State Feedback Control and Data Logging with Modula-2

Wallenborg, Anders

1987

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

Link to publication

Citation for published version (APA):
Wallenborg, A. (1987). *State Feedback Control and Data Logging with Modula-2*. (Technical Reports TFRT-7360). Department of Automatic Control, Lund Institute of Technology (LTH).

Total number of authors:
1

CODEN: LUTFD2/(TFRT-7360)/1-32/(1987)

State Feedback Control and Data Logging with Modula-2

Anders Wallenborg

Department of Automatic Control
Lund Institute of Technology
May 1987

Department of Automatic Control **Lund Institute of Technology** P.O. Box 118 S-221 00 Lund Sweden	*Document name* Report *Date of issue* May 1987 *Document Number* CODEN: LUTFD2/(TFRT-7360)/1-32/(1987)
Author(s) Anders Wallenborg	*Supervisor* *Sponsoring organisation*

Title and subtitle
State Feedback Control and Data Logging with Modula-2.

Abstract

This report describes two real time programs for IBM PC/AT or compatible MS-DOS computers. The programs are written in Modula-2, and use the real time kernel and the graphics module developed at the Department of Automatic Control, LTH.

The first program, DSF, implements a digital state feedback regulator and observer. One major feature of the program is the facility to read regulator parameters from a text file with a format compatible with the parameters files in SIMNON. It is demonstrated how such parameter files can be generated automatically with PC-MATLAB. DSF also includes a screen plot of the measured signals and the control output, and a facility to log the data plotted on the screen. The minimum sampling interval is 30 ms. This limit is set by the time required to plot data on the screen.

The second program, LOG, is an off-spring of the DSF program. It is a program for data logging written to enable sampling intervals down to 10 ms (this is a hard limit set by the current version of the real time kernel). The logged data can be plotted on the screen in real time. Sampling intervals smaller than 20 ms can only be achieved with the screen plot off. A special feature in LOG is the use of the programmable gain facility in the A/D converter boards RT-800 and RT-802 from Analog Devices. Thereby quantization errors can be reduced for low level signals without the use of external amplifiers.

Key words
Real Time Program, Modula-2, State Feedback Control, Data Logging

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title		*ISBN*
Language English	*Number of pages* 32	*Recipient's notes*
Security classification		

The report may be ordered from the Department of Automatic Control or borrowed through the University Library 2, Box 1010, S-221 03 Lund, Sweden, Telex: 33248 lubbis lund.

STATE FEEDBACK CONTROL AND DATA LOGGING WITH MODULA-2

ANDERS WALLENBORG

Department of Automatic Control
Lund Institute of Technology
May 1987

CONTENTS

1. INTRODUCTION

This report describes two real time programs for IBM PC/AT or compatible MS-DOS computers. The programs are written in Modula-2 [Wirth, 1985], [Logitech, 1985], and they use the real time kernel and the graphics module developed by Leif Andersson at the Department of Automatic Control, LTH.

The first program, DSF, implements a digital state feedback regulator and observer. Originally it was written to implement a controller for a flexible servo system [Wallenborg, 1987]. The program has a general structure, however, and can be used to implement a state feedback controller and observer for any process. One major feature of the program is the facility to read regulator parameters from a text file with a format compatible with the parameter files in SIMNON [Elmqvist, Åström and Schönthal, 1986]. Such parameter files can be generated automatically with PC-MATLAB [MathWorks Inc., 1986] and Ctrl-C [Systems Control Technology Inc., 1986]. See Section 2.4 for further details.
This approach enables the control design process to be completely automated. First the state feedback controller is designed with a control analysis and design package, for example PC-MATLAB, and a parameter file is generated. This file can then be loaded into SIMNON to simulate the regulator performance, possibly with a nonlinear process model. Finally the regulator parameters are loaded into the DSF program and the design is tested experimentally. All this can be done without manual input of regulator parameters, which minmizes the risk for fatal parameter input errors etc. DSF also includes a screen plot of the measured signals and the control output, and a facility to log the data plotted on the screen. The lower limit for the sampling interval ($h = 30$ ms) is set by the time required to plot data on the screen.

The second program, LOG, is an off-spring of the DSF program. It is a program for data logging written to enable sampling intervals down to 10 ms (this is a hard limit set by the current version of the real time kernel). The logged data can be plotted on the screen in real time. In the LOG program the log and plot functions have been completely separated, since a 10 ms sampling interval can only be achieved with the screen plot off. A special feature in LOG is the use of the programmable gain facility in the A/D converter boards RT-800 and RT-802 from Analog Devices. Thereby quantization errors can be reduced for low level signals without the use of external amplifiers. As a consequence of this, the LOG program requires a computer equipped with one of these boards. See Section 3.2 for further details.

2. DSF - DIGITAL STATE FEEDBACK CONTROL

DSF implements a digital state feedback regulator and an observer which estimates the state vector. The program includes facilities for real time control, plotting of signals on the screen and data logging. Logged data can be written to a text file. This file can then be converted offline to a binary data file and plotted with IDPAC or SIMNON. The regulator parameters are read from a text file, which has a format compatible with the parameter files in SIMNON. The operator communication is very simple. It is based on a screen menu, where the operator selects the desired command by clicking a mouse button inside one of the menu rectangles.

2.1 SOFTWARE ARCHITECTURE

The program is divided into three process modules and three monitor modules. There is also an additional module called Graphuti, which contains some utility procedures for graphics. Each module is separately compiled.

Monitor modules:

RegData	Regulator data and parameters.
PlotBuff	Plot data buffer.
LogBuff	Log data buffer.

Process modules:

Regulate	The regulator process.
Plot	The plot process.
OpCom	Operator communication.

RegData

RegData implements a monitor in which the regulator parameters are stored in a record of type RegDataType (see Appendix A). The parameters can be accessed by two monitor entry procedures, GetRegData and PutRegdata. This provides mutual exclusion. The RegData module also contains a procedure ReadParFile, which reads regulator parameters from a disk file. ReadParFile is no monitor entry procedure, i.e. it does not put the read parameters into the RegData monitor. This must be done with PutRegData. The file operations have been implemented with procedures from the standard module FileSystem [Logitech, 1985]. Another standard module for formatted input/output is InOut. This module is easier to use, but it cannot be used together with the Graphics module.

