ParType = RECORD
  ulimon, awon, umax, umin, k, ti, td, n, b, h, tr : REAL;
END;

AuxVarType = RECORD
  us, ui, e, aw, ai, bi : REAL;
END;

PROCEDURE WriteState(S : SystemType; VAR State : StateType);

PROCEDURE ReadState(S : SystemType; VAR State : StateType);

PROCEDURE WriteStateVariable(
  VAR State : StateType;
  Name : ARRAY OF CHAR;
  Value : REAL;
```

```
  VAR NameOk : BOOLEAN);

PROCEDURE ReadStateVariable(
  VAR State : StateType;
  Name : ARRAY OF CHAR;
  VAR Value : REAL;
  VAR NameOk : BOOLEAN);

PROCEDURE GetNumberOfStates() : CARDINAL;

PROCEDURE GetStateName(
  Number : CARDINAL; VAR Name : ARRAY OF CHAR;
  VAR NumberOk : BOOLEAN);

PROCEDURE WriteNew(S : SystemType; VAR New : NewType);

PROCEDURE ReadNew(S : SystemType; VAR New : NewType);

PROCEDURE WriteNewVariable(
  VAR New : NewType;
  Name : ARRAY OF CHAR;
  Value : REAL;
  VAR NameOk : BOOLEAN);

PROCEDURE ReadNewVariable(
  VAR New : NewType;
  Name : ARRAY OF CHAR;
  VAR Value : REAL;
  VAR NameOk : BOOLEAN);

PROCEDURE GetNumberOfNews() : CARDINAL;

PROCEDURE GetNewName(
  Number : CARDINAL; VAR Name : ARRAY OF CHAR;
  VAR NumberOk : BOOLEAN);

PROCEDURE WriteInput(S : SystemType; VAR Input : InputType);

PROCEDURE ReadInput(S : SystemType; VAR Input : InputType);

PROCEDURE WriteInputVariable(
  VAR Input : InputType;
  Name : ARRAY OF CHAR;
  Value : REAL;
  VAR NameOk : BOOLEAN);

PROCEDURE ReadInputVariable(
  VAR Input : InputType;
  Name : ARRAY OF CHAR;
  VAR Value : REAL;
  VAR NameOk : BOOLEAN);

PROCEDURE GetNumberOfInputs() : CARDINAL;

PROCEDURE GetInputName(
  Number : CARDINAL; VAR Name : ARRAY OF CHAR;
  VAR NumberOk : BOOLEAN);

PROCEDURE WriteOutput(S : SystemType; VAR Output : OutputType);

PROCEDURE ReadOutput(S : SystemType; VAR Output : OutputType);

PROCEDURE WriteOutputVariable(
  VAR Output : OutputType;
  Name : ARRAY OF CHAR;
  Value : REAL;
  VAR NameOk : BOOLEAN);

PROCEDURE ReadOutputVariable(
  VAR Output : OutputType;
  Name : ARRAY OF CHAR;
```

```
  VAR Value : REAL;
  VAR NameOk : BOOLEAN);

PROCEDURE GetNumberOfOutputs() : CARDINAL;

PROCEDURE GetOutputName(
  Number : CARDINAL; VAR Name : ARRAY OF CHAR;
  VAR NumberOk : BOOLEAN);

PROCEDURE WritePar(S : SystemType; VAR Par : ParType);

PROCEDURE ReadPar(S : SystemType; VAR Par : ParType);

PROCEDURE WriteParVariable(
  VAR Par : ParType;
  Name : ARRAY OF CHAR;
  Value : REAL;
  VAR NameOk : BOOLEAN);

PROCEDURE ReadParVariable(
  VAR Par : ParType;
  Name : ARRAY OF CHAR;
  VAR Value : REAL;
  VAR NameOk : BOOLEAN);

PROCEDURE GetNumberOfPars() : CARDINAL;

PROCEDURE GetParName(
  Number : CARDINAL; VAR Name : ARRAY OF CHAR;
  VAR NumberOk : BOOLEAN);

PROCEDURE WriteAuxVar(S : SystemType; VAR AuxVar : AuxVarType);

PROCEDURE ReadAuxVar(S : SystemType; VAR AuxVar : AuxVarType);

PROCEDURE WriteAuxVarVariable(
  VAR AuxVar : AuxVarType;
  Name : ARRAY OF CHAR;
  Value : REAL;
  VAR NameOk : BOOLEAN);

PROCEDURE ReadAuxVarVariable(
  VAR AuxVar : AuxVarType;
  Name : ARRAY OF CHAR;
  VAR Value : REAL;
  VAR NameOk : BOOLEAN);

PROCEDURE GetNumberOfAuxVars() : CARDINAL;

PROCEDURE GetAuxVarName(
  Number : CARDINAL; VAR Name : ARRAY OF CHAR;
  VAR NumberOk : BOOLEAN);

PROCEDURE IncrementTime(S : SystemType; VAR T : Time);

PROCEDURE Init(VAR S : SystemType);

PROCEDURE UpDate(S : SystemType);

END pid.
```

```
IMPLEMENTATION MODULE pid;

