Computer Implementation of Control Systems

Nielsen, Lars

1991

CODEN: LUTFD2/(TFRT-7476)/1–135/(1991)

Computer Implementation of Control Systems

Lars Nielsen

Department of Automatic Control
Lund Institute of Technology
May 1991

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

Document name
Report

Date of issue
May 1991

Document Number
CODEN: LUTFD2/(TFRT-7476)/1-135/(1991)

Author(s)
Lars Nielsen

Supervisor

Sponsoring organisation

Title and subtitle
Computer Implementation of Control Systems

Abstract

The first verison of the text for the course Computer Implementation of Control Systems.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
135

Security classification

Recipient's notes

Preface

This text was written during the spring 1991 as course material in the course Computer Implementation of Control Systems. It builds on a tradition within the Department of Automatic Control, and several persons have contributed directly or indirectly. They are all acknowledged and the bibliography in Chapter 11 tries to give a description of this background. The text material were in some cases originally directed to an audience with a different background, and could be made more tailored to the present course. Some of the new material is, due to the time constraints in the writing process, very brief and should be accompanied with a lecture to be filled in. The text would thus benefit from a second iteration.

The present course requires a background in elementary real-time programming. It was also taught in connection with robotics material, this year based on Craig's book. To make the present text self-contained it would be necessary to include a chapter on real-time programming treating the basics like processes, semaphores, and so on. It would also be necessary to include more material on robotics, and perhaps another application area like maybe batch control, treating the basics of sequences, trajectories, limitations from kinematics and dynamics, and ways of programming. This could be done by using examples, e.g. by expanding Chapter 5, since it is not necessary to go into detail to be able to discuss how certain realities influence the software solutions.

1

Introduction

GOAL: To outline the background and aim of the book.

The implementor of embedded control systems may have to face certain consequences of his mistakes. In 1981, a robotic manipulator suddenly, due to a software error, started to move with high speed to the boundary of its workspace, and a nearby worker was crushed to death. In 1989, a prototype JAS airplane crashed due to a mistake in the control system (an improper anti-windup scheme). The monthly newsletter *Software Engineering Notes* contains several more examples where the public or the environment have been at risk. There is a desire to build even more complex systems, but at the same time to minimize risks. Because of this desire and examples as those above, computer implementation of control systems is now evolving as a new distinct engineering discipline. This computer engineering discipline is of substantial industrial importance, and as a branch of computer science it deserves the same attention as those branches that deals with the computer itself, its architecture or its general purpose use.

The aim of this book is to give an overview of the relevant topics in computer implementation of control systems, and also to describe some of the research topics. The focus is to describe the engineering discipline, rather than trying to find a theoretic basis. As an engineering discipline it is not on a firm basis yet. Instead the relevant knowledge is taken from different areas of engineering, and reflecting this fact, the presentation in this book is structured in three parts. Part I, that includes Chapters 2-5, is oriented towards the problems connected with control engineering, and attempts to describe some of the needed and desired functionalities in present and future systems. Chapter 2 illustrates some of the algorithmic considerations that is necessary to consider when transferring a text book algorithm to computer code. Chapter 3 gives a first illustration on a possible implementation. The discussion is held on the level of abstraction which deals with parallel processes. Sequencing control is treated in Chapter 4. Some different methods are presented, with emphasis on GRAFCET which currently seems to be the fastest growing method. Chapter 5 treats the handling of set-points, trajectories and paths. The treatment shows some of the available ideas to deal with constraints given by the controlled object, and briefly discusses the consequences for the implementation. Part II, that includes Chapters 6-8, is more implementation oriented. Structuring issues are of great importance when dealing with large and complex systems. The software in embedded systems is typically structured in layers, where the different layers may reflect the solutions to the problems considered in Chapters 2-5, or may reflect the tools required to create a man-machine interface with sufficient capabilities to handle the system. The handling of time is basic, and Chapter 6

presents an example of a real-time kernel. Chapter 7 shows how a number of support modules can be implemented on top of the real-time kernel. A step further is taken in Chapter 8 where a special language is used to define a control system. Part III gives examples of actual designs of implemented control systems by presenting the resulting interface to the operator, some of the underlying structures, and other design considerations. This part also includes methods that are current topics of research.

2

PID Control

K. J. Åström

GOAL: To present engineering issues when implementing a control algorithm.

The major goal of this chapter is to give a feeling for the algorithmic considerations when implementing a control algorithm. The PID controller is a common and simple controller, but it is complicated enough to require general considerations on wind-up and bumpless transfer. Other aspects of the PID controller are also discussed. It is necessary to be aware of these aspects, e.g. tuning procedures, when implementing the tools for the man-machine interface.

2.1 Background

Many control problems can be solved using a PID-controller. This controller is named after its function which can be described as

$$u(t) = K_c \left[e(t) + \frac{1}{T_i} \int^t e(s)\, ds + T_d \frac{de(t)}{dt} \right] = P + I + D \tag{2.1}$$

where u is the controller output, and e is the error, i.e. the difference between command signals u_c (the set point) and process output y (the measured variable). The control action is thus composed of three terms, one part (P) is proportional to the error, another (I) is proportional to the integral of the error, and a third (D) is proportional to the derivative of the error. Special cases are obtained by only using some of the terms i.e. P, I, PI, or PD controllers. The PI controller is most common. It is also possible to have more complicated controllers e.g. an additional derivative term, which gives a PIDD or a DPID controller. The name PID controller is often used as a generic name for all these controllers.

The PID controller is very common. It is used to solve many control problems. The controller can be implemented in many different ways. For control of large industrial processes it is very common to have control rooms filled with several hundred PID controllers. The algorithm can also be programmed into a computer system that can control many

loops. This is the standard approach to control of large industrial processes. Many special purpose control systems also use PID control as the basic algorithm.

The PID controller was originally implemented using analog techniques. The technology has developed through many different stages, pneumatic, relay and motors, transistors and integrated circuits. In this development much know-how was accumulated that was embedded into the analog design. In this process several useful modifications to the "textbook" algorithm given by Equation (2.1) were made. Many of these modifications were not published, but kept as proprietary techniques by the manufacturers.

Today virtually all PID regulators are implemented digitally. Early implementations of digital PID controllers were often a pure translation of the "textbook" algorithm which left out many of the good extra features of the analog design. The failures renewed the interest in PID control.

It is essential for any user of control systems to master PID control, to understand how it works and to have the ability to use, implement and tune PID controllers. This chapter provides this knowledge. In Section 2.2 we will discuss the basic algorithm and several modifications of the linear, small-signal behavior of the algorithm. These modifications give significant improvements of the performance. Modifications of the nonlinear, large-signal behavior of the basic algorithm are discussed in Section 2.3. This includes integrator windup and mode changes. Section 2.4 deals with regulator tuning. This covers simple empirical rules for tuning as well as tuning based on mathematical models. In Section 2.5 we discuss implementation of PID controllers using analog and digital techniques and Section 2.6 gives a summary.

2.2 The Control Algorithm

In this section the PID algorithm will be discussed in more detail. Properties of proportional, integral, and derivative action will be explained. Based on the insight gained we will also make several modifications of the small-signal behavior of the algorithm. This will lead to an improved algorithm which is significantly better than the basic algorithm given by Equation (2.1).

Proportional Action

A proportional controller can be described by

$$u = K_c(u_c - y) + u_b = K_c e + u_b \tag{2.2}$$

The control signal is thus proportional to the error. Notice that there is also a reset or a bias term u_b. The purpose of this term is to provide an adjustment so that the desired steady state value can be obtained. Without the reset term it is necessary to have an error to generate a control signal that it different from zero. The reset term can be adjusted manually to give the correct control signal and zero error at a desired operating point.

Equation (2.2) holds for a limited range only because the output of a controller is always limited. If we take this into account the input output relation of a proportional controller can be represented as in Figure 2.1. The range of input signals where the controller is linear is called the *proportional band.* Let p_B denote the proportional band and u_{min} and u_{max} the limits of the control variable. The following relation then holds

$$K_c = \frac{u_{max} - u_{min}}{p_B} \tag{2.3}$$

The proportional band is given in terms of the units of the measured value or in relative units. It is in practice often used instead of the controller gain. A proportional band of 50% thus implies that the controller has a gain of 2.

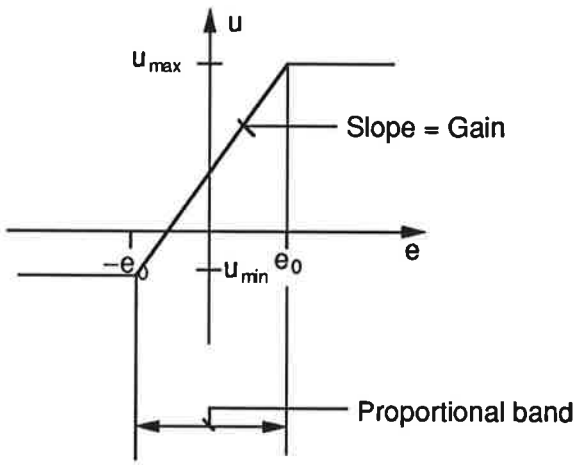

Figure 2.1 Input-output relation of a proportional controller.