PlotBuff

PlotBuff implements a ring buffer monitor with 25 elements. Each element in the buffer is a record containing the data to be plotted, the sampling interval, and the least significant part of the absolute time variable. (See PlotBuff.Def in Appendix A and the type declaration 'Time' in the Kernel module for details). The purpose of this buffer is to buffer plot data while the screen plot area is refreshed. DSF has primarily been used with a sampling interval $h = 40$ ms. A buffer with 25 elements then corresopnds to a time interval of one second. This is sufficient to allow refreshing of the screen after the plot has reached the right end of the screen plot area.

There are five monitor entry procedures: PutPlotBuffer, GetPlotBuffer, EnablePlot, DisablePlot and PlotEnabled. PutPlotBuffer is used to put a new plot data element in the buffer, and GetPlotBuffer reads the oldest data element in the buffer. When the buffer is full, the oldest data element is overwritten by PutPlotBuffer. If the buffer is empty, GetPlotBuffer will wait until data are available, i.e. until PutPlotBuffer has been executed. EnablePlot and DisablePlot set the plot enable flag to true and false, respectively. If the plot enable flag is false, GetPlotBuffer will wait until the flag becomes true. PlotEnabled returns the current state of the plot enable flag (true/false).

LogBuff

LogBuff implements a ring buffer monitor with 500 elements. Each element in the buffer is a record containing the log data and the least significant part of the absolute time variable. (See LogBuff.Def in Appendix A and the type 'Time' in the Kernel module for details). The purpose of this buffer is to log data plotted on the screen. With a sampling interval of h = 40 ms, 500 elements correspond to a time interval of 20 seconds which is the length of the time axis on the screen plot. The buffer length is defined as a constant in the implementation module, and can be changed easily.

There are three monitor entry procedures: PutLogBuffer, GetLogBuffer and WriteLogBuffer. PutLogBuffer is used to put a new log data element in the buffer, and GetLogBuffer reads the oldest data element in the buffer. When the buffer is full the oldest data element is overwritten by PutLogBuffer. If the buffer is empty, GetLogBuffer will wait until data are available, i.e. until PutLogBuffer has been executed. WriteLogBuffer enters the log buffer monitor, copies the current content of the complete log buffer to a local data array, leaves the buffer monitor, and finally writes the copy of the log buffer on an ASCII text file.

The file operations have been implemented with procedures from the standard module FileSystem. See Appendix A for details. Writing data on the disk in real time is a time consuming operation, and so is plotting curves on the screen. Therefore it is recommended to disable the plot process while writing log data on a disk file.

Regulate

The module Regulate contains the regulator process. It calculates a new control output signal based on the current parameters in RegData, and sends plot data to the plot process via the plot buffer PlotBuff. The details of the control algorithm are given in Section 2.2.

Plot

The Plot module contains the plot process. It reads plot data from PlotBuff, plots the data on the screen and sends the plotted data to LogBuff. The plot is frozen by a call to DisablePlot from OpComProcess (cf. PlotBuff above and OpCom below). This means that the plot process will be halted when calling GetPlotBuff. When the plot process is waiting, the log buffer is not updated. Thus the log buffer will only contain the data plotted on the screen. This function was chosen beacuse it provides a way of obtaining hard copies of the curves plotted on the screen by writing the content of the log buffer on a disk file. The plot is restarted by a call to EnablePlot. For each plotted point the current horizontal (time) coordinate is incremented with the current sampling interval. When the right end of the screen is reached, the screen is refreshed and the horizontal coordinate is reset to zero. Hence the screen plot does not use the absolute time in the plot data record. The absolute time is sent to the log buffer, however. If the plot has been frozen this can be detected by a a jump in the absolute time values of the log data.

OpCom

The OpCom module is divided into two processes, OpComProcess and IOProcess. OpComProcess is a high priority process which handles the command decoding and initiates the selected action. Commands are entered by clicking with the mouse in one of the command menu rectangles on the screen. IOProcess has low priority and handles file I/O operations and operator dialog.

Priorities and sampling intervals

The processes have the following priorities (high numbers indicate low priority):

Regulate	10
OpComProcess	15
Plot	20
IOProcess	30

Regulate has the highest priority to ensure that the control algorithm is executed on time. The time consuming IOProcess must have the lowest priority. In particular it must have lower priority than Plot, otherwise the Regulate process may be halted. The reason is that Regulate must enter the PlotBuff monitor via PutPlotBuffer each sampling interval, to send plot data to the plot process. If Plot is interrupted by IOProcess while executing GetPlotBuffer, i.e. while Plot is inside the RegData monitor, then Regulate will be halted.

OpComProcess is normally in a wait state, waiting for a mouse click. It has been given a high priority to ensure fast response to operator commands. OpComProcess may therefore interrupt Plot while inside the PlotBuff monitor, and thus halt the execution of Regulate (cf. above). This can be accepted, however, since the command decoding and execution are very fast operations. One might also consider to give OpComProcess maximum priority. This would guarantee that the program always reacts to operator commands, but it would cause more frequent interrupts of the regulator process.

The minimum sampling interval with the plot process running is between 20 and 30 ms. Note that the current implementation of the Kernel gives a clock interrupt once every 10 ms. Therefore it only makes sense to use sampling intervals which are multiples of 10 ms.

2.2 CONTROL ALGORITHM

The control law implemented in DSF is a standard state feedback regulator. The state vector is estimated with an observer which uses the latest available measured process output. For futher details, see Chapters 9 and 15 in [Åström and Wittenmark, 1984]. The regulator equations are:

$$\begin{cases} \hat{x}(k|k) = \hat{x}(k|k-1) + K(y(k)-C\hat{x}(k|k-1)) \\ u(k) = l_r y_r(k) - L\hat{x}(k|k) \\ \hat{x}(k+1|k) = \Phi\hat{x}(k|k) + \Gamma u(k) \end{cases} \tag{2.1}$$

The notation $\hat{x}(k|k-1)$ indicates that the estimate at time k is based on data available at time k-1. The observer structure in (2.1) is sometimes called a "current observer" [Franklin and Powell, 1980], since it uses the latest (current) measurement value.