FROM Storage IMPORT ALLOCATE;
FROM Strings IMPORT CompareStr, Copy;
FROM MathLib IMPORT round;
FROM Kernel IMPORT
  Semaphore, InitSem, Wait, Signal, Time, IncTime;

CONST

NameLength = 20;

NumberOfStates = 2;

NumberOfNews = 2;

NumberOfInputs = 2;

NumberOfOutputs = 1;

NumberOfPars = 11;

NumberOfAuxVars = 6;

TYPE

SystemType = POINTER TO RECORD
  Mutex : Semaphore;
  State : StateType;
  New : NewType;
  Input : InputType;
  Output : OutputType;
  Par : ParType;
  AuxVar : AuxVarType;
END;

PROCEDURE WriteState(S : SystemType; VAR State : StateType);
BEGIN
  Wait(S^.Mutex);
  S^.State := State;
  Signal(S^.Mutex);
END WriteState;

PROCEDURE ReadState(S : SystemType; VAR State : StateType);
BEGIN
  Wait(S^.Mutex);
  State := S^.State;
  Signal(S^.Mutex);
END ReadState;

PROCEDURE WriteStateVariable(
  VAR State : StateType;
  Name : ARRAY OF CHAR;
  Value : REAL;
  VAR NameOk : BOOLEAN);
BEGIN
  NameOk := TRUE;
  IF CompareStr(Name,'i') = 0 THEN
    State.i := Value;
  ELSIF CompareStr(Name,'x') = 0 THEN
    State.x := Value;
  ELSE
    NameOk := FALSE;
  END;
END WriteStateVariable;

PROCEDURE ReadStateVariable(
  VAR State : StateType;
  Name : ARRAY OF CHAR;
  VAR Value : REAL;
  VAR NameOk : BOOLEAN);
```

```
BEGIN
  Value := 0.0;
  NameOk := TRUE;
  IF CompareStr(Name,'i') = 0 THEN
    Value := State.i;
  ELSIF CompareStr(Name,'x') = 0 THEN
    Value := State.x;
  ELSE
    NameOk := FALSE;
  END;
END ReadStateVariable;

PROCEDURE GetNumberOfStates() : CARDINAL;
BEGIN
  RETURN NumberOfStates;
END GetNumberOfStates;

PROCEDURE GetStateName(
  Number : CARDINAL; VAR Name : ARRAY OF CHAR;
  VAR NumberOk : BOOLEAN);
BEGIN
  NumberOk := TRUE;
  Copy(' ',0,1,Name);
  IF (Number <= NumberOfStates) AND (Number > 0) THEN
    CASE Number OF
      1 : Copy('i',0,NameLength,Name); |
      2 : Copy('x',0,NameLength,Name);
    END;
  ELSE
    NumberOk := FALSE;
  END;
END GetStateName;

PROCEDURE WriteNew(S : SystemType; VAR New : NewType);
BEGIN
  Wait(S^.Mutex);
  S^.New := New;
  Signal(S^.Mutex);
END WriteNew;

PROCEDURE ReadNew(S : SystemType; VAR New : NewType);
BEGIN
  Wait(S^.Mutex);
  New := S^.New;
  Signal(S^.Mutex);
END ReadNew;

PROCEDURE WriteNewVariable(
  VAR New : NewType;
  Name : ARRAY OF CHAR;
  Value : REAL;
  VAR NameOk : BOOLEAN);
BEGIN
  NameOk := TRUE;
  IF CompareStr(Name,'ni') = 0 THEN
    New.ni := Value;
  ELSIF CompareStr(Name,'nx') = 0 THEN
    New.nx := Value;
  ELSE
    NameOk := FALSE;
  END;
END WriteNewVariable;

PROCEDURE ReadNewVariable(
  VAR New : NewType;
  Name : ARRAY OF CHAR;
  VAR Value : REAL;
  VAR NameOk : BOOLEAN);
BEGIN
  Value := 0.0;
  NameOk := TRUE;
```

```
  IF CompareStr(Name,'ni') = 0 THEN
    Value := New.ni;
  ELSIF CompareStr(Name,'nx') = 0 THEN
    Value := New.nx;
  ELSE
    NameOk := FALSE;
  END;
END ReadNewVariable;

PROCEDURE GetNumberOfNews() : CARDINAL;
BEGIN
  RETURN NumberOfNews;
END GetNumberOfNews;

PROCEDURE GetNewName(
  Number : CARDINAL; VAR Name : ARRAY OF CHAR;
  VAR NumberOk : BOOLEAN);
BEGIN
  NumberOk := TRUE;
  Copy(' ',0,1,Name);
  IF (Number <= NumberOfNews) AND (Number > 0) THEN
    CASE Number OF
      1 : Copy('ni',0,NameLength,Name); |
      2 : Copy('nx',0,NameLength,Name);
    END;
  ELSE
    NumberOk := FALSE;
  END;
END GetNewName;

PROCEDURE WriteInput(S : SystemType; VAR Input : InputType);
BEGIN
  Wait(S^.Mutex);
  S^.Input := Input;
  Signal(S^.Mutex);
END WriteInput;