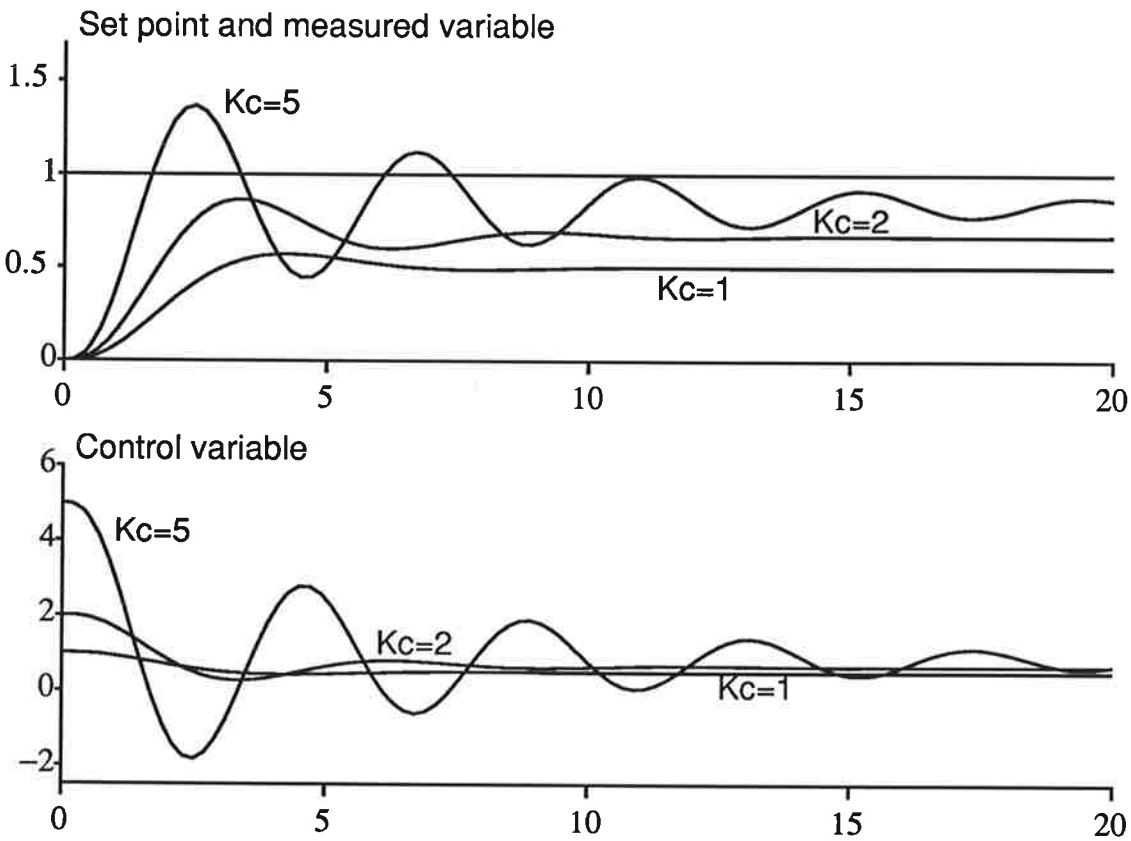

Figure 2.2 Illustration of proportional control. The process has the transfer function $G_p(s) = (s+1)^{-3}$.

The properties of a proportional controller are illustrated in Figure 2.2. This figure shows the step response of the closed loop system with proportional control. The figure shows clearly that there is a steady state error. The error decreases when the controller gain is increased, but the system then becomes oscillatory. It is easy to calculate the steady state error. The Laplace transforms of the error $e = u_c - y$ is given by

$$E(s) = \frac{1}{1 + G_p(s)G_c(s)} U_c(s)$$

where U_c is the Laplace transforms of the command signal. With $G_p(0) = 1$ and $G_c(0) = K_c = 1$ we find that the error due to a command signal is 50% as is seen in the figure.

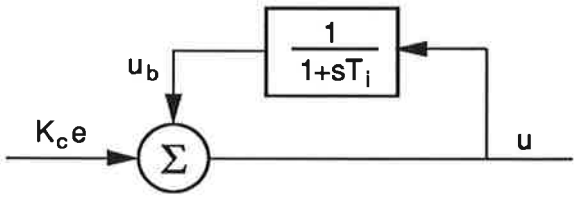

Figure 2.3 Controller with automatic reset.

Integral Action

Early controllers for process control had proportional action only. The reset adjustment u_b in (2.2) was used to ensure that the desired steady state value was obtained. Since it was tedious to adjust the reset manually there was a strong incentive to automate the reset. One way to do this is illustrated in Figure 2.3. The idea is to low pass filter the controller output to find the bias and add this signal to the controller output. It is straight forward to analyze the system in Figure 2.3. We get

$$U(s) = K_c E(s) + \frac{1}{1 + sT_i} U(s)$$

Solving for U we get

$$U(s) = K_c(1 + \frac{1}{sT_i})E(s)$$

Conversion to the time domain gives

$$u(t) = K_c \left[e(t) + \frac{1}{T_i} \int^t e(s)\, ds \right] = P + I \tag{2.4}$$

which is the input-output relation for a PI controller. Parameter T_i, which has dimension time, is called *integral time* or *reset time*. The properties of a PI controller are illustrated in Figure 2.4. The figure illustrates that the idea of automatic reset or PI control works very well in the specific case.

It is straight forward to show that a controller with integral action will always give a correct steady state. To do this assume that there exist a steady state. The process input, the output, and the error then are then constant. Let e_0 denote the error and u_0 the process input. It follows from the control law Equation (2.4) that

$$u_0 = K_c(e_0 + t\frac{e_0}{T_i})$$

This contradicts the assumption that u_0 is a constant unless e_0 is zero. The argument will obviously hold for any controller with integral action. Notice, however, that a stationary solution may not necessarily exist.

Another intuitive argument that also gives insight into the benefits of integral control is to observe that with integral action a small control error that has the same sign over a long time period may generate a large control signal.

Sometimes a controller of the form

$$u(t) = K_i \int^t e(s)\, ds = I \tag{2.5}$$

is used. This is called an I controller or a *floating controller*. The name floating relates to the fact that with integral control there is not a direct correspondence between the error and the control signal.

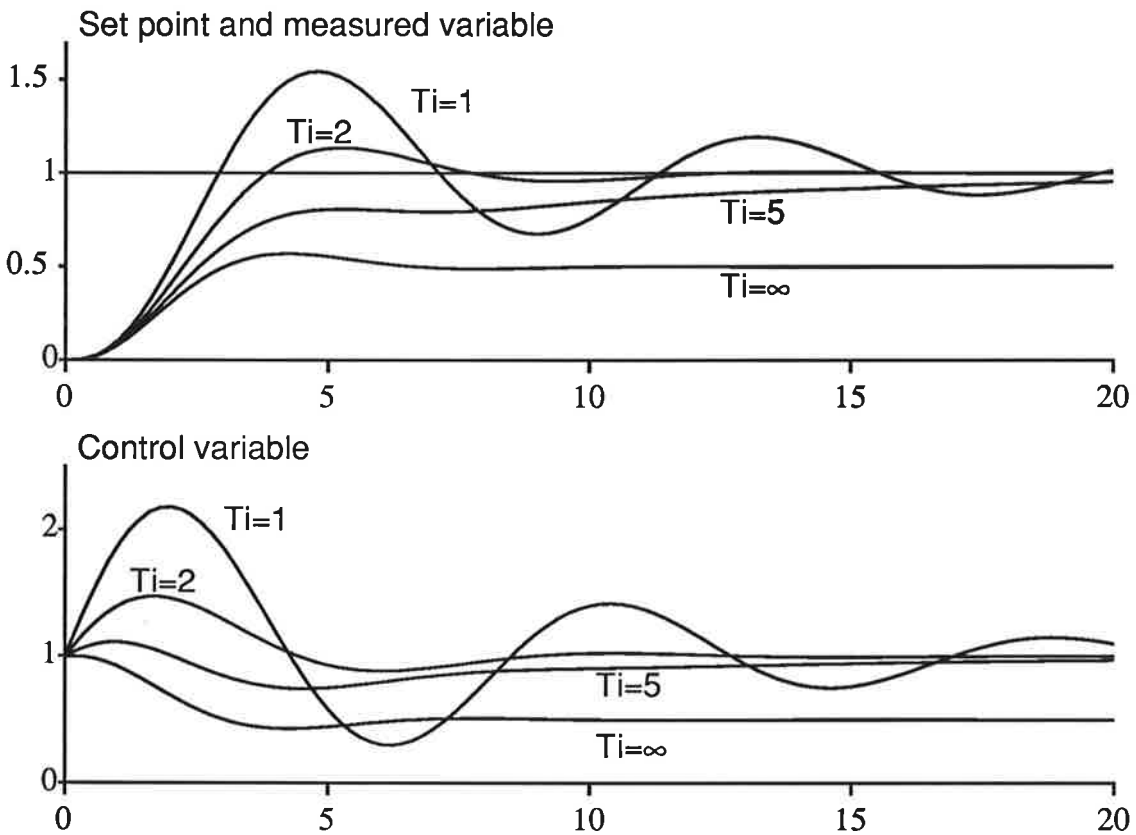

Figure 2.4 Illustration of PI control. The process has the transfer function $G_p(s) = (s+1)^{-3}$. The controller has gain $K_c = 1$.

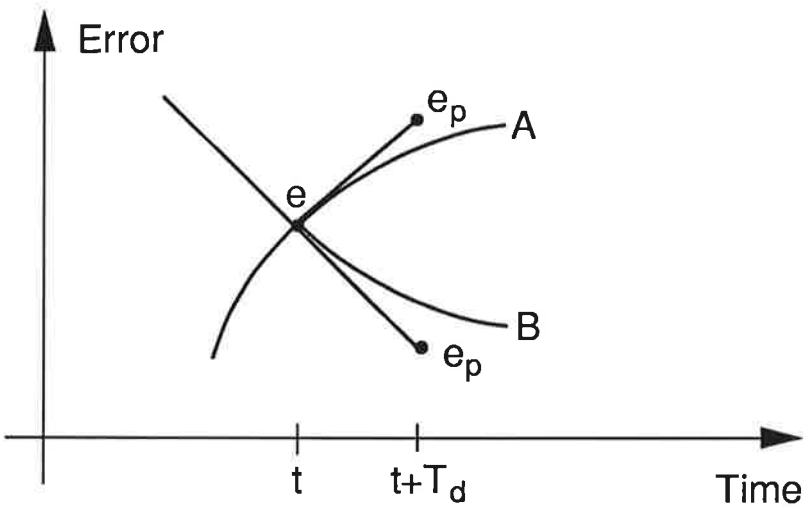

Figure 2.5 Illustrates predictive nature of proportional and derivative control.