Windup protection

In order to avoid estimator windup it is important to feed the observer with the actual output control signal. Therefore the regulator is implemented as

$$\begin{cases} \hat{x}(k|k) = \hat{x}(k|k-1) + K(y(k)-C\hat{x}(k|k-1)) \\ v(k) = l_r y_r(k) - L\hat{x}(k|k) \\ \text{IF RegulatorOn THEN } u(k)=\text{Sat}(v(k),u_{max},u_{min}) \text{ ELSE } u(k)=0 \\ \hat{x}(k+1|k) = \Phi\hat{x}(k|k) + \Gamma u(k) \end{cases} \tag{2.2}$$

where

$$\text{Sat}(u,u_{max},u_{min}) = \begin{cases} u_{min} ; & u \le u_{min} \\ u ; & u_{min} \le u \le u_{max} \\ u_{max} ; & u \ge u_{max} \end{cases}$$

and u(k) is the control output that is sent to the D/A converter. The boolean variable RegulatorOn is used to turn the regulator on and off. Note that the observer equations are updated each sampling interval even when the regulator is off. This means that the estimated state values will be correct when the regulator is turned on again, and thus undesired transients are avoided. This way of disabling the regulator is better than the more primitive method of just stopping the regulator process in the program. The drawback is that the major part of the regulator computations have to be done also when the regulator is off.

Also note that the control output is set to zero explicitly when the regulator is off. If this is not done, the D/A converter will hold the last output control signal. It is of particular importance that the D/A output is reset to zero before stopping the program execution. Otherwise disastrous transients may occur if a new process is connected to the D/A output. In DSF, the D/A output is reset automatically before terminating the program (cf. the command EXIT in Section 2.3, Operator Communication).

Parameter changes

An important feature with the state feedback structure is that the states often have a physical interpretation like speed, position, pressure, temperature etc. This enables us to make (almost) bumpless parameter changes without any special precautions. A change of observer gains (K), process model (Φ, Γ) or sampling interval will not have any immediate effect on the estimated state values, and hence no transients appear in the control signal. The only parameters that have an immediate effect on the control output are the feedback gains (L). A change of feedback gains may therefore give a step in the control signal. If the parameter change is done at steady state operating conditions, this step is normally small.

2.3 OPERATOR COMMUNICATION

The operator communication has been kept as simple as possible to avoid an excessive amount of graphics code. It is based on a command menu on the screen. The operator selects the desired command by clicking a mouse button inside one of the menu rectangles. The regulator reference value is entered by clicking a mouse button at the appropriate level in the plot area. The reference value change is limited to avoid excessive reference step amplitudes. The current reference change limit is 1.

Command menu:

EXIT	Terminate program execution and return to MS-DOS. The regulator is first turned off. Then the program waits three sampling intervals, to ensure that the regulator has been turned off, before it terminates. Thus the D/A output will always be reset to zero before termination of the program.
REG	Regulator on/off. In the OFF state, indicated by a red REG rectangle, the D/A output is reset to zero. The ON state is indicated by a green REG rectangle.
PLOT	Screen plot on/off. On is indicated by a green PLOT rectangle and OFF by a red rectangle. The log buffer is only updated when the plot is ON.
PAR	Enter new regulator parameters from disk file. The program prompts for parameter file name and regulator order. The file name should have the structure 'filename.ext'. No default names or extensions are used. The current maximum regulator order is 3. After succesful reading of the new parameters, the user is prompted to confirm that they should be entered in the RegData monitor. If this is not done, the new parameters are discarded.
LOG	Write current content of the log buffer on a disk file. The program prompts for log file name (default = LOG.T).

Plotted signals:

Black	Reference signal
White	Measured process output (Analog Input 0)
Red	Auxiliary input signal (Analog Input 1)
Green	Control signal (Analog Output 0)

The number of plotted signals can easily be increased. This also increases the computational load, however.

2.4 PARAMETER FILES

The parameter file in DSF contains the sampling interval h [ms], the state space model matrices Φ, Γ, and C, the controller gain vectors L and K and the reference signal gain l_r. Parameter files can be generated with a PC-MatLab function called DSFP (see Appendix B). The DSF command PAR (cf. Section 2.3) is used to read a new parameter file. The format of the parameter file is compatible with the parameter files in SIMNON. Therefore the parameter files can also be used in SIMNON where they are read with the command GET. Parameter files can also be generated with Ctrl-C. Only minor modifications are necessary in the PC-Matlab function DSFP to convert it to a Ctrl-C function.

3. LOG - DATA LOGGING

The LOG program includes facilities for data logging and plotting of measured signals on the screen. Logged data can be written to a text file. The text file format is chosen so that it can be converted off line to a binary data file and plotted with IDPAC or SIMNON. A special feature is the possibility to change the A/D converter gain. By selecting a suitable gain the quantization error can be reduced for low level signals without the use of external amplifiers. The operator communication is very simple. It is based on a screen menu, where the operator selects the desired command by clicking a mouse button inside one of the menu rectangles.

3.1 SOFTWARE ARCHITECTURE

The program is divided into three process modules and three monitor modules. LOG also uses the graphics utility procedures in the GraphUti module, and a special version of the AnalogIO module which includes the programmable A/D converter gain facility. Each module is separately compiled.

Monitor modules:

LogPar	Log parameters and global data.
LogBuff	Log data buffer.
PlotBuff	Plot data buffer.

Process modules:

Scan	Scans the A/D inputs and sends data to LogBuff and PlotBuff.
Plot	Plots measured data on the screen.
OpCom	Operator communication.

LogPar

The LogPar module implements a monitor which protects global log parameters. The log parameters are stored in a record of type LogParType (see Appendix C). The parameters can be accessed with two monitor entry procedures, GetLogPar and PutLogPar, which provide mutual exclusion.

LogBuff

The LogBuff module implements a ring buffer monitor with 1000 elements. It is similar to the LogBuff module in the DSF program (cf. Section 2.1).

PlotBuff

The PlotBuff module implements a ring buffer monitor with 25 elements. It is similar to the PlotBuff module in the DSF program (cf. Section 2.1).

Scan

The Scan module contains a process which reads the A/D input channels. The sampling interval, A/D gain and a boolean variable 'LogOn' are read from the LogPar monitor. The measured data are sent to the plot process via the plot buffer PlotBuff. When LogOn is TRUE, the scan process also sends the measured data to the log buffer. The log function is therefore enabled and disabled from the scan process. This is different from the DSF program, where the plot process sends log data to the log buffer, and hence controls the updating of the log. In LOG, however, this mechanism cannot be used because the log and plot functions must be separated. This, in turn, depends on the fact that a 10 ms sampling interval can only be achieved with the screen plot disabled. Another reason is that when the log is turned on, we do not want the log buffer to be filled with old data from the plot buffer. That is what happens if the log data are sent to the log buffer from the plot process, and the log is stopped by freezing the plot. In DSF this can be accepted, however, since there the intended use of the log function is to save plotted data.

Plot

The Plot module contains the plot process. It reads plot data from PlotBuff and plots the data on the screen. The plot process is completely separated from the log buffer, and the screen plot can be frozen while the log is still active. The plot is frozen in the same way as in DSF by halting the plot process with the plot enable flag in PlotBuffer.