PROCEDURE ReadInput(S : SystemType; VAR Input : InputType);
BEGIN
  Wait(S^.Mutex);
  Input := S^.Input;
  Signal(S^.Mutex);
END ReadInput;

PROCEDURE WriteInputVariable(
  VAR Input : InputType;
  Name : ARRAY OF CHAR;
  Value : REAL;
  VAR NameOk : BOOLEAN);
BEGIN
  NameOk := TRUE;
  IF CompareStr(Name,'r') = 0 THEN
    Input.r := Value;
  ELSIF CompareStr(Name,'y') = 0 THEN
    Input.y := Value;
  ELSE
    NameOk := FALSE;
  END;
END WriteInputVariable;

PROCEDURE ReadInputVariable(
  VAR Input : InputType;
  Name : ARRAY OF CHAR;
  VAR Value : REAL;
  VAR NameOk : BOOLEAN);
BEGIN
  Value := 0.0;
  NameOk := TRUE;
  IF CompareStr(Name,'r') = 0 THEN
    Value := Input.r;
  ELSIF CompareStr(Name,'y') = 0 THEN
```

```
    Value := Input.y;
  ELSE
    NameOk := FALSE;
  END;
END ReadInputVariable;

PROCEDURE GetNumberOfInputs() : CARDINAL;
BEGIN
  RETURN NumberOfInputs;
END GetNumberOfInputs;

PROCEDURE GetInputName(
  Number : CARDINAL; VAR Name : ARRAY OF CHAR;
  VAR NumberOk : BOOLEAN);
BEGIN
  NumberOk := TRUE;
  Copy(' ',0,1,Name);
  IF (Number <= NumberOfInputs) AND (Number > 0) THEN
    CASE Number OF
      1 : Copy('r',0,NameLength,Name); |
      2 : Copy('y',0,NameLength,Name);
    END;
  ELSE
    NumberOk := FALSE;
  END;
END GetInputName;

PROCEDURE WriteOutput(S : SystemType; VAR Output : OutputType);
BEGIN
  Wait(S^.Mutex);
  S^.Output := Output;
  Signal(S^.Mutex);
END WriteOutput;

PROCEDURE ReadOutput(S : SystemType; VAR Output : OutputType);
BEGIN
  Wait(S^.Mutex);
  Output := S^.Output;
  Signal(S^.Mutex);
END ReadOutput;

PROCEDURE WriteOutputVariable(
  VAR Output : OutputType;
  Name : ARRAY OF CHAR;
  Value : REAL;
  VAR NameOk : BOOLEAN);
BEGIN
  NameOk := TRUE;
  IF CompareStr(Name,'u') = 0 THEN
    Output.u := Value;
  ELSE
    NameOk := FALSE;
  END;
END WriteOutputVariable;

PROCEDURE ReadOutputVariable(
  VAR Output : OutputType;
  Name : ARRAY OF CHAR;
  VAR Value : REAL;
  VAR NameOk : BOOLEAN);
BEGIN
  Value := 0.0;
  NameOk := TRUE;
  IF CompareStr(Name,'u') = 0 THEN
    Value := Output.u;
  ELSE
    NameOk := FALSE;
  END;
END ReadOutputVariable;

PROCEDURE GetNumberOfOutputs() : CARDINAL;
```

```
BEGIN
  RETURN NumberOfOutputs;
END GetNumberOfOutputs;

PROCEDURE GetOutputName(
  Number : CARDINAL; VAR Name : ARRAY OF CHAR;
  VAR NumberOk : BOOLEAN);
BEGIN
  NumberOk := TRUE;
  Copy(' ',0,1,Name);
  IF (Number <= NumberOfOutputs) AND (Number > 0) THEN
    CASE Number OF
      1 : Copy('u',0,NameLength,Name);
    END;
  ELSE
    NumberOk := FALSE;
  END;
END GetOutputName;

PROCEDURE WritePar(S : SystemType; VAR Par : ParType);
BEGIN
  Wait(S^.Mutex);
  S^.Par := Par;
  Signal(S^.Mutex);
END WritePar;

PROCEDURE ReadPar(S : SystemType; VAR Par : ParType);
BEGIN
  Wait(S^.Mutex);
  Par := S^.Par;
  Signal(S^.Mutex);
END ReadPar;

PROCEDURE WriteParVariable(
  VAR Par : ParType;
  Name : ARRAY OF CHAR;
  Value : REAL;
  VAR NameOk : BOOLEAN);
BEGIN
  NameOk := TRUE;
  IF CompareStr(Name,'ulimon') = 0 THEN
    Par.ulimon := Value;
  ELSIF CompareStr(Name,'awon') = 0 THEN
    Par.awon := Value;
  ELSIF CompareStr(Name,'umax') = 0 THEN
    Par.umax := Value;
  ELSIF CompareStr(Name,'umin') = 0 THEN
    Par.umin := Value;
  ELSIF CompareStr(Name,'k') = 0 THEN
    Par.k := Value;
  ELSIF CompareStr(Name,'ti') = 0 THEN
    Par.ti := Value;
  ELSIF CompareStr(Name,'td') = 0 THEN
    Par.td := Value;
  ELSIF CompareStr(Name,'n') = 0 THEN
    Par.n := Value;
  ELSIF CompareStr(Name,'b') = 0 THEN
    Par.b := Value;
  ELSIF CompareStr(Name,'h') = 0 THEN
    Par.h := Value;
  ELSIF CompareStr(Name,'tr') = 0 THEN
    Par.tr := Value;
  ELSE
    NameOk := FALSE;
  END;
END WriteParVariable;