Derivative Action

A controller with proportional action has a significant disadvantage because it does not anticipate what is happening in the future. This is illustrated in Figure 2.5 which shows two error curves, A and B. At time t a proportional controller will give the same control action for both error curves. A significant improvement can be obtained by introducing prediction.

A simple way to predict is to extrapolate the error curve along its tangent. This means

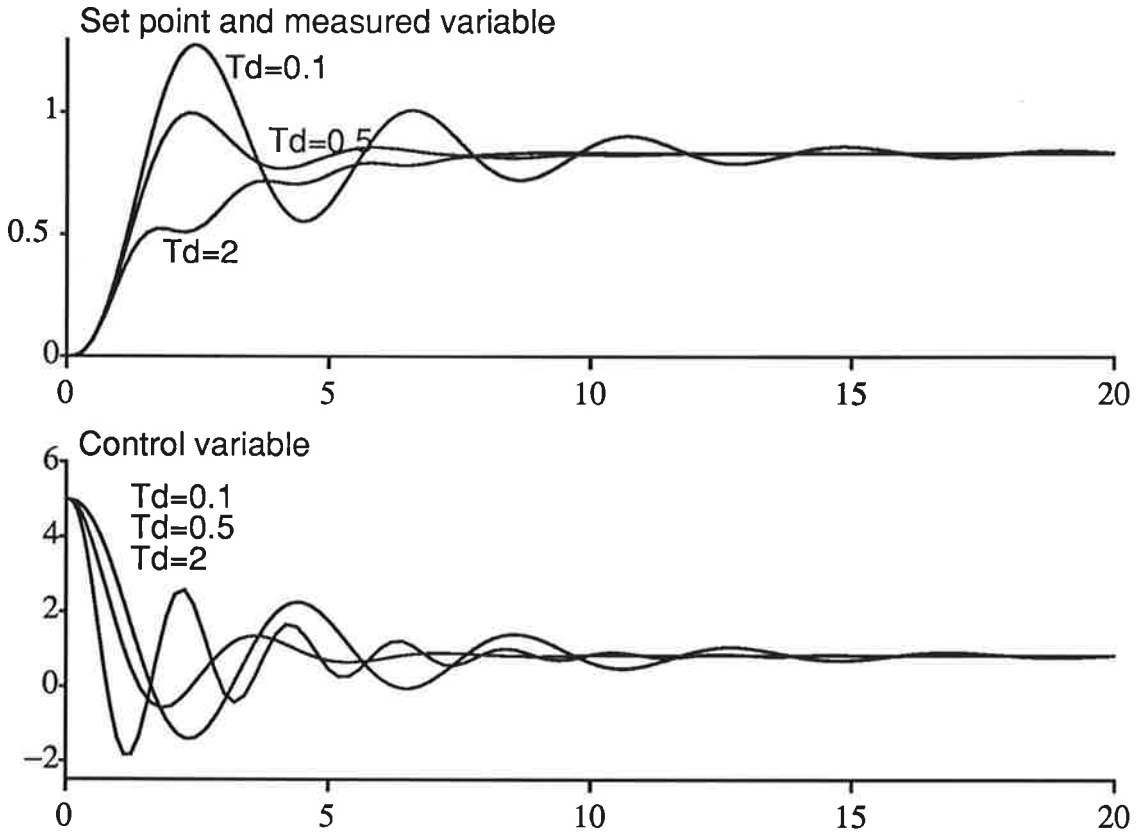

Figure 2.6 Illustration of the damping properties of derivative action. The process has the transfer function $G_p(s) = (s+1)^{-3}$. The gain is $K_c = 5$ and T_d is varied.

that control action is based on the predicted error e_p defined by

$$e_p(t) = e(t) + T_d \frac{de(t)}{dt} \tag{2.6}$$

This gives the control law

$$u(t) = K_c \left[e(t) + T_d \frac{de(t)}{dt} \right] = P + D \tag{2.7}$$

which is a PD controller. With such a controller the control actions for the curves A and B in Figure 2.5 will be quite different. Parameter T_d, which has dimension time, is called *derivative time*. It can be interpreted as a prediction horizon.

The fact that control is based on the predicted output implies that it is possible to improve the damping of an oscillatory system. The properties of a controller with derivative action are illustrated in Figure 2.6. This figure shows that the oscillations are more damped when derivative action is used.

Notice in Figure 2.6 that the output approaches an exponential curve for large values of T_d. This can easily be understood from the following intuitive discussion. If the derivative time is longer than the other time constants of the system the feedback loop can be interpreted as a feedback system that tries to make predicted error e_p small. This implies that

$$e_p = e + T_d \frac{de}{dt} = 0$$

This differential equation has the solution $e(t) = e(0)e^{-t/T_d}$. For large T_d the error thus goes to zero exponentially with time constant T_d.

A drawback with derivative action is that parameter T_d has to be chosen carefully. Industrial PID controllers often have potentiometers to set the parameters K_c, T_i, and T_d. Because of the difficulty in adjusting derivative time T_d the potentiometer for T_d is made so that derivative action can be switched off. In practical industrial installations we often find that derivative action is switched off.

Use of derivative action in the controller has demonstrated that prediction is useful. Prediction by linear extrapolation has, however, some obvious limitations. If a mathematical model of a system is available it is possible to predict more accurately. Much of the thrust of control theory has been to use mathematical models for this purpose. This has led to controllers with observers and Kalman filters.

Limitation of Derivative Gain

A pure derivative can and should not be implemented, because it will give a very large amplification of measurement noise. The gain of the derivative must thus be limited. This can be done by approximating the transfer function sT_d as follows

$$sT_d \approx \frac{sT_d}{1 + sT_d/N} \tag{2.8}$$

The transfer function on the right approximates the derivative well at low frequencies but the gain is limited to N at high frequencies. Parameter N is therefore called *maximum derivative gain.* Typical values of N are in the range 10–20. The approximation given by Eq. (2.8) gives a phase advance of 90° for low frequencies. For $\omega = \sqrt{N/T_d}$ the phase advance has dropped to 45°.

Modification of Set Point Response

In the basic algorithm given by Eq. (2.1) the control action is based on error feedback. This means that the control signal is obtained by filtering the control error. Since the error is the difference between the set point and the measured variable it means that the set point and the measured variable are treated in the same way. There are several advantages in providing separate signal treatments of those signals.

It was observed empirically that it is often advantageous to not let the derivative act on the command signal or to let it act on a fraction of the command signal only. The reason for this is that a step change in the command signal will make drive the output of the control signal to its limits. This may result in large overshoots in the step response. To avoid this the derivative term can be modified to

$$D(s) = \frac{sT_d}{1 + sT_d/N}(\gamma U_c(s) - Y(s)) \tag{2.9}$$

If parameter γ is zero, which is the most common case, the derivative action does not operate on the set point.

It has also been found suitable to let only a fraction β of the command signal act on the proportional part. The PID algorithm obtained then becomes

$$\begin{aligned} U(s) = K_c\Big[\beta U_c(s) - Y(s) &+ \frac{1}{sT_i}(U_c(s) - Y(s)) \\ &+ \frac{sT_d}{1 + sT_d/N}(\gamma U_c(s) - Y(s))\Big] \end{aligned} \tag{2.10}$$

where U, U_c, and Y denote the Laplace transforms of u, u_c, and y. The idea to provide different signal paths for the process output and the command signal is a good way to

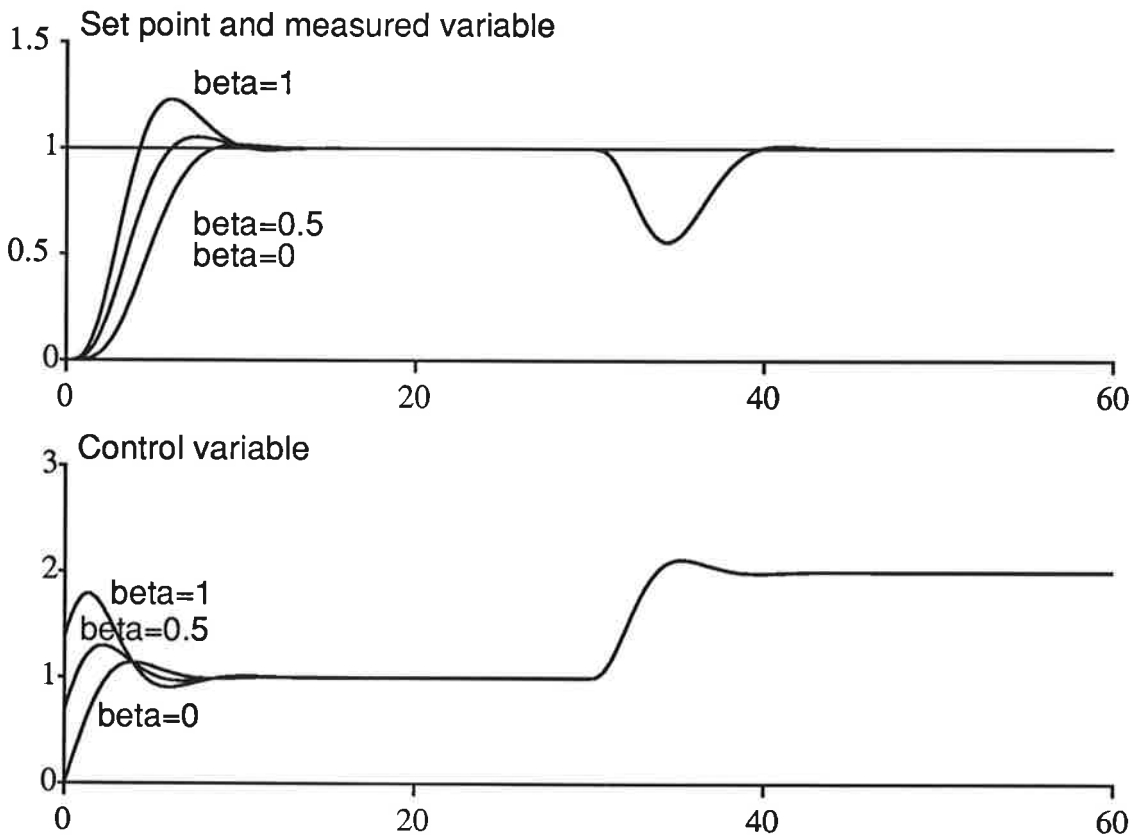

Figure 2.7 Response of system to setpoint changes and load disturbances for controller with different values of parameter β.