OpCom

The OpCom module is divided into two processes, OpComProcess with high priority and IOProcess with low priority. This module is similar to the OpCom module in DSF with some obvious changes in the command menu and the operator dialogue.

Priorities and sampling rate

The processes have the following priorities (high numbers indicate low priority):

Scan	10
OpComProcess	15
Plot	20
IOProcess	30

The priority considerations in LOG are the same as in DSF. Scan has the highest priority to guarantee that the A/D inputs are read at correct times. The time consuming IOProcess must have the lowest priority for the same reason as in the DSF program to avoid blocking the Scan process.

The minimum sampling interval is approximately 20 ms with the plot process running, and 10 ms with the plot process stopped.

3.2 A/D CONVERSION WITH PROGRAMMABLE GAIN

The Analog Devices RTI-800 and RTI-802 boards have A/D converters with programmable gain. This feature is not used in the ADIn procedure of the standard AnalogIO module, which has a fixed gain corresponding to a full scale input range of ±10 V. With a 12 bit converter, the corresponding quantization interval is 5 mV. This may be unacceptable when measuring low level signals. Therefore a modified version of the AnalogIO module has been written, where the ADIn procedure has an extended parameter list including the A/D converter gain. The new procedure heading is

```
    PROCEDURE ADIn(Channel:CARDINAL; Gain:INTEGER) : REAL;
```

See Appendix C for further details. As in the standard version, this procedure returns a value in the interval (-1 .. 1) from the channel number 'Channel'. The returned value is calculated as

```
  output = input*Gain/10
```

Allowed gain values are 1, 10 100 and 500. The maximum gain value corresponds to a full scale input range of ±20 mV. This should be sufficient for most applications. For example, it enables direct connection of thermocouples to the

A/D converter input. When working with such low signal levels, the wiring and grounding on the back plane must be made with great care to avoid disturbances. The A/D converter gain can be changed on line and has the default value Gain = 1 when the program is started.

3.3 OPERATOR COMMUNICATION

The operator communication is similar to DSF. The desired command is selected by clicking a mouse button inside one of the menu rectangles. When necessary the program prompts for further information from the keyboard.

Command menu:

EXIT	Terminate program execution and return to MS-DOS.
LOG	Log on/off. The OFF state is indicated by a red LOG rectangle, and the ON state is indicated by a green rectangle.
PLOT	Screen plot on/off. On is indicated by a green PLOT rectangle and OFF by a red rectangle.
PAR	Enter new log parameters. Current gain and sampling interval [h] values are displayed, and the program prompts for new values. Enter RETURN if the current value is ok.
SAVE	Write current content of the log buffer on a disk file. The program prompts for log file name (default = LOG.T). The file name should have the structure 'filename.ext'. After succesful writing of the log file, the user receives the message 'Done'. In other cases, an error message is displayed.

Plotted signals:

Blue	Analog Input 0
Red	Analog Input 1
Black	Analog Input 3

The number of plotted signals can be increased easily. When three signals are plotted, the minimum samplig interval for the LOG program is h = 20 ms. In order to log data with a sampling interval of 10 ms, the plot must be disabled.

4. REFERENCES

Elmqvist, H., Åström, K.J., and Schönthal, T.S. (1986): "SIMNON - User's Guide for MS-DOS Computers". Dept. of Automatic Control, Lund Institute of Technology.

Franklin, G.F. and Powell, J.D. (1980): "Digital Control of Dynamic Systems". Addison-Wesley, Reading, Mass., USA.

Logitech, Inc. (1985): "Modula-2/86 User's Manual. Release 2.00". Redwood City, CA, USA, December 1985.

MathWorks, Inc. (1986): "PC-MATLAB for MS-DOS Personal Computers. User's Guide". Sherborn, MA, USA.

Systems Control Technology, Inc. (1986): "Ctrlc-C User's Guide". Palo Alto, CA, USA.

Åström, K.J. and Wittenmark, B. (1984): "Computer Controlled Systems - Theory and design". Prentice-Hall, Englewood Cliffs, N.J., USA.

Wallenborg, A. (1987): "Control of Flexible Servo Systems". Licentiate thesis, Dept. of Automatic Control, Lund Institute of Technology. CODEN:LUTFD2/(TFRT-3188)/1-94/(1987).

Wirth, N. (1985): "Programming in MODULA-2". Third edition. Springer-Verlag, New York.

5. ACKNOWLEDGEMENTS

Two essential prerequisites for the DSF and LOG programs are the modules Kernel and Graphics, written by Leif Andersson. Kernel provides the necessary real time primitives, and it is a convenient tool for real time programming in Modula-2 on the IBM PC/AT. Many of the ideas about the software architecture have come from Kjell Gustafsson, and I have also had interesting and fruitful discussions with Ola Dahl and Per Hagander. Their contributions are gratefully acknowledged.

Appendix A

Excerpts from the DSF source code.