PROCEDURE ReadParVariable(
  VAR Par : ParType;
  Name : ARRAY OF CHAR;
  VAR Value : REAL;
```

```
  VAR NameOk : BOOLEAN);
BEGIN
  Value := 0.0;
  NameOk := TRUE;
  IF CompareStr(Name,'ulimon') = 0 THEN
    Value := Par.ulimon;
  ELSIF CompareStr(Name,'awon') = 0 THEN
    Value := Par.awon;
  ELSIF CompareStr(Name,'umax') = 0 THEN
    Value := Par.umax;
  ELSIF CompareStr(Name,'umin') = 0 THEN
    Value := Par.umin;
  ELSIF CompareStr(Name,'k') = 0 THEN
    Value := Par.k;
  ELSIF CompareStr(Name,'ti') = 0 THEN
    Value := Par.ti;
  ELSIF CompareStr(Name,'td') = 0 THEN
    Value := Par.td;
  ELSIF CompareStr(Name,'n') = 0 THEN
    Value := Par.n;
  ELSIF CompareStr(Name,'b') = 0 THEN
    Value := Par.b;
  ELSIF CompareStr(Name,'h') = 0 THEN
    Value := Par.h;
  ELSIF CompareStr(Name,'tr') = 0 THEN
    Value := Par.tr;
  ELSE
    NameOk := FALSE;
  END;
END ReadParVariable;

PROCEDURE GetNumberOfPars() : CARDINAL;
BEGIN
  RETURN NumberOfPars;
END GetNumberOfPars;

PROCEDURE GetParName(
  Number : CARDINAL; VAR Name : ARRAY OF CHAR;
  VAR NumberOk : BOOLEAN);
BEGIN
  NumberOk := TRUE;
  Copy(' ',0,1,Name);
  IF (Number <= NumberOfPars) AND (Number > 0) THEN
    CASE Number OF
      1 : Copy('ulimon',0,NameLength,Name); |
      2 : Copy('awon',0,NameLength,Name); |
      3 : Copy('umax',0,NameLength,Name); |
      4 : Copy('umin',0,NameLength,Name); |
      5 : Copy('k',0,NameLength,Name); |
      6 : Copy('ti',0,NameLength,Name); |
      7 : Copy('td',0,NameLength,Name); |
      8 : Copy('n',0,NameLength,Name); |
      9 : Copy('b',0,NameLength,Name); |
      10 : Copy('h',0,NameLength,Name); |
      11 : Copy('tr',0,NameLength,Name);
    END;
  ELSE
    NumberOk := FALSE;
  END;
END GetParName;

PROCEDURE WriteAuxVar(S : SystemType; VAR AuxVar : AuxVarType);
BEGIN
  Wait(S^.Mutex);
  S^.AuxVar := AuxVar;
  Signal(S^.Mutex);
END WriteAuxVar;

PROCEDURE ReadAuxVar(S : SystemType; VAR AuxVar : AuxVarType);
BEGIN
  Wait(S^.Mutex);
```

```
  AuxVar := S^.AuxVar;
  Signal(S^.Mutex);
END ReadAuxVar;

PROCEDURE WriteAuxVarVariable(
  VAR AuxVar : AuxVarType;
  Name : ARRAY OF CHAR;
  Value : REAL;
  VAR NameOk : BOOLEAN);
BEGIN
  NameOk := TRUE;
  IF CompareStr(Name,'us') = 0 THEN
    AuxVar.us := Value;
  ELSIF CompareStr(Name,'ul') = 0 THEN
    AuxVar.ul := Value;
  ELSIF CompareStr(Name,'e') = 0 THEN
    AuxVar.e := Value;
  ELSIF CompareStr(Name,'aw') = 0 THEN
    AuxVar.aw := Value;
  ELSIF CompareStr(Name,'ai') = 0 THEN
    AuxVar.ai := Value;
  ELSIF CompareStr(Name,'bi') = 0 THEN
    AuxVar.bi := Value;
  ELSE
    NameOk := FALSE;
  END;
END WriteAuxVarVariable;

PROCEDURE ReadAuxVarVariable(
  VAR AuxVar : AuxVarType;
  Name : ARRAY OF CHAR;
  VAR Value : REAL;
  VAR NameOk : BOOLEAN);
BEGIN
  Value := 0.0;
  NameOk := TRUE;
  IF CompareStr(Name,'us') = 0 THEN
    Value := AuxVar.us;
  ELSIF CompareStr(Name,'ul') = 0 THEN
    Value := AuxVar.ul;
  ELSIF CompareStr(Name,'e') = 0 THEN
    Value := AuxVar.e;
  ELSIF CompareStr(Name,'aw') = 0 THEN
    Value := AuxVar.aw;
  ELSIF CompareStr(Name,'ai') = 0 THEN
    Value := AuxVar.ai;
  ELSIF CompareStr(Name,'bi') = 0 THEN
    Value := AuxVar.bi;
  ELSE
    NameOk := FALSE;
  END;
END ReadAuxVarVariable;