separate command signal response from the response to load disturbances. Alternatively it may be viewed as a way to position the closed loop zeros.

The advantages of a simple way to separately adjust responses to load disturbances and set points are illustrated in Figure 2.7. In this case parameters K_c, T_i, and T_d are chosen to give a good response to load disturbances. With $\beta = 1$ the response to set point changes has a large overshoot, which can be adjusted by changing parameter β.

There are also several other variations of the PID algorithm that are used in commercial systems. An extra first order lag may be used in series with the controller to obtain a high frequency roll-off. In some applications it has also been useful to include nonlinearities. The proportional term $K_c e$ can thus be replaced by $K_c e|e|$ or by $K_c e^3$. Analogous modifications of the derivative term have also been used.

Velocity Algorithms

The PID algorithms described so far are called *position algorithms* because the algorithm gives the controller output. In some cases it is natural to perform the integration outside the algorithm. A typical case is when the controller drives a motor. In such a case the controller should give the velocity of the control signal as an output. Algorithms of this type are called *velocity algorithms*. The basic form of a velocity algorithm is

$$U(s) = K_c \left[sE(s) + \frac{1}{T_i} E(s) - \frac{s^2 T_d}{1 + sT_d/N} Y(s) \right] \tag{2.11}$$

Notice that the velocity algorithm in this form can not be used for a controller that has no integral action because it will not be able to keep a stationary value.

Other Parameterizations of the Controller

The algorithm given by Eq. (2.1) or the modified version (2.10) are called a *noninteracting PID controller* or a *parallel form.* There are other parameterizations of the controllers. This is not a major issue in principle, but it can cause significant confusion if we are not aware of it. Therefore we will also give some of the other parameterizations. To avoid unnecessary notations we show the alternative representations in the basic form only.

An alternative to Equation (2.1) that is common in commercial controllers is

$$G'(s) = K_c \frac{(1 + sT_i')(1 + sT_d')}{sT_i'(1 + sT_d'/N')} \tag{2.12}$$

This is called an *interacting form* or a *series form.* Notice that this is the form naturally obtained when integral action is implemented as automatic reset. See Figure 2.3. Simple calculations show that the parameters of (2.12) and (2.1) are related by

$$\begin{aligned} K_c &= K_c' \frac{T_i' + T_d'}{T_i' T_d'} \\ T_i &= T_i' + T_d' \\ T_d &= \frac{T_i' T_d'}{T_i' + T_d'} \end{aligned} \tag{2.13}$$

Since the numerator poles of (2.12) are real the parameters of (2.13) can be computed from (2.1) only if

$$T_i \geq 4T_d \tag{2.14}$$

Then

$$\begin{aligned} K_c' &= \frac{K}{2}\left(1 + \sqrt{1 - \frac{4T_d}{T_i}}\right) \\ T_i' &= \frac{T_i}{2}\left(1 + \sqrt{1 - \frac{4T_d}{T_i}}\right) \\ T_d' &= \frac{T_i}{2}\left(1 - \sqrt{1 - \frac{4T_d}{T_i}}\right) \end{aligned} \tag{2.15}$$

There are also many other parameterizations e.g.

$$G(s) = K + \frac{K_i}{s} + K_d s$$

or

$$G(s) = K + \frac{1}{T_i'' s} + T_d'' s$$

These forms typically appeared when programmers and other persons with no knowledge about control started to develop controllers. These parameterizations have caused a lot of confusion as well as lost production when these forms have been confused with (2.1) without recognizing that the numerical values of the parameters are different.

Summary

The different control actions P, I, and D have been discussed. An intuitive account of their properties have been given. Proportional control provides the basic feedback action, integral action makes sure that the desired steady state is obtained and derivative action provides prediction, which can be used to stabilize an unstable system or to improve the damping of an oscillatory system. The advantages of modifying the idealized algorithm given by Equation (2.1) to

$$U(s) = K_c\Big[\beta U_c(s) - Y(s) + \frac{1}{sT_i}(U_c(s) - Y(s)) + \frac{sT_d}{1 + sT_d/N}(\gamma U_c(s) - Y(s))\Big] \tag{2.16}$$

have been discussed. This formula includes limitation of the derivative gain and facilities for separate control of response to load disturbances and set point changes. The controller in (2.16) has six parameters, the primary PID parameters K_c, T_i, T_d, and maximum derivative gain N, further parameters β and γ are used for set point weighting.

2.3 Anti-Windup and Mode Switches

Many properties of a control system can be understood from linear analysis of small signal behavior. The reason for this is that a control system tries to keep process variables close to their equilibrium. There are, however, some nonlinear phenomena that must be taken into account. There are inherent limitations in the output of any actuator. The speed of a motor, the motion of a linear actuator, the opening of a valve, or the torque of an engine are always limited. Another source of large changes is that real systems can operate in different modes. The behavior of a system during mode changes can naturally not be understood from linear analysis. In this section we will discuss these aspects of PID control that must be considered when designing a good control system. The discussion will result in modifications of the large signal behavior of the PID controller.

Integrator Windup

The combination of a saturating actuator and a controller with integral action give rise to a phenomena called *integrator windup*. If the control error is so large that the integrator saturates the feedback path will be broken because the actuator will remain saturated even if the process output changes. The integrator, being an unstable system, may then integrate up to a very large value. When the error changes sign the integral may be so large that it takes considerable time until the integral assumes a normal value again. The phenomena is also called *reset windup*. It is illustrated in Figure 2.8, which shows a simulation of a process with a PI controller. The process dynamics can be described as an integrator and the process input is limited to the range $-0.1 \leq u \leq 0.1$. The controller parameters are $K_c = 1$ and $T_i = 1$. When a command signal in the form of a unit step is applied the computed control signal is so large that the process actuator saturates immediately at its high limit. Since the process dynamics is an integrator the process output increases linearly with rate 0.1 and the error also decreases linearly. The control signal will, however, remain saturated even when the error becomes zero because the control signal is given by

$$u(t) = K_c e(t) + I$$

The integral has obtained a large value during the transient. The value is proportional to the dashed area in the figure. The control signal does not leave the saturation until

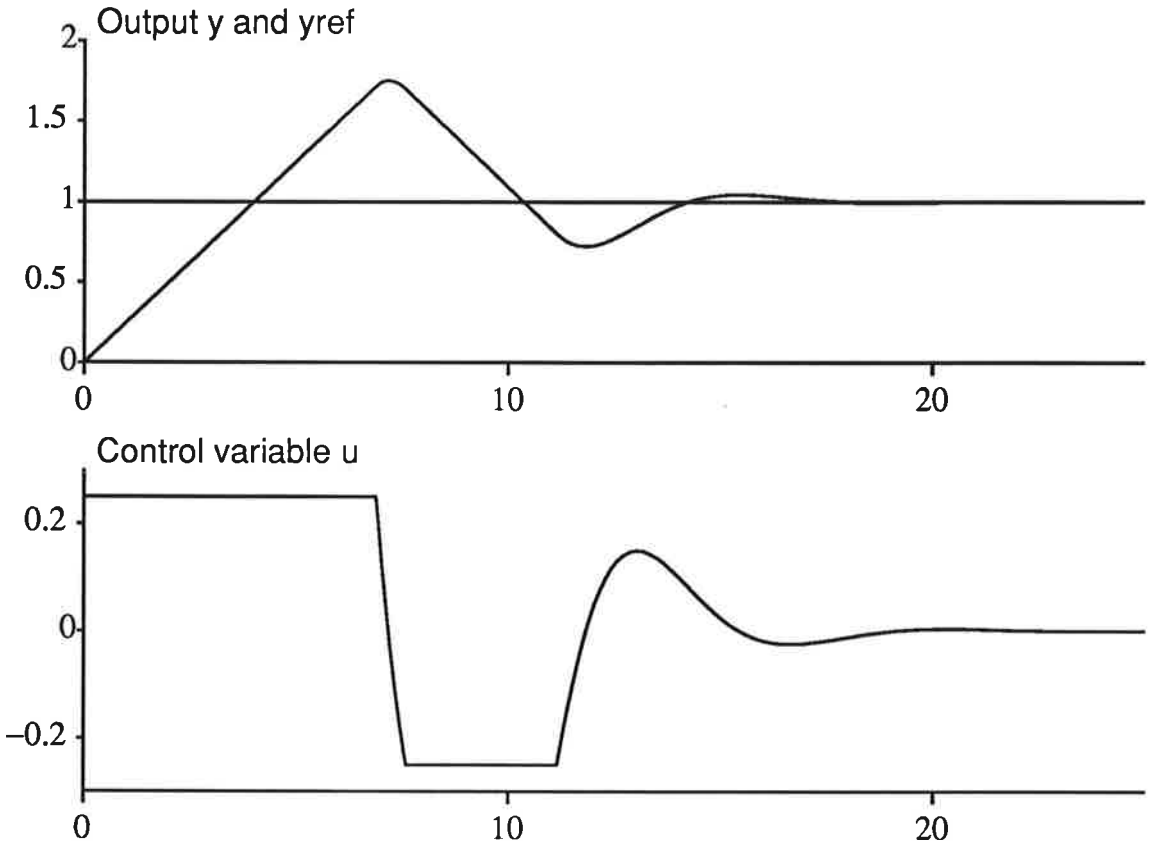

Figure 2.8 Illustration of integrator windup.

the error has been negative for a sufficiently long time to reduce the value of the integral. The net effect is a large overshoot. When the control signal finally leaves the saturation it changes rapidly and saturates again at the lower actuator limit.