CONTENTS:

```
MODULE Dsf;
(*
    Main program for digital state feedback control.
    Controller parameters are read from a SIMNON parameter file.
    The screen plot is continuously logged in a log buffer.
    Log buffer contents can be saved on a disk file.

    Monitor modules:
    ---------------
    RegData      Regulator data and parameters
    PlotBuff     Plot data buffer
    LogBuff      Log data buffer

    Process modules:
    ---------------
    Regulate     Gets parameters from RegData, calculates a new
                 control signal and sends plot data to PlotBuffer.
    Plot         Gets plot data from PlotBuffer and plots them on the
                 screen. Puts plot data in LogBuff.
    OpCom        Operator communication. OpCom is divided into two
                 processes. One (with high priority) waits for a
                 mouse click in one of the menu rectangles and initiates
                 the selected action.
                 The second process has low priority and handles file
                 oriented I/O operations.
                 The reference value for the regulator is changed by
                 pointing at the corresponding level in the plot and
                 clicking the mouse button.

    Author: Anders Wallenborg
*)
IMPORT RTMouse, PlotBuff, LogBuff, RegData, Regulate, Plot, OpCom;

BEGIN
  RTMouse.Init;

  RegData.Init;              (* initialize monitors *)
  PlotBuff.Init;
  LogBuff.Init;

  Regulate.Start;            (* start processes *)
  Plot.Start;
  OpCom.Start;

  OpCom.WaitForExit;
END Dsf.
```

```
DEFINITION MODULE GraphUtility;

(*
   A collection of routines to make life easier for the graphics
   programmer.
*)

FROM Graphics IMPORT
  handle, point, color;

EXPORT QUALIFIED
  SetPoint, PlotLine, PlotStair, PromptAndRead;


PROCEDURE SetPoint(VAR p:point;  x,y:REAL);

PROCEDURE PlotLine(p1,p2:point;  h:handle;  linecolor:color);

PROCEDURE PlotStair(p1,p2:point;  h:handle;  linecolor:color);

PROCEDURE PromptAndRead(h:handle;  p:point;  promptstr:ARRAY OF CHAR;
                        VAR str:ARRAY OF CHAR);

END GraphUtility.



DEFINITION MODULE LogBuff;
(*
   This module implements a ring buffer containing items of type
   'LogDataType'.
*)
EXPORT QUALIFIED
  LogDataType, nlog, Init, PutLogBuffer, GetLogBuffer, WriteLogBuffer;

CONST
  nlog = 4;

TYPE
  LogDataType = RECORD
                  data : ARRAY[1..nlog] OF REAL;
                  time : CARDINAL;
                END;

PROCEDURE Init;

PROCEDURE PutLogBuffer(Item:LogDataType);
(*
   Put an item in the log buffer. If the buffer is full the oldest
   item will be overwritten.
*)

PROCEDURE GetLogBuffer(VAR Item:LogDataType);
(*
   Get an item from the log buffer. If the buffer is empty,
   'GetLogBuffer' will wait until data are available, i.e.
   until 'PutLogBuffer' has been executed.
*)

PROCEDURE WriteLogBuffer(filename: ARRAY OF CHAR; VAR error: CARDINAL);
(*
   Write log buffer contents on disk file.
   The string 'filename' should have the structure 'DK:name.ext'
   No file name check is done.
   error = 0 : Log written succesfully on file
           1 : Log buffer empty
           2 : File open error
           3 : Write error
           4 : File close error
*)

END LogBuff.



DEFINITION MODULE OpCom;
(*
   This module contains the operator communication.
   It is started by calling 'Start'.
   'WaitForExit' is a procedure which blocks the calling process
   until the operator selects a command which terminates program
   execution.
*)
EXPORT QUALIFIED Start, WaitForExit;

PROCEDURE Start;

PROCEDURE WaitForExit;

END OpCom.



DEFINITION MODULE Plot;
(*
   This module contains the plot process.
   It is started by calling 'Start'.
*)
EXPORT QUALIFIED Start;

PROCEDURE Start;

END Plot.



DEFINITION MODULE PlotBuff;
(*
   This module implements a ring buffer containing items of type
   'PlotDataType'.
*)
EXPORT QUALIFIED
```

```
  PlotDataType, Init, PutPlotBuffer, GetPlotBuffer,
  EnablePlot, DisablePlot, PlotEnabled;

TYPE
  PlotDataType = RECORD
                  yref, y1, y2, u : REAL;
                  time, h : CARDINAL;
                 END;

PROCEDURE Init;

PROCEDURE PutPlotBuffer(Item : PlotDataType);
(*
   Put an item in the buffer. If the buffer is full,
   then the oldest item in the buffer will be overwritten.
*)

PROCEDURE GetPlotBuffer(VAR Item : PlotDataType);
(*
   Get an item from the buffer. If the buffer is empty 'GetPlotBuffer'
   will wait until data are available, i.e. until 'PutPlotBuffer'
   has been executed.
*)

PROCEDURE EnablePlot;
(*
   Set the plot enable flag. If this flag is false, calls to
   'GetPlotBuffer' will wait until it becomes true.
*)

PROCEDURE DisablePlot;
(*
   Reset the plot enable flag.
*)

PROCEDURE PlotEnabled():BOOLEAN;
(*
   Returns the current value of the plot enable flag.
*)

END PlotBuff.



DEFINITION MODULE RegData;
(*
   This module defines a monitor for the regulator parameters of a
   state feedback control algorithm with observer.
   Author: Anders Wallenborg
*)

FROM FileSystem IMPORT File;

EXPORT QUALIFIED
  RegDataType, nmax, Init, PutRegData, GetRegData, ReadParFile;
```

```
CONST nmax = 3;     (* maximum system order *)

TYPE
  RegDataType = RECORD
                 RegulatorOn : BOOLEAN;
                 yref : REAL;
                 h : CARDINAL;        (* sampling interval *)
                 n : CARDINAL;        (* system order *)
                 F : ARRAY [1..nmax],[1..nmax] OF REAL;
                 G : ARRAY [1..nmax] OF REAL;
                 C : ARRAY [1..nmax] OF REAL;
                 K : ARRAY [1..nmax] OF REAL;
                 L : ARRAY [1..nmax] OF REAL;
                 lr : REAL;
                END;

PROCEDURE Init;

PROCEDURE PutRegData(Item: RegDataType);
(*
   Put regulator data in monitor
*)
PROCEDURE GetRegData(VAR Item: RegDataType);
(*
   Get regulator data from monitor
*)
PROCEDURE ReadParFile(filename: ARRAY OF CHAR; order: CARDINAL;
                      VAR Item: RegDataType; VAR error: CARDINAL);
(*
   Read regulator parameters from SIMNON parameter file.
   The string 'filename' should have the structure 'DK: name.ext'
   No file name check is done.