PROCEDURE GetNumberOfAuxVars() : CARDINAL;
BEGIN
  RETURN NumberOfAuxVars;
END GetNumberOfAuxVars;

PROCEDURE GetAuxVarName(
  Number : CARDINAL; VAR Name : ARRAY OF CHAR;
  VAR NumberOk : BOOLEAN);
BEGIN
  NumberOk := TRUE;
  Copy(' ',0,1,Name);
  IF (Number <= NumberOfAuxVars) AND (Number > 0) THEN
    CASE Number OF
      1 : Copy('us',0,NameLength,Name); |
      2 : Copy('ul',0,NameLength,Name); |
      3 : Copy('e',0,NameLength,Name); |
      4 : Copy('aw',0,NameLength,Name); |
      5 : Copy('ai',0,NameLength,Name); |
```

```
      6 : Copy('bi',0,NameLength,Name);
    END;
  ELSE
    NumberOk := FALSE;
  END;
END GetAuxVarName;

PROCEDURE IncrementTime(S : SystemType; VAR T : Time);
BEGIN
  Wait(S^.Mutex);
  WITH S^.Par DO
    IncTime(T,round(1000.0*h));
  END;
  Signal(S^.Mutex);
END IncrementTime;

PROCEDURE Init(VAR S : SystemType);
BEGIN
  NEW(S);
  InitSem(S^.Mutex,1);
  WITH S^.State DO
    i := 0.0;
    x := 0.0;
  END;
  WITH S^.New DO
    ni := 0.0;
    nx := 0.0;
  END;
  WITH S^.Input DO
    r := 0.0;
    y := 0.0;
  END;
  WITH S^.Output DO
    u := 0.0;
  END;
  WITH S^.Par DO
    tr := 1.0e5;
    h := 0.1;
    b := 1.0;
    n := 10.0;
    td := 0.0;
    ti := 1.0;
    k := 1.0;
    umin :=  - 1.0e3;
    umax := 1.0e3;
    awon := 0.0;
    ulimon := 0.0;
  END;
  WITH S^.AuxVar DO
    us := 0.0;
    u1 := 0.0;
    e := 0.0;
    aw := 0.0;
    ai := 0.0;
    bi := 0.0;
  END;
END Init;

PROCEDURE UpDate(S : SystemType);
BEGIN
  Wait(S^.Mutex);
  WITH S^.State DO WITH S^.New DO WITH S^.Input DO
  WITH S^.Output DO WITH S^.Par DO WITH S^.AuxVar DO
    e := r - y;
    ai := td / (td + n * h);
    bi := k * td * n / (td + n * h);
    nx := ai * x + bi * (1.0 - ai) * y;
    u1 := k * b * r - k * y + i + x - bi * y;
    IF u1 < umin THEN
      us := umin;
    ELSIF u1 < umax THEN
```

```
      us := ui;
    ELSE
      us := umax;
    END;
    IF ulimon > 0.5 THEN
      u := us;
    ELSE
      u := ui;
    END;
    IF awon > 0.5 THEN
      aw := h / tr * (u - ui);
    ELSE
      aw := 0.0;
    END;
    ni := i + k * h * e / ti + aw;
    i := ni;
    x := nx;
  END; END; END; END; END; END;
  Signal(S^.Mutex);
END UpDate;

END pid.
```

```
MODULE Main;

(*   A simple program for single loop control  *)

(*   The program implements a general single loop controller  *)
(*   with simple user interaction. It is assumed that the     *)
(*   control algorithm is written in a standardized format.   *)
(*   This makes it easy to change control algorithm, just     *)
(*   change the module name in the first import statement.    *)
(*   The rest of the code is independent of the particular    *)
(*   algorithm used. The interaction is command driven with   *)
(*   commands for reading and writing parameters and inputs,  *)
(*   and for reading outputs. There are also commands for     *)
(*   connecting inputs and outputs to AD and DA converters.   *)

FROM pid IMPORT
  SystemType, InputType, OutputType, ParType,
  WriteInput, WriteInputVariable,
  ReadInput, ReadInputVariable,
  ReadOutput, ReadOutputVariable,
  WritePar, WriteParVariable,
  ReadPar, ReadParVariable,
  IncrementTime, Init, UpDate;

FROM TextWindows IMPORT
  WindowType, MakeTextWindow, WriteLine, ReadLine,
  WriteText, WriteLn, WriteReal, ReadReal;

FROM Misc IMPORT
  StringAssign, StringEq;

FROM MathLib IMPORT round;

IMPORT RTMouse;

FROM Kernel IMPORT
  CreateProcess, Semaphore, InitSem, Wait, Signal,
  SetPriority, Time, GetTime, IncTime, WaitUntil;

FROM AnalogIO IMPORT ADIn, DAOut;

CONST MaxConnections = 30;