Anti-windup

In a good PID controller it is necessary to avoid integrator windup. There are several ways to avoid integrator windup. One possibility is to stop updating the integral when the actuator saturates. This is called *conditional integration.* Another method is illustrated in the block diagram in Figure 2.9. In this method an extra feedback path is provided by measuring the actuator output and forming an error signal (e_s) as the difference between the actuator output (u_c) and the controller output (v). This error is fed back to the integrator through the gain $1/T_t$. The error signal e_s is zero when the actuator does not saturate. When the actuator saturates the extra feedback path tries to make the error signal e_s equal zero. This means that the integrator is reset so that the controller output tracks the saturation limits. The method is therefore called *tracking.* The integrator is reset to the saturation limits at a rate corresponding to the time constant T_t which is called the tracking time constant. The advantage of this scheme for anti-windup is that it can be applied to any actuator as long as the actuator output is measured. If the actuator output is not measured the actuator can be modeled and an equivalent signal can be generated from a mathematical model as shown in Figure 2.9b). It is thus useful for actuators having a dead-zone or an hysteresis.

Figure 2.10 shows the improved behavior obtained with a controller having anti-windup based on tracking. The system simulated is the same as in Figure 2.8. Notice the drastic improvement in performance.

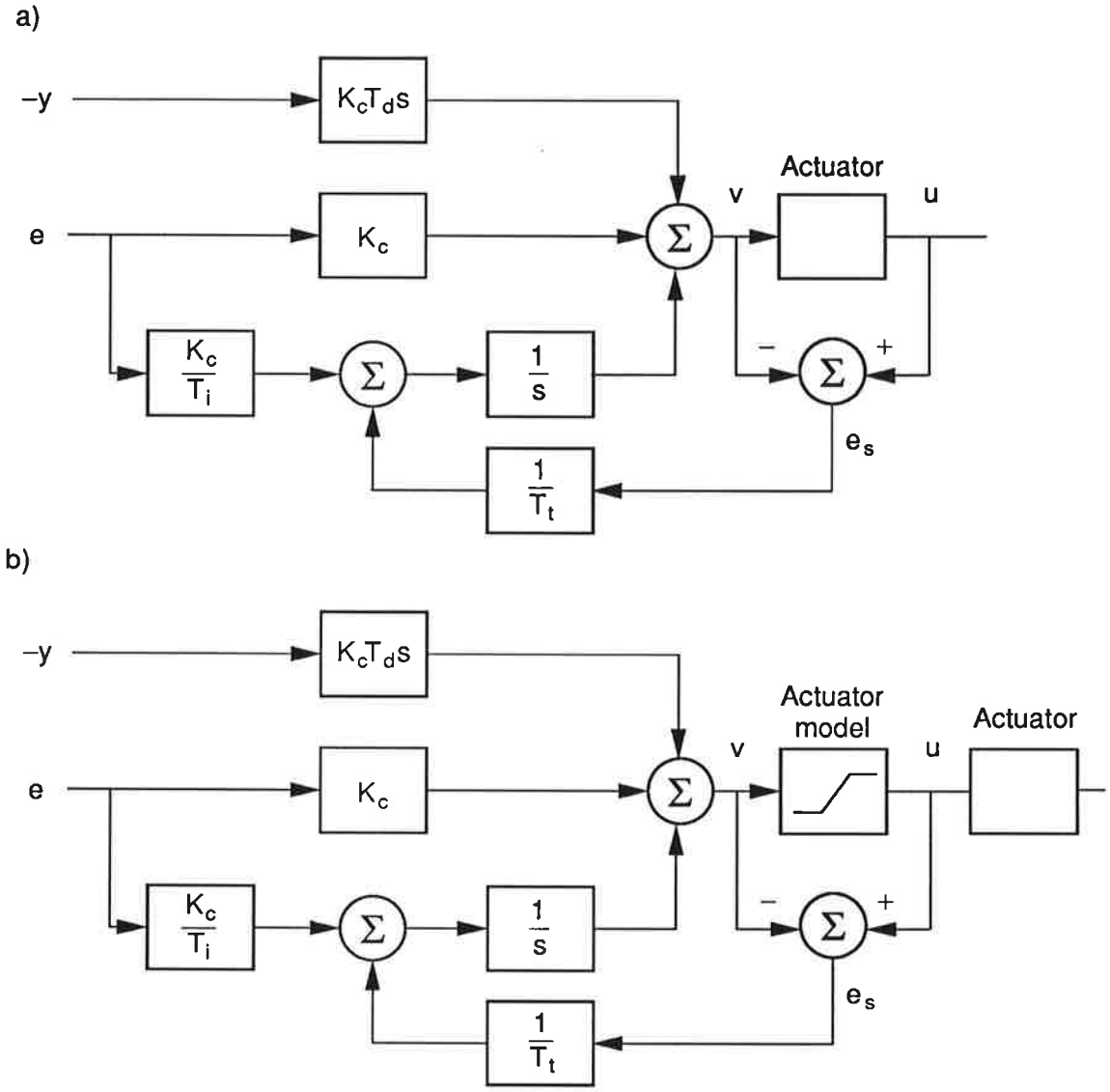

Figure 2.9 Regulator with anti-windup. A system where the actuator output is measured is shown in a) and a system where the actuator output is estimated from a mathematical model is shown in b).

Bumpless Mode Changes

Practically all PID controllers can run in at least two modes: manual and automatic. In the manual mode the controller output is manipulated directly by the operator typically by push buttons that increase or decrease the controller output. When there are changes of modes and parameters it is important to avoid switching transients.

Since a controller is a dynamic system, it is necessary to make sure that the state of the system is correct when switching between manual and automatic mode. When the system is in manual mode, the controller produces a control signal that may be different from the manually generated control signal. It is necessary to make sure that the value of the integrator is correct at the time of switching. This is called *bumpless transfer*. Bumpless switching is easy to obtain for a regulator in incremental form. This is shown in Figure 2.11. The integrator is provided with a switch so that the signals are either chosen from the manual or the automatic increments. Since the switching only influences the increments there will not be any large transients.

A related scheme for a position algorithm where integral action is implemented as automatic reset is shown in Figure 2.11b). Notice that in this case the tracking time

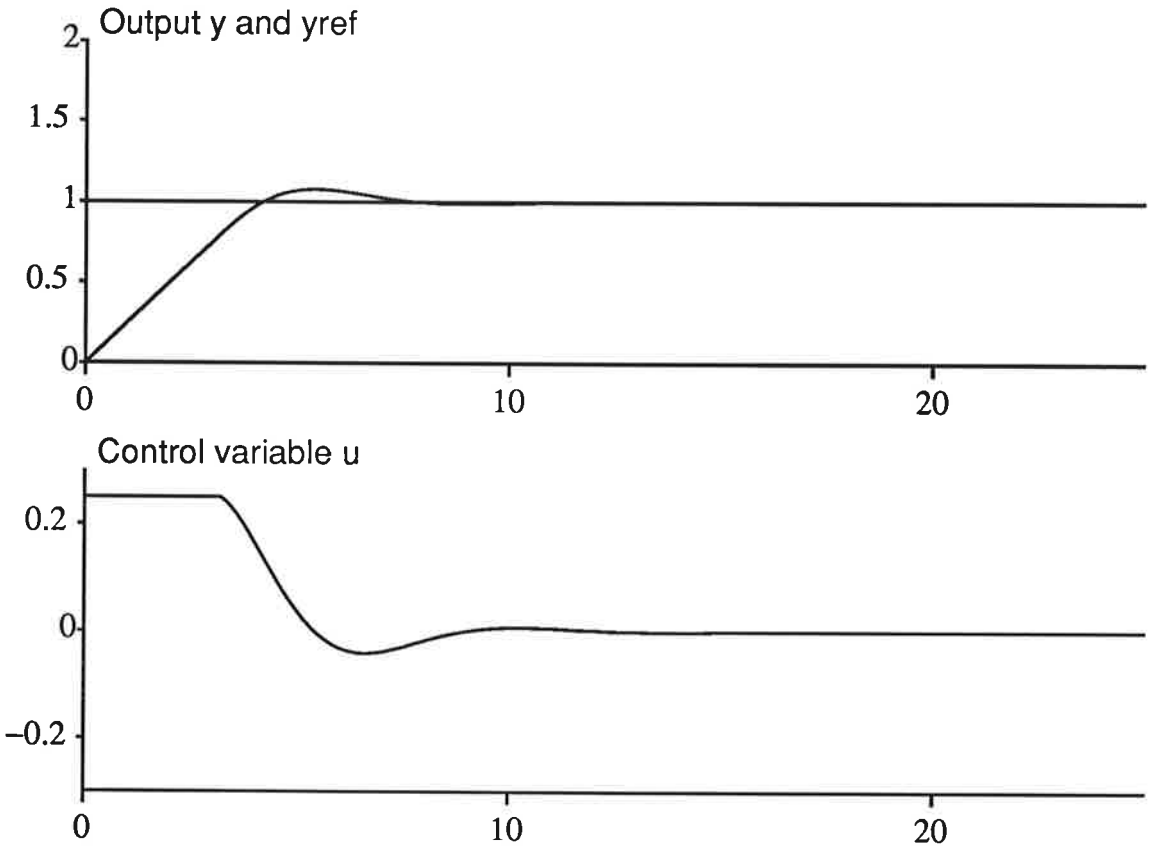

Figure 2.10 Illustration of controller with anti-windup using tracking. Compare with Figure 2.8.

constant is equal to T_i'.