   order = system (regulator) order
   error = 0 : Parameter file read succesfully
           1 : File open error
           2 : Data conversion error
           3 : File close error
*)

END RegData.



DEFINITION MODULE Regulate;
(*
   This module contains the regulator process.
   It is started by calling 'Start'.
*)
EXPORT QUALIFIED Start;

PROCEDURE Start;

END Regulate.
```

```
IMPLEMENTATION MODULE LogBuff;

FROM ConvReal IMPORT
  RealToString;
FROM FileSystem IMPORT
  File, Response, Lookup, Delete, Close, WriteChar;
FROM Kernel IMPORT
  Semaphore, Event, Signal, Wait, Cause, Await, InitSem, InitEvent;
FROM NumberConversion IMPORT
  CardToString;

CONST
  BufferLength = 500;

TYPE
  Index = [0..BufferLength-1];

VAR
  mutex : Semaphore;
  nonempty : Event;
  buffer : ARRAY Index OF LogDataType;      (* buffer data vector *)
  writepos : Index;                         (* next write position *)
  readpos : Index;                          (* next read position *)
  count : [0..BufferLength];
  log : ARRAY Index OF LogDataType;         (* output data vector *)


(* entry *) PROCEDURE Init;
BEGIN
  InitSem(mutex,1);
  InitEvent(nonempty,mutex);
  writepos := 0;
  readpos := 0;
  count := 0;
END Init;


(* entry *) PROCEDURE PutLogBuffer(Item: LogDataType);
BEGIN
  Wait(mutex);
  buffer[writepos] := Item;
  writepos := (writepos+1) MOD BufferLength;
  IF count=BufferLength THEN
    readpos := (readpos+1) MOD BufferLength;   (* data lost *)
  ELSE
    count := count + 1;
  END;
  Cause(nonempty);
  Signal(mutex);
END PutLogBuffer;


(* entry *) PROCEDURE GetLogBuffer(VAR Item: LogDataType);
BEGIN
  Wait(mutex);
  WHILE count=0 DO  Await(nonempty);  END;
  Item := buffer[readpos];
  readpos := (readpos+1) MOD BufferLength;
  count := count - 1;
  Signal(mutex);
END GetLogBuffer;


PROCEDURE WriteString(s:ARRAY OF CHAR; VAR f:File; VAR ok:BOOLEAN);
CONST
  nul = 00C;
VAR
  i : CARDINAL;
BEGIN
  i := 0;
  ok := TRUE;
  LOOP
    IF (s[i]=nul) OR (i>HIGH(s)) THEN EXIT; END;
    WriteChar(f,s[i]);
    IF f.res # done THEN ok := FALSE; EXIT; END;
    i := i+1;
  END;
END WriteString;


(* entry *) PROCEDURE WriteLogBuffer(filename:ARRAY OF CHAR;
                                    VAR error:CARDINAL);
CONST
  eol = 36C;
  nul = 00C;
  cwidth = 6;        (* cardinal string width *)
  rwidth = 7;        (* real string width *)
VAR
  logfile : File;
  i, rp : Index;               (* rp = local read position variable *)
  n : [0..BufferLength];
  k : [1..nlog];
  str : ARRAY [0..79] OF CHAR;
  ok : BOOLEAN;
BEGIN
  error := 0;
  i := 0;
  Wait(mutex);
  IF count = 0 THEN
    error := 1;     (* logbuffer empty *)
  ELSE   (* copy buffer to log vector *)
    n := count;
    rp := readpos;
    FOR i:=0 TO n-1 DO
      log[i] := buffer[rp];
      rp := (rp+1) MOD BufferLength;
    END;
  END;
  Signal(mutex);
  Delete(filename,logfile);
  Lookup(logfile,filename,TRUE);
  IF (logfile.res # done) AND (error=0) THEN error := 2; END;
  i:=0;
  LOOP
    IF (error # 0) THEN  EXIT;  END;
    CardToString(log[i].time,str,cwidth);
    WriteString(str,logfile,ok);
    IF NOT ok THEN error := 3;  EXIT;  END;
    WriteChar(logfile,' ');
    IF logfile.res # done THEN error := 3; EXIT; END;
    FOR k := 1 TO nlog DO
      RealToString(log[i].data[k],str,rwidth);
      WriteString(str,logfile,ok);
      IF NOT ok THEN error := 3; EXIT; END;
      WriteChar(logfile,' ');
      IF logfile.res # done THEN error := 3; EXIT; END;
    END;
    WriteChar(logfile,eol);
    IF logfile.res # done THEN error := 3; EXIT; END;
    IF i >= n-1 THEN EXIT; END;
    i := i+1;
  END; (* loop *)
  Close(logfile);
  IF (logfile.res # done) AND (error=0) THEN
    error := 4;
  END;
END WriteLogBuffer;

END LogBuff.



IMPLEMENTATION MODULE Regulate;
(*
   This module contains the regulator process 'Process', which
   implements a digital state feedback controller and an observer
   which uses the latest measured value in the state estimates.
   The regulator process is started by calling 'Start'.
   Regulator data are stored in the monitor 'Regdata'.
   The regulator process sends plot data to the plot buffer 'PlotBuffer'.
   Author: Anders Wallenborg
*)
FROM AnalogIO IMPORT
  ADIn, DAOut;
FROM Kernel IMPORT
  Time, GetTime, IncTime, WaitUntil,
  SetPriority, CreateProcess;
FROM PlotBuff IMPORT
  PlotDataType, PutPlotBuffer;
FROM RegData IMPORT
  RegDataType, nmax, GetRegData, PutRegData;

CONST
  regpriority = 10;
  umax = 1.0;
  umin = -1.0;
```

```
(* Process *) PROCEDURE Process;

VAR
  plotdata : PlotDataType;
  regdata : RegDataType;
  y, y2, u : REAL;
  tsamp : Time;
  x : ARRAY [1..nmax] OF REAL;        (*  xest[k|k]    *)
  xe : ARRAY [1..nmax] OF REAL;       (*  xest[k|k-1]  *)
  ye : REAL;                          (*  yest[k|k-1]  *)
  i,j : CARDINAL;
BEGIN
  SetPriority(regpriority);
  ye := 0.0;
  GetRegData(regdata);
  FOR i:=1 TO regdata.n DO xe[i] := 0.0; END;
  GetTime(tsamp);
  LOOP
    GetRegData(regdata);
    WITH regdata DO
      y := ADIn(0);
      y2 := ADIn(1);
      FOR i:=1 TO n DO                      (*  x = xe + K*(y-ye)  *)
        x[i] := xe[i] +.K[i]*(y-ye);
      END;
      IF RegulatorOn THEN
        u := lr*yref;
        FOR i:=1 TO n DO                    (*  u = lr*yr - L*x  *)
          u := u - L[i]*x[i];
        END;
        IF u > umax THEN
          u := umax;
        ELSIF u < umin THEN
          u := umin;
        END;
      ELSE
        u := 0.0;
      END;
      DAOut(0,u);
      FOR i:=1 TO n DO                      (* xe[k+1] = Fx[k] + Gu[k] *)
        xe[i] := G[i]*u;
        FOR j:=1 TO n DO
          xe[i] := xe[i] + F[i,j]*x[j];
        END;
      END;
      ye := 0.0;
      FOR i:=1 TO n DO                      (* ye[k+1] = Cxe[k+1] *)
        ye := ye + C[i]*xe[i];
      END;
    END; (* with *)
    plotdata.yref := regdata.yref;
    plotdata.y1 := y;
    plotdata.y2 := y2;
    plotdata.u := u;
    plotdata.time := tsamp.lo;
    plotdata.h := regdata.h;
    PutPlotBuffer(plotdata);
    IncTime(tsamp,regdata.h);
    WaitUntil(tsamp);
  END; (* loop *)
END Process;


PROCEDURE Start;
BEGIN
  CreateProcess(Process,10000);
END Start;

END Regulate.
```