TYPE

ConnectionData = RECORD
                   Name : ARRAY [0..20] OF CHAR;
                   Channel : CARDINAL;
                   Connected : BOOLEAN;
                 END;

ConnectionVector = ARRAY [1..MaxConnections] OF ConnectionData;

MonitorData =
  RECORD
    Running : BOOLEAN;
    ConnectedInputs,
    ConnectedOutputs : ConnectionVector;
  END;

CommandType = (ReadInputVar, WriteInputVar, ConnectInputVar,
               ReadOutputVar, ConnectOutputVar,
               ReadParVar, WriteParVar,
               RunController, StopController,
               Exit, NoCommand, ErrorCommand);

CONST NumberOfCommands = 12;

VAR

TheController : SystemType;
```

```
ControllerInputs : InputType;
ControllerOutputs : OutputType;
ControllerParameters : ParType;

ControllerMonitor : RECORD
                      Mutex : Semaphore;
                      ControllerData : MonitorData;
                    END;

MenuWindow, CommandWindow : WindowType;
CommandTexts : ARRAY CommandType OF ARRAY [0..30] OF CHAR;

TheEnd : Semaphore;

PROCEDURE InitMain;
VAR k : CARDINAL;
BEGIN
  RTMouse.Init;
  Init(TheController);
  InitSem(ControllerMonitor.Mutex,1);
  WITH ControllerMonitor.ControllerData DO
    Running := FALSE;
    FOR k := 1 TO MaxConnections DO
      StringAssign(" ",ConnectedInputs[k].Name);
      StringAssign(" ",ConnectedOutputs[k].Name);
      ConnectedInputs[k].Channel := 0;
      ConnectedOutputs[k].Channel := 0;
      ConnectedInputs[k].Connected := FALSE;
      ConnectedOutputs[k].Connected := FALSE;
    END;
  END;
  MenuWindow := MakeTextWindow(0.1,0.25,0.7,0.9);
  CommandWindow := MakeTextWindow(0.1,0.05,0.7,0.2);
  InitCommandTexts;
  DisplayMenu;
  InitSem(TheEnd,0);
  CreateProcess(Controller,5000);
  CreateProcess(OpCom,5000);
END InitMain;

(* Process *) PROCEDURE Controller;
VAR Inputs : InputType;
    Outputs : OutputType;
    k : CARDINAL;
    Next : Time;
    ControllerData : MonitorData;
    Dummy : BOOLEAN;
    OutValue : REAL;
BEGIN
  SetPriority(10);
  GetTime(Next);
  LOOP
    Wait(ControllerMonitor.Mutex);
    ControllerData := ControllerMonitor.ControllerData;
    Signal(ControllerMonitor.Mutex);
    IF ControllerData.Running THEN
      WITH ControllerData DO
        ReadInput(TheController,Inputs);
        FOR k := 1 TO MaxConnections DO
          WITH ConnectedInputs[k] DO
            IF Connected THEN
              WriteInputVariable(Inputs,Name,ADIn(Channel),Dummy);
            END;
          END;
        END;
        WriteInput(TheController,Inputs);
        UpDate(TheController);
        ReadOutput(TheController,Outputs);
        FOR k := 1 TO MaxConnections DO
          WITH ConnectedOutputs[k] DO
            IF Connected THEN
```

```
                ReadOutputVariable(Outputs,Name,OutValue,Dummy);
                DAOut(Channel,OutValue);
              END;
            END;
          END;
        END; (* WITH ControllerData *)
      END; (* IF ControllerData.Running *)
      IncrementTime(TheController,Next);
      WaitUntil(Next);
    END; (* LOOP *)
  END Controller;

  (* Process *) PROCEDURE OpCom;
  VAR Command : CommandType;
  BEGIN
    SetPriority(20);
    LOOP
      Command := GetCommand();
      DecodeCommand(Command);
    END;
  END OpCom;

  PROCEDURE InitCommandTexts;
  BEGIN
    StringAssign("readinput",CommandTexts[ReadInputVar]);
    StringAssign("writeinput",CommandTexts[WriteInputVar]);
    StringAssign("connectinput",CommandTexts[ConnectInputVar]);
    StringAssign("readoutput",CommandTexts[ReadOutputVar]);
    StringAssign("connectoutput",CommandTexts[ConnectOutputVar]);
    StringAssign("readpar",CommandTexts[ReadParVar]);
    StringAssign("writepar",CommandTexts[WriteParVar]);
    StringAssign("run",CommandTexts[RunController]);
    StringAssign("stop",CommandTexts[StopController]);
    StringAssign("exit",CommandTexts[Exit]);
    CommandTexts[NoCommand,0] := CHR(0);
  END InitCommandTexts;

  PROCEDURE GetCommand() : CommandType;
  VAR CommandText : ARRAY [0..30] OF CHAR;
      k, Command : CommandType;
      Found : BOOLEAN;
  BEGIN
    ReadLine(CommandWindow,"> ",CommandText);
    k := VAL(CommandType,0);
    Found := FALSE;
    Command := NoCommand;
    WHILE (k <= VAL(CommandType,NumberOfCommands-2)) AND NOT Found DO
      Found := StringEq(CommandText,CommandTexts[k]);
      IF Found THEN
        Command := k;
      END;
      INC(k);
    END;
    IF Found THEN
      RETURN Command;
    ELSE
      RETURN ErrorCommand;
    END;
  END GetCommand;