More elaborate schemes have to be used for general PID algorithms on position form. Such a regulator is built up of a manual control modules and one PID module each having an integrator. See Figure 2.12.

Bumpless Parameter Changes

It is also necessary to make sure that there are no unnecessary transients when parameters are changed. A regulator is a dynamical system. A change of the parameters of a dynamical system will naturally result in changes of its output even if the input is kept constant. Abrupt changes in the output can be avoided by a simultaneous change of the state of the system. The changes in the output will also depend on the chosen realization. With a PID regulator it is natural to require that there be no drastic changes in the output if the parameters are changed, when the error is zero. This holds for all incremental algorithms because the output of an incremental algorithm is zero when the input is zero irrespective of the parameter values. It also holds for a position algorithm with the structures shown in Figure 2.11b) and 2.11c). For a position algorithm it depends, however, on the implementation. Assume e.g. that the state chosen to implement the integral action is chosen as

$$x_I = \int^t e(s)ds$$

The integral term is then

$$I = \frac{K_c}{T_i} x_I$$

A change of K_c or T_i then results in a change of I and thus an abrupt change of the controller output. To avoid bumps when the parameters are changed the state should be

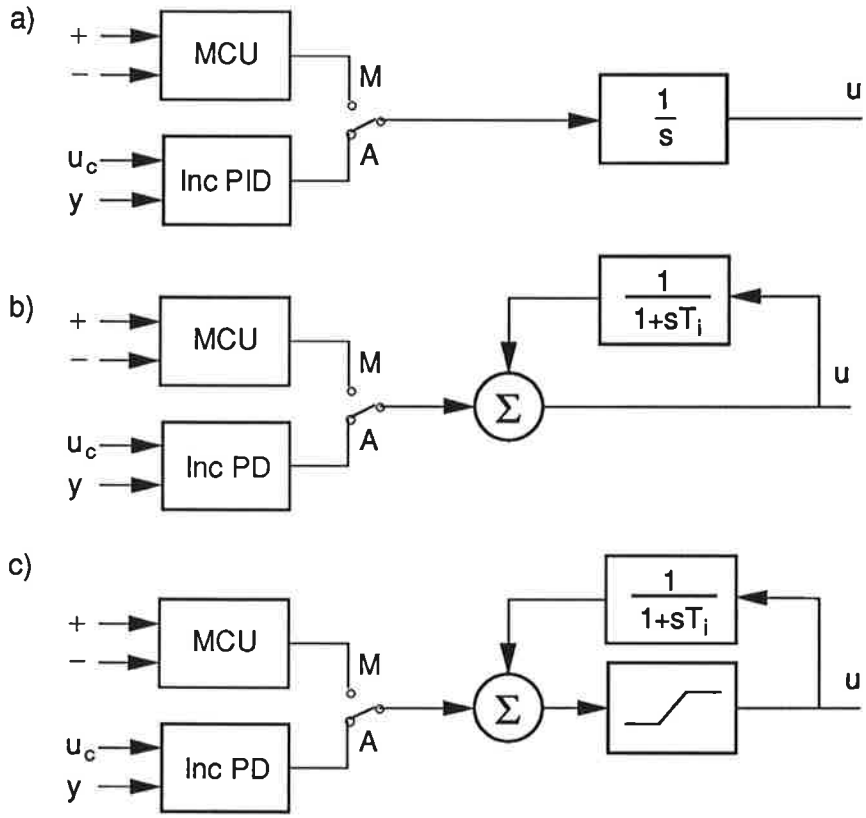

Figure 2.11 Regulators with bumpless transfer from manual (M) to automatic (A) mode. The regulator in a) is incremental. The regulators in b) and c) are special forms of position algorithms. The regulator in c) has anti-windup. MCU is a Manual Control Unit.

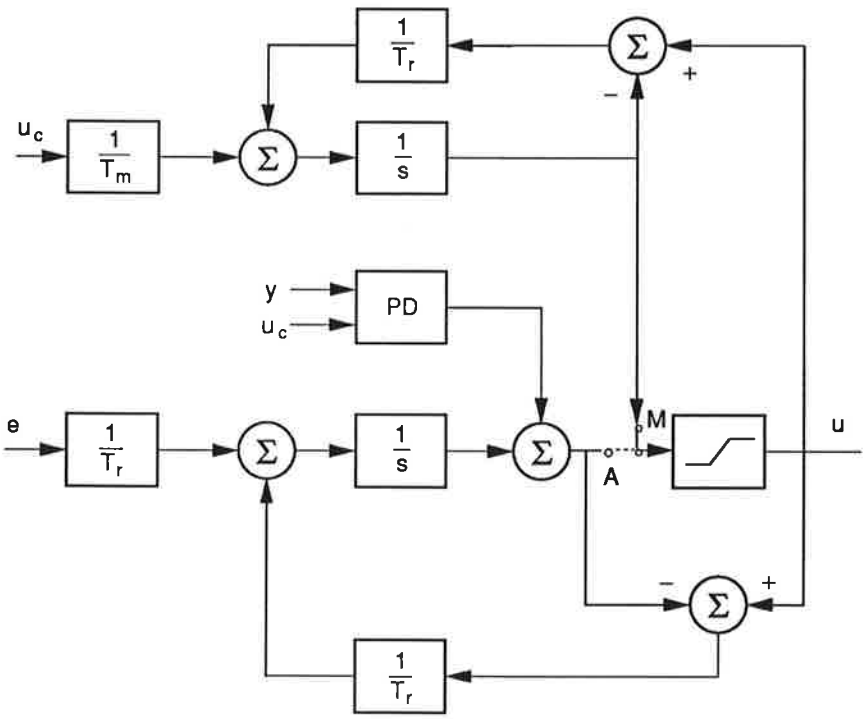

Figure 2.12 PID regulator with bumpless switching between manual (M) and automatic (A) control.

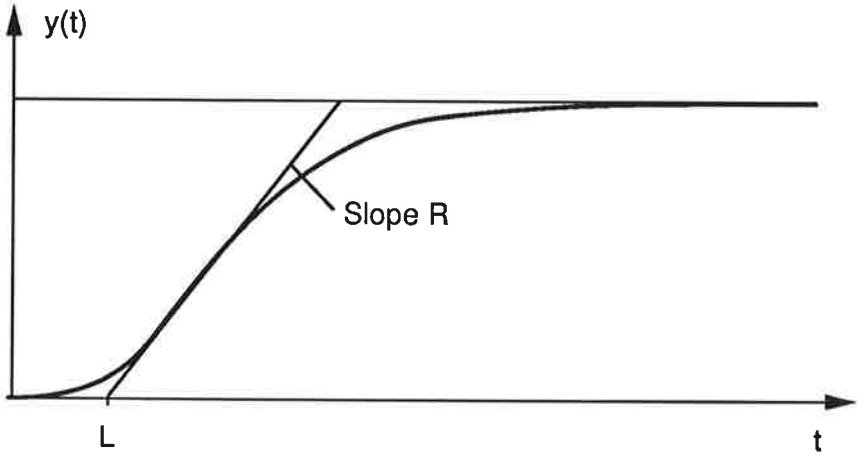

Figure 2.13 Determination of parameters $a = RL$ and L from the unit step response of a process.

chosen as

$$x_I = \frac{K_c}{T_i} \int^t e(s)ds$$

The integral term then becomes

$$I = x_I$$

With this realization a sudden change parameters will not give a sudden change of the controller output.

2.4 Tuning

To make a system with a PID controller work well it is not sufficient to have a good control algorithm. It is also necessary to determine suitable values of the controller parameters. The parameters are K_c, T_i, T_d, N, T_t, β, γ, u_{min} and u_{max}. The primary parameters are

- Controller gain K_c
- Integral time T_i
- Derivative time T_d

Maximum derivative gain N can often be given a fixed default value e.g. $N = 10$. The tracking time constant (T_t) can be chosen in the range $T_d \leq T_t \leq T_i$. In some implementation it has to be equal to T_i in other cases it can be chosen freely. Parameters u_{min} and u_{max} should be chosen inside but close to the saturation limits. Parameters β and γ should be chosen between 0 and 1. Parameter γ can often be set to zero.

There are two different approaches to determine the controller parameters. One method is to install a PID controller and to adjust the parameters until the desired performance is obtained. This is called *empirical tuning*. The other method is to first find a mathematical model of the process and then apply some control design method to determine the controller parameters. Many good tuning procedures combine the approaches.

Much effort has been devoted to find suitable techniques for tuning PID controllers. Significant empirical experience has also been gathered over many years. Two empirical methods developed by Ziegler and Nichols in 1942, the *step response method* and the *ultimate-period method*, have found wide spread use. These methods will be described in the following. A technique based on mathematical modeling will also be given.

The step response method.