Appendix B

This appendix contains a PC-MATLAB procedure called DSFP which can be used to generate parameter files for DSF and SIMNON. A sample parameter file is also included.

```
function [] = dsfp(F,G,C,K,L,lr,h,syst);
%
% function [] = dsfp(F,G,C,K,L,lr,syst);
%
% Conversion of discrete time state feedback controller parameters to a
% SIMNON parameter file with the system name 'syst'.
%
%   F = Fi matrix
%   G = Gamma matrix
%   C = C matrix
%   K = observer gain vector
%   L = state feedback gain vector
%   lr= reference signal gain
%   h = sampling interval [sec]
%   syst = SIMNON system name (should be entered within single quotes)
%
%   Output file name: DSFPAR.T
%   Maximum system order = 5
%
% Author: Anders Wallenborg
%
[m,n] = size(F);
%
syst=['[',syst,']'];
Fstr=['F11: '; 'F12: '; 'F13: '; 'F14: '; 'F15: ';
      'F21: '; 'F22: '; 'F23: '; 'F24: '; 'F25: ';
      'F31: '; 'F32: '; 'F33: '; 'F34: '; 'F35: ';
      'F41: '; 'F42: '; 'F43: '; 'F44: '; 'F45: ';
      'F51: '; 'F52: '; 'F53: '; 'F54: '; 'F55: '];
Fstr = [Fstr(1:n,:);
        Fstr(6:5+n,:);
        Fstr(11:10+n,:);
        Fstr(16:15+n,:);
        Fstr(21:20+n,:)];
Gstr = ['G1:  '; 'G2:  '; 'G3:  '; 'G4:  '; 'G5:  '];
Cstr = ['C1:  '; 'C2:  '; 'C3:  '; 'C4:  '; 'C5:  '];
Kstr = ['K1:  '; 'K2:  '; 'K3:  '; 'K4:  '; 'K5:  '];
Lstr = ['L1:  '; 'L2:  '; 'L3:  '; 'L4:  '; 'L5:  '];
lrstr = 'lr:  ';
hstr = 'h: ';
%
for i=1:n
   s = sprintf('%g',G(i));
   gpar(i,1:length(s)) = s;
   s = sprintf('%g',C(i));
   cpar(i,1:length(s)) = s;
   s = sprintf('%g',K(i));
   kpar(i,1:length(s)) = s;
   s = sprintf('%g',L(i));
   lpar(i,1:length(s)) = s;
   for j=1:n
      s = sprintf('%g',F(i,j));
      fpar((i-1)*n+j,1:length(s)) = s;
   end
end
lrpar = sprintf('%g',lr);
hpar = num2str(h);
%
diary dsfpar.t ;
disp(syst);
disp([hstr,hpar]);
disp([Fstr(1:n*n,:), fpar]);
disp([Gstr(1:n,:), gpar]);
disp([Cstr(1:n,:), cpar]);
disp([Kstr(1:n,:), kpar]);
disp([Lstr(1:n,:), lpar]);
disp([lrstr, lrpar]);
diary off;
```

Sample parameter file generated with DSFP:

```
[reg]
h: 0.0400
F11: 0.897191
F12: 0.085234
F13: 4.181328
F21: 0.012501
F22: 0.984771
F23: -0.618110
F31: -0.038329
F32: 0.038632
F33: 0.902112
G1:  43.746512
G2:  0.191013
G3:  -0.888648
C1:  0.100000
C2:  0.000000
C3:  0.000000
K1:  8.186183
K2:  8.154908
K3:  1.055569
L1:  0.016797
L2:  0.037445
L3:  -0.086916
lr:  0.554039
```

Appendix C

Excerpts from the LOG source code. Only modules which are significantly different from the DSF program have been included.