  PROCEDURE DecodeCommand(Command : CommandType);
  BEGIN
    CASE Command OF
      ReadInputVar : DecodeReadInputVar; |
      WriteInputVar : DecodeWriteInputVar; |
      ConnectInputVar : DecodeConnectInputVar; |
      ReadOutputVar : DecodeReadOutputVar; |
      ConnectOutputVar : DecodeConnectOutputVar; |
      ReadParVar : DecodeReadParVar; |
      WriteParVar : DecodeWriteParVar; |
      RunController : DecodeRunController; |
```

```
    StopController : DecodeStopController; |
    Exit : DecodeExit; |
    NoCommand : DecodeNoCommand; |
    ErrorCommand : DecodeErrorCommand;
  END;
END DecodeCommand;

PROCEDURE DecodeReadInputVar;
VAR Name : ARRAY [0..30] OF CHAR;
    NameOk : BOOLEAN;
    Value : REAL;
BEGIN
  ReadLine(CommandWindow,"Input name > ",Name);
  ReadInput(TheController,ControllerInputs);
  ReadInputVariable(ControllerInputs,Name,Value,NameOk);
  IF NOT NameOk THEN
    WriteLine(CommandWindow,"ERROR : Illegal input name");
  ELSE
    WriteText(CommandWindow,Name);
    WriteText(CommandWindow," = ");
    WriteReal(CommandWindow,Value);
    WriteLn(CommandWindow);
  END;
END DecodeReadInputVar;

PROCEDURE DecodeWriteInputVar;
VAR Name : ARRAY [0..30] OF CHAR;
    NameOk : BOOLEAN;
    Value : REAL;
    ValueOk : BOOLEAN;
BEGIN
  ReadLine(CommandWindow,"Input name > ",Name);
  ReadInput(TheController,ControllerInputs);
  ReadInputVariable(ControllerInputs,Name,Value,NameOk);
  IF NOT NameOk THEN
    WriteLine(CommandWindow,"ERROR : Illegal input name");
  ELSE
    ReadReal(CommandWindow,"Value > ",Value,ValueOk);
    IF NOT ValueOk THEN
      WriteLine(CommandWindow,"ERROR : Real number expected");
    ELSE
      WriteInputVariable(ControllerInputs,Name,Value,NameOk);
      WriteInput(TheController,ControllerInputs);
    END;
  END;
END DecodeWriteInputVar;

PROCEDURE DecodeConnectInputVar;
VAR Name : ARRAY [0..30] OF CHAR;
    NameOk : BOOLEAN;
    Value : REAL;
    RealChannel : REAL;
    RealChannelOk : BOOLEAN;
    Channel : CARDINAL;
    ChannelOk : BOOLEAN;
    k : CARDINAL;
    Found : BOOLEAN;
BEGIN
  ReadLine(CommandWindow,"Input name > ",Name);
  ReadInput(TheController,ControllerInputs);
  ReadInputVariable(ControllerInputs,Name,Value,NameOk);
  IF NOT NameOk THEN
    WriteLine(CommandWindow,"ERROR : Illegal input name");
  ELSE
    ReadReal(CommandWindow,"Channel > ",RealChannel,RealChannelOk);
    IF NOT RealChannelOk THEN
      WriteLine(CommandWindow,"ERROR : Real number expected");
    ELSE
      Channel := round(RealChannel);
      ChannelOk := (Channel < 4);
      IF NOT ChannelOk THEN
```

```
        WriteLine(CommandWindow,"ERROR : Illegal channel");
      ELSE
        WITH ControllerMonitor DO
          WITH ControllerData DO
            k := 1;
            Found := FALSE;
            WHILE (k < MaxConnections) AND NOT Found DO
              Wait(Mutex);
              IF NOT ConnectedInputs[k].Connected THEN
                Found := TRUE;
                ConnectedInputs[k].Connected := TRUE;
                StringAssign(Name,ConnectedInputs[k].Name);
                ConnectedInputs[k].Channel := Channel;
              END;
              Signal(Mutex);
              INC(k);
            END;
            IF NOT Found THEN
              WriteLine(CommandWindow,"ERROR : no available connection");
            END;
          END; (* WITH *)
        END; (* WITH *)
      END; (* IF NOT ChannelOK *)
    END; (* IF NOT RealChannelOk *)
  END; (* IF NOT NameOk *)
END DecodeConnectInputVar;

PROCEDURE DecodeReadOutputVar;
VAR Name : ARRAY [0..30] OF CHAR;
    NameOk : BOOLEAN;
    Value : REAL;
BEGIN
  ReadLine(CommandWindow,"Output name > ",Name);
  ReadOutput(TheController,ControllerOutputs);
  ReadOutputVariable(ControllerOutputs,Name,Value,NameOk);
  IF NOT NameOk THEN
    WriteLine(CommandWindow,"ERROR : Illegal output name");
  ELSE
    WriteText(CommandWindow,Name);
    WriteText(CommandWindow," = ");
    WriteReal(CommandWindow,Value);
    WriteLn(CommandWindow);
  END;
END DecodeReadOutputVar;