In this method the unit step response of the process is determined experimentally. For a process where a controller is installed it can be done as follows: Connect a recorder to the measured variable. Set the controller to manual. Change the controller manually. Calculate scale factors so that the unit step response is obtained. The procedure gives a curve like the one shown in Figure 2.13. Draw the tangent to the step response with the steepest slope. Find the intersection of the tangent with the axes are determined graphically. See Figure 2.13. The controller parameters are then given from Table 2.1. To carry out the graphical construction it is necessary that the step response is such that it is possible to find a tangent with the steepest slope.

The Ziegler-Nichols method was designed to give good response to load disturbances. The design criterion was to achieve *quarter-amplitude-damping* (QAM), which means that the amplitude of an oscillation should be reduced to one quarter after one period of the oscillation. This corresponds to a relative damping of $\zeta = 0.22$ for a second order system. This damping is quite low. In many cases it would be desirable to have better damping this can be achieved by modifying the parameter in Table 2.1.

A good method should not only give a result it should also indicate the range of validity. In the step response method the transfer function of the process is approximated by

$$G(s) = K_p \frac{e^{-sL}}{1 + sT} \tag{2.17}$$

It has already been shown how parameter L can be determined. This parameter is called the *apparent dead-time* because it is an approximation of the dead-time in the process dynamics. Parameter T which is called *apparent time constant* can be determined by

Table 2.1 PID parameters obtained from the Ziegler Nichols step response method.

Regulator type	K_c	T_i	T_d
P	$1/a$		
PI	$0.9/a$	$3L$	
PID	$1.2/a$	$2L$	$0.5L$

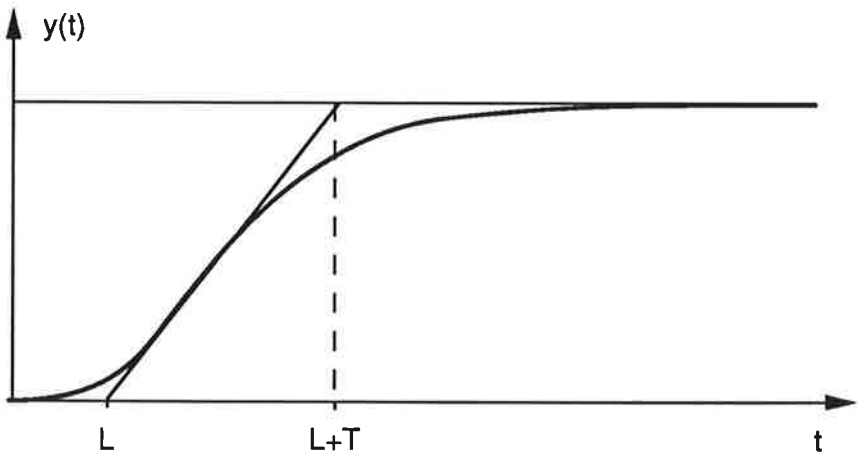

Figure 2.14 Determination of apparent dead-time L and apparent time constant T from the unit step response of a process.

approximating the step response as is indicated in Figure 2.14. The ratio

$$\tau = L/T$$

is useful in order to characterize a control problem. Roughly speaking a process is easy to control if τ is small and difficult to control if τ is large. Ziegler-Nichols tuning is applicable if

$$0.15 < L/T < 0.6 \tag{2.18}$$

The Ultimate-Sensitivity Method

This method is based on the determination of the point where the Nyquist curve of the open loop system intersects the negative real axis. This is done by connecting the controller to the process and setting its parameters so that pure proportional control is obtained. The gain of the controller is then increased until the closed loop system reaches the stability limit. The gain (K_u) when this occurs and the period time of the oscillation (T_u) is determined. These parameters are called *ultimate gain* and *ultimate period.* The regulator parameters are then given by Table 2.2. The ultimate sensitivity method is also based on quarter amplitude damping. The numbers in the table can be modified to give a better damped system.

Let K_p be the static gain of a stable process. The dimension free number K_pK_u can be used to determine if the method can be used. It has been established empirically that the Ziegler-Nichols ultimate sensitivity method can be used if

$$2 < K_pK_u < 20 \tag{2.19}$$

Table 2.2 PID parameters obtained from Ziegler-Nichols ultimate sensitivity method.

Regulator type	K_c	T_i	T_d
P	$0.5K_u$		
PI	$0.45K_u$	$T_u/1.2$	
PD	$0.6K_u$	$T_u/2$	$T_u/8$

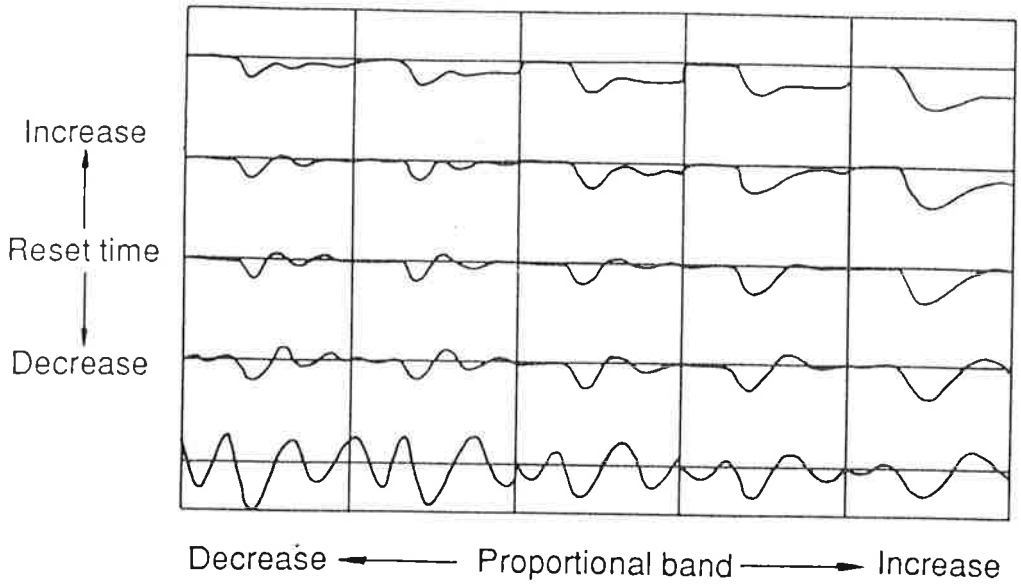

Figure 2.15 Tuning chart for a PI controller. With permission from Foxboro.

Manual Tuning

The Ziegler-Nichols tuning rules give ballpark figures. Experience has shown that the ultimate sensitivity method is more reliable than the step response method. To obtain systems with good performance it is often necessary to do manual fine tuning. This requires insight into how control performance is influenced by the controller parameters. A good way to obtain this knowledge is to use computer simulations. A good experiment is to introduce set point changes and load disturbances. Manufacturers of controllers also provide tuning charts that summarizes experiments of this type for standard cases. An example of such a chart is given in Figure 2.15. Based on experiments rules of the type

Increasing the gain decreases steady state error and stability

Increasing T_i improves stability

Increasing T_d improves stability

can be used for the manual tuning. Notice, however, that simplified rules of this type have limitations. We will illustrate the tuning rules and the use of manual fine tuning by two examples

EXAMPLE 2.1—PI Control
Consider a system with the transfer function

$$G(s) = \frac{1}{(s+1)^4}$$

Let us first apply the step response method. In this case we can find the steepest slope of the tangent and apparent dead-time analytically. Straight forward calculations give $L = 1.42$. The apparent time constant is obtained from the condition $T + L = 4$. Hence $T = 2.58$ and $L/T = 0.55$. Table 2.1 then gives $K_c = 1.88$ and $T_i = 4.70$ for a PI controller.

Applying the ultimate sensitivity method we find that $\omega = 1$ gives the intersection with the negative real axis. I.e.

$$G(i \cdot 1) = -0.25$$

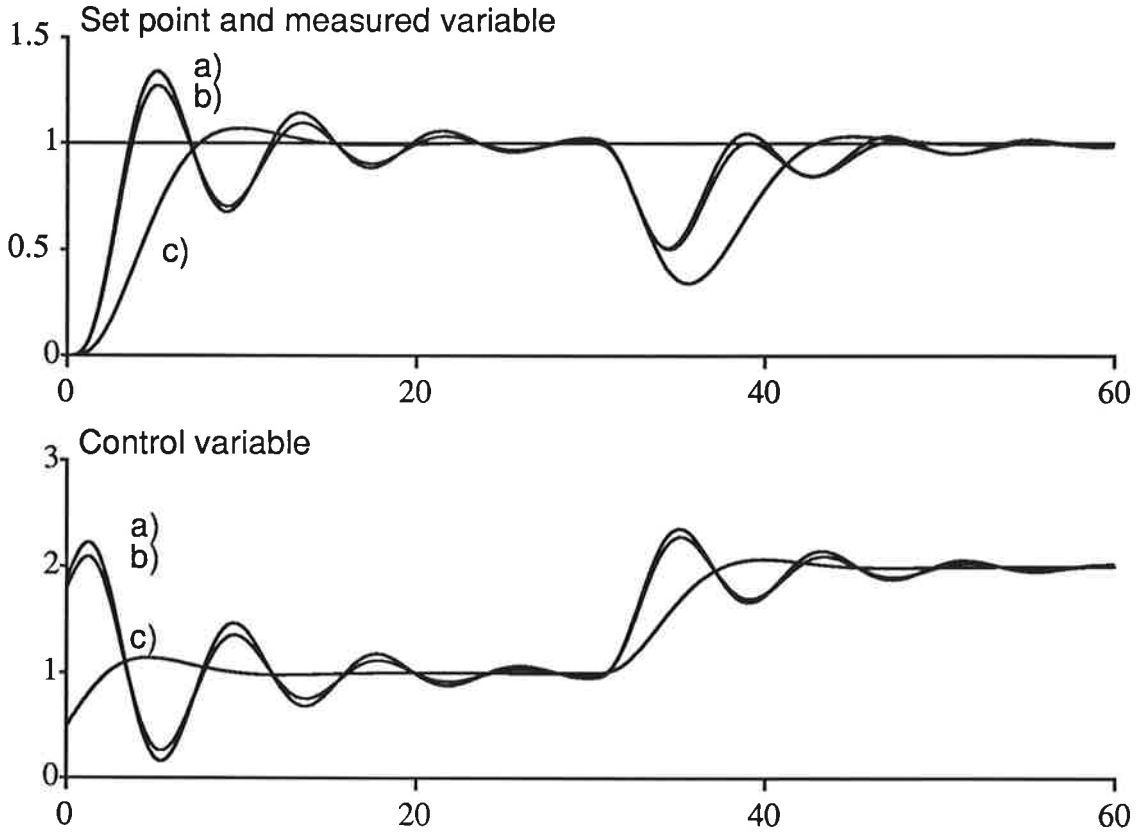

Figure 2.16 Simulation of PI controllers obtained by a) the step response method, b) the ultimate sensitivity method and c) by manual fine tuning.