CONTENTS:

```
MODULE Log;
(*
    Main log program.

    Monitor modules:
    LogPar      Log parameters and global data
    LogBuff     Log data buffer
    PlotBuff    Plot data buffer

    Process modules:
    Scan        Scans AD inputs. Sends data to LogBuff and PlotBuff.
    Plot        Gets plot data from PlotBuff and plots them on the
                screen.
    OpCom       Operator communication. OpCom is divided into two
                processes. One (with high priority) waits for a
                mouse click in one of the menu rectangles and initiates
                the selected action.
                The second process has low priority and handles file and
                keyboard oriented I/O operations.

    Author: Anders Wallenborg
*)
IMPORT RTMouse, PlotBuff, LogBuff, LogPar, Scan, Plot, OpCom;

BEGIN
  RTMouse.Init;

  LogPar.Init;                (* initialize monitors *)
  PlotBuff.Init;
  LogBuff.Init;

  Scan.Start;                (* start processes *)
  Plot.Start;
  OpCom.Start;

  OpCom.WaitForExit;
END Log.
```

```
DEFINITION MODULE AnalogIO;
(*
  Analog input/output. This version of AnalogIO is for the Analog Devices
  RTI-800 and RTI-802 boards. User programmable A/D converter gain included.
*)
EXPORT QUALIFIED ADIn, DAOut;

PROCEDURE ADIn(Channel : CARDINAL; Gain : INTEGER) : REAL;
(*
  Returns a value in the interval [-1.0..1.0], corresponding to input*Gain/10,
  from channel number Channel. Allowed channel numbers are 0 through 3.
  Allowed gain values are 1, 10, 100 and 500.
*)
PROCEDURE DAOut(Channel : CARDINAL; Value : REAL);
(*
  Outputs a value in the interval [-1.0..1.0], corresponding to [-10.0 V..10.0 V],
  to channel number Channel. Allowed channel numbers are 0 and 1.
*)
END AnalogIO.



DEFINITION MODULE LogPar;
(*
   This module defines a monitor for the logger parameters.
   Author: Anders Wallenborg
*)

EXPORT QUALIFIED
  LogParType, Init, PutLogPar, GetLogPar;

TYPE
  LogParType = RECORD
                  LogOn : BOOLEAN;
                  h : CARDINAL;         (* sampling interval *)
                  gain : INTEGER;       (* AD converter gain *)
               END;

PROCEDURE Init;

PROCEDURE PutLogPar(par: LogParType);
(*
   Put log parameters in monitor
*)
PROCEDURE GetLogPar(VAR par: LogParType);
(*
   Get log parameters from monitor
*)

END LogPar.
```

```
DEFINITION MODULE Scan;
(*
   This module contains the AD input scan process.
   The process is started by calling 'Start'.
*)

EXPORT QUALIFIED Start;

PROCEDURE Start;

END Scan.



IMPLEMENTATION MODULE AnalogIO;
(*
   This version of AnalogIO is for the Analog Devices RTI-800
   and RTI 802 boards.
   Author (original version): Leif Andersson

   Revised 1987-04-15 by Anders Wallenborg:
   The ADIN procedure parameter list has been extended and includes the
   parameter 'Gain' to enable user selected A/D converter gain.
*)
FROM SYSTEM IMPORT INBYTE, OUTBYTE;

FROM MathLib IMPORT round, float;

(*$R-*)(*$S-*)(*$T-*)

PROCEDURE ADIn(Channel:CARDINAL; Gain:INTEGER) : REAL;
(*
  Returns a value in the interval [-1.0..1.0] from channel number Channel.
  Allowed channel numbers are 0 through 15.
  The returned value is input*Gain/10.
  Allowed gain values are 1, 10, 100 and 500.
*)
CONST
  ADmax = 15;              (* Maximum channel number *)
  ADBASE = 0300H;
  ADSTATUS = ADBASE;       (* Analog in status register *)
  ADCHAN = ADBASE + 1;     (* Channel/gain register *)
  ADCONV = ADBASE + 2;     (* Convert command *)
  ADINLOW = ADBASE + 3;    (* Converted value, low byte *)
  ADINHIGH = ADBASE + 4;   (* Converted value, high byte *)

VAR
  DLO, DHI,             (* Low and high bytes of input value *)
  i, j,                 (* Dummy variables for busy wait loop *)
  status: CARDINAL;     (* Used for checking that A/D conversion is complete *)
  RealInValue : REAL;   (* Value returned *)
  InValue: INTEGER;     (* Input value converted to 2 bytes *)

BEGIN
  IF Channel > ADmax THEN ERROR END;
  (* Set appropriate gain bits in channel/gain register *)
  IF Gain = 1 THEN
     Channel := Channel + 0;
  ELSIF Gain = 10 THEN
     Channel := Channel + 32;
  ELSIF Gain = 100 THEN
     Channel := Channel + 64;
  ELSIF Gain = 500 THEN
     Channel := Channel + 96;
  ELSE
     ERROR
  END;
  OUTBYTE(ADCHAN,Channel);

  (* Start conversion and wait for Busy State bit to equal 0 *)
  OUTBYTE(ADCONV,0);
  status := 0;
  REPEAT INBYTE(ADSTATUS,status) UNTIL (6 IN BITSET(status));

  (* Read data *)
  INBYTE(ADINLOW,DLO); INBYTE(ADINHIGH,DHI);
  InValue := INTEGER(DHI*256 + DLO);
  RealInValue:=float(InValue)/2048.0;
  RETURN RealInValue;

END ADIn;

PROCEDURE DAOut(Channel : CARDINAL; Value : REAL);
(*
   Outputs a value in the interval [-1.0..1.0] to channel number Channel.
   Allowed channel numbers are 0 and 1.
*)
CONST
  DACSELECT = 0310H; (* DAC select register *)
  DACLOW = DACSELECT + 1;  (* Output value, low byte *)
  DACHIGH = DACSELECT + 2; (* Output value, high byte *)
  DAmax = 7; (* Maximum output channel number *)

VAR
  DLO, DHI,            (* Low and high bytes of output value *)
  OutValue : CARDINAL; (* Output value converted to 2 bytes *)
BEGIN
  (* Set the channel *)
  IF Channel > DAmax THEN ERROR END;
  OUTBYTE(DACSELECT,Channel);
  (* Convert Value and output the result *)

  IF Value >  1.0 THEN Value :=  1.0 END;
  IF Value < -1.0 THEN Value := -1.0 END;
  OutValue := round(2047.0 * Value);
  DLO := OutValue MOD 256;
  DHI := OutValue DIV 256;
  OUTBYTE(DACLOW,DLO); OUTBYTE(DACHIGH,DHI);
END DAOut;

PROCEDURE ERROR;
```

```
(*
   This error procedure is used to terminate program execution.
   It forces the program to crash by using an array index outside
   the index range.
   The program crash gives the error message 'Range Error ...'
*)
VAR v     : ARRAY[1..2] OF INTEGER;
    dummy : INTEGER;

BEGIN
  dummy := -1;
  v[dummy] := 1;
END ERROR;

END AnalogIO.



IMPLEMENTATION MODULE Scan;
(*
   This module contains a process 'Process', which scans the analog
   input channels. The process is started by calling 'Start'.
   Global data are stored in the monitor 'LogPar'.
   The process sends nlog data to the log buffer 'LogBuff'.
   The process sends nplot data to the plot process via the plot buffer
   'PlotBuff'.
   Note: It is assumed that nlog >= nplot.

   Author: Anders Wallenborg
*)

FROM AnalogIO IMPORT
  ADIn;
FROM Kernel IMPORT
  Time, GetTime, IncTime, WaitUntil,
  SetPriority, CreateProcess;
FROM LogBuff IMPORT
  LogDataType, nlog, PutLogBuffer;
FROM LogPar IMPORT
  LogParType, GetLogPar;
FROM PlotBuff IMPORT
  PlotDataType, nplot, PutPlotBuffer;

CONST
  regpriority = 10;

(* Process *) PROCEDURE Process;

VAR
  y : ARRAY [1..nlog] OF REAL;
  logpar : LogParType;
  logdata : LogDataType;
  plotdata : PlotDataType;
  tsamp : Time;
  k : CARDINAL;
BEGIN
```

```
  SetPriority(regpriority);
  GetTime(tsamp);
  FOR k := 1 TO nlog DO  y[k] := 0.0; END;
  LOOP
    GetLogPar(logpar);
    FOR k := 1 TO nlog DO
      y[k] := ADIn(k-1,logpar.gain);
    END;
    IF logpar.LogOn THEN
      FOR k := 1 TO nlog DO
        logdata.data[k] := y[k];
      END;
      logdata.time := tsamp.lo;
      PutLogBuffer(logdata);
    END;
    FOR k := 1 TO nplot DO
      plotdata.y[k] := y[k];
    END;
    plotdata.time := tsamp.lo;
    plotdata.h := logpar.h;
    PutPlotBuffer(plotdata);
    IncTime(tsamp,logpar.h);
    WaitUntil(tsamp);
  END; (* loop *)
END Process;


PROCEDURE Start;
BEGIN
  CreateProcess(Process,1000);
END Start;

END Scan.
```