PROCEDURE DecodeConnectOutputVar;
VAR Name : ARRAY [0..30] OF CHAR;
    NameOk : BOOLEAN;
    Value : REAL;
    RealChannel : REAL;
    RealChannelOk : BOOLEAN;
    Channel : CARDINAL;
    ChannelOk : BOOLEAN;
    k : CARDINAL;
    Found : BOOLEAN;
BEGIN
  ReadLine(CommandWindow,"Output name > ",Name);
  ReadOutput(TheController,ControllerOutputs);
  ReadOutputVariable(ControllerOutputs,Name,Value,NameOk);
  IF NOT NameOk THEN
    WriteLine(CommandWindow,"ERROR : Illegal output name");
  ELSE
    ReadReal(CommandWindow,"Channel > ",RealChannel,RealChannelOk);
    IF NOT RealChannelOk THEN
      WriteLine(CommandWindow,"ERROR : Real number expected");
    ELSE
      Channel := round(RealChannel);
      ChannelOk := (Channel < 4);
      IF NOT ChannelOk THEN
        WriteLine(CommandWindow,"ERROR : Illegal channel");
      ELSE
```

```
        WITH ControllerMonitor DO
          WITH ControllerData DO
            k := 1;
            Found := FALSE;
            WHILE (k < MaxConnections) AND NOT Found DO
              Wait(Mutex);
              IF NOT ConnectedOutputs[k].Connected THEN
                Found := TRUE;
                ConnectedOutputs[k].Connected := TRUE;
                StringAssign(Name,ConnectedOutputs[k].Name);
                ConnectedOutputs[k].Channel := Channel;
              END;
              Signal(Mutex);
              INC(k);
            END;
            IF NOT Found THEN
              WriteLine(CommandWindow,"ERROR : no available connection");
            END;
          END; (* WITH *)
        END; (* WITH *)
      END; (* IF NOT ChannelOK *)
    END; (* IF NOT RealChannelOk *)
  END; (* IF NOT NameOk *)
END DecodeConnectOutputVar;

PROCEDURE DecodeReadParVar;
VAR Name : ARRAY [0..30] OF CHAR;
    NameOk : BOOLEAN;
    Value : REAL;
BEGIN
  ReadLine(CommandWindow,"Par name > ",Name);
  ReadPar(TheController,ControllerParameters);
  ReadParVariable(ControllerParameters,Name,Value,NameOk);
  IF NOT NameOk THEN
    WriteLine(CommandWindow,"ERROR : Illegal par name");
  ELSE
    WriteText(CommandWindow,Name);
    WriteText(CommandWindow," = ");
    WriteReal(CommandWindow,Value);
    WriteLn(CommandWindow);
  END;
END DecodeReadParVar;

PROCEDURE DecodeWriteParVar;
VAR Name : ARRAY [0..30] OF CHAR;
    NameOk : BOOLEAN;
    Value : REAL;
    ValueOk : BOOLEAN;
BEGIN
  ReadLine(CommandWindow,"Par name > ",Name);
  ReadPar(TheController,ControllerParameters);
  ReadParVariable(ControllerParameters,Name,Value,NameOk);
  IF NOT NameOk THEN
    WriteLine(CommandWindow,"ERROR : Illegal par name");
  ELSE
    ReadReal(CommandWindow,"Value > ",Value,ValueOk);
    IF NOT ValueOk THEN
      WriteLine(CommandWindow,"ERROR : Real number expected");
    ELSE
      WriteParVariable(ControllerParameters,Name,Value,NameOk);
      WritePar(TheController,ControllerParameters);
    END;
  END;
END DecodeWriteParVar;

PROCEDURE DecodeRunController;
BEGIN
  WITH ControllerMonitor DO
    Wait(Mutex);
    ControllerData.Running := TRUE;
    Signal(Mutex);
```

```
  END;
  WriteLine(CommandWindow,"Controller started");
END DecodeRunController;

PROCEDURE DecodeStopController;
BEGIN
  WITH ControllerMonitor DO
    Wait(Mutex);
    ControllerData.Running := FALSE;
    Signal(Mutex);
  END;
  WriteLine(CommandWindow,"Controller stopped");
END DecodeStopController;

PROCEDURE DecodeExit;
BEGIN
  Signal(TheEnd);
END DecodeExit;

PROCEDURE DecodeNoCommand;
BEGIN
END DecodeNoCommand;

PROCEDURE DecodeErrorCommand;
BEGIN
  WriteLine(CommandWindow,"ERROR : Illegal Command");
END DecodeErrorCommand;

PROCEDURE DisplayMenu;
VAR k : CommandType;
BEGIN
  WriteLine(MenuWindow,"Available Commands");
  WriteLn(MenuWindow);
  FOR k := VAL(CommandType,0) TO VAL(CommandType,NumberOfCommands-3) DO
    WriteLine(MenuWindow,CommandTexts[k]);
  END;
END DisplayMenu;

BEGIN
  InitMain;
  Wait(TheEnd);
END Main.
```