This implies that $K_u = 4$ and $T_u = 6.28$. Since $K_p = 1$ it follows that $K_u K_c = 4$. The ultimate sensitivity method can thus be applied. Table 2.2 gives the $K_c = 1.8$ and $T_i = 5.2$.

Figure 2.16 we show a simulation of the controllers obtained. In the simulation a unit step command is applied at time zero. At time $t = 30$ a disturbance in the form of a negative unit step is applied to the process input. The figure clearly shows the oscillatory nature of the responses due to the design rule based on quarter amplitude damping. The damping can be improved by manual tuning. After some experimentation we find the parameters $K_c = 0.5$ and $T_i = 2$. This controller gives better damping. The response to the load disturbance is however larger than for the Ziegler-Nichols tuning. □

EXAMPLE 2.2—PID Control

Consider the same system as in the previous example but now use PID control. The step response method, Table 2.1, gives the parameters $K_c = 2.63$, $T_i = 2.85$, $T_d = 0.71$ for a PID controller. Applying the ultimate sensitivity method we find from Table 2.2 $K_c = 2.4$, $T_i = 3.14$ and $T_d = 0.785$.

Figure 2.17 shows a simulation of the controllers obtained. The poor damping is again noticeable. Manual fine tuning gives the parameters $K_c = 1.39$, $T_i = 2.73$, $T_d = 0.793$, and $\beta = 0$ which gives significantly better damping. Notice that it was necessary to put $\beta = 0$ in order to avoid the large overshoot in response to command signals. □

Model Based Tuning

Strictly speaking the Ziegler-Nichols methods are also based on mathematical models. Now we will, however, assume that a mathematical model of the process is available in the form of a transfer function. The controller parameters can then be computed by using

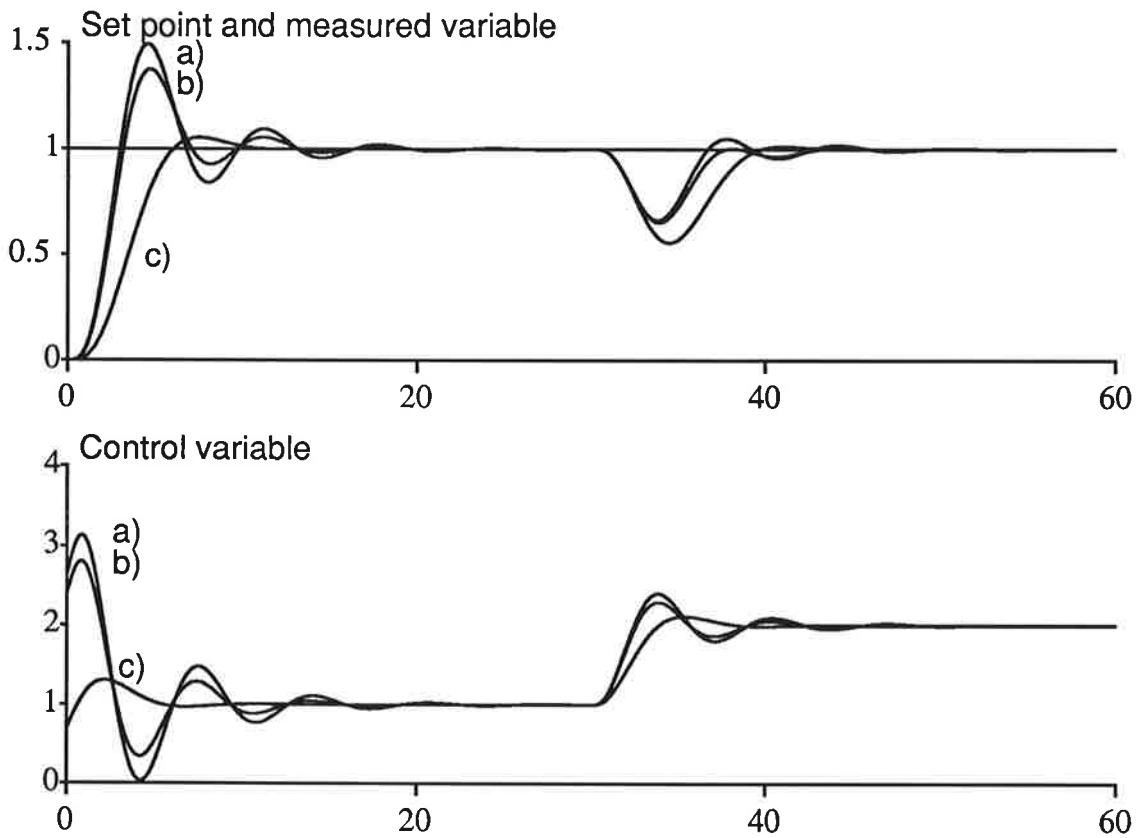

Figure 2.17 Simulation of PID controllers obtained by a) the step response method, b) the ultimate sensitivity method and c) by manual fine tuning.

some method for control system design. We will use a pole placement design technique.

PI control of a first order system

The case when the process is of first order will now be considered. Assume that the process dynamics can be described by the transfer function

$$G_p(s) = \frac{b}{s+a} \tag{2.20}$$

In Laplace transform notation the process is thus described by

$$Y(s) = G_p(s)U(s) \tag{2.21}$$

A PI controller is described by

$$U(s) = K_c\left(\beta U_c(s) - Y(s) + \frac{1}{sT_i}\left(U_c(s) - Y(s)\right)\right) \tag{2.22}$$

Elimination of $U(s)$ between (2.22) and (2.21) gives

$$Y(s) = G_p(s)K_c(\beta + \frac{1}{sT_i})U_c(s) - G_p(s)G_c(s)Y(s)$$

where

$$G_c(s) = K_c\frac{1+sT_i}{sT_i}$$

Hence

$$(1 + G_p(s)G_c(s))Y(s) = G_p(s)K_c(\beta + \frac{1}{sT_i})U_c(s) \tag{2.23}$$

The characteristic equation for the closed loop system is thus

$$1 + G_p(s)G_c(s) = 1 + \frac{bK_c(1 + sT_i)}{sT_i(s + a)} = 0$$

which can be simplified to

$$s(s + a) + bK_c s + \frac{bK_c}{T_i} = s^2 + (a + bK_c)s + \frac{bK_c}{T_i} = 0 \tag{2.24}$$

The closed loop system is thus of second order. By choosing the controller parameters any characteristic equation of second order can be obtained. Let the desired closed loop poles be $-\zeta\omega \pm i\omega\sqrt{1 - \zeta^2}$. The closed loop characteristic equation is then

$$s^2 + 2\zeta\omega s + \omega^2 = 0 \tag{2.25}$$

Identification of coefficients of equal powers of s in (2.24) and (2.25) gives

$$a + bK_c = 2\zeta\omega$$

$$\frac{bK_c}{T_i} = \omega^2$$

Solving these linear equations we get

$$\begin{aligned} K_c &= \frac{2\zeta\omega - a}{b} \\ T_i &= \frac{2\zeta}{\omega} - \frac{a}{\omega^2} \end{aligned} \tag{2.26}$$

This method of designing a controller is called *pole placement* because the controller parameters are determined in such a way that the closed loop characteristic polynomial has specified poles. Notice that for large values of ω we get

$$K_c \approx \frac{2\zeta\omega}{b}$$

$$T_i \approx \frac{2\zeta}{\omega}$$

The controller parameters thus do not depend on a. The reason for this is simply that for high frequencies the transfer function (2.20) can be approximated by $G(s) = b/s$.

Command Signal Response

With controller parameters given by Equation (2.26) the transfer function from the command signal to the process output is

$$G(s) = \frac{(1 + \beta T_i s)\omega^2}{s^2 + 2\zeta\omega s + \omega^2} \tag{2.27}$$

The transfer function thus has a zero at

$$s = -\frac{1}{\beta T_i}$$

The position of the zero can be adjusted by choosing parameter β. Parameter β thus allows us to also position a zero of the closed loop system. Notice that the process (2.20) does not have any zeros. The zero is introduced by the controller if $\beta \neq 0$. For large values of ω the zero becomes $s = -\omega/(2\zeta\beta)$. With $\beta = 1$ this gives a noticeable increase of the overshoot. To avoid this parameter β should be smaller than $0.25/\zeta$.

PID control of a second order system

The case when the process is of second order will now be considered. Assume that the process dynamics can be described by the transfer function

$$G_p(s) = \frac{b}{s^2 + a_1 s + a_2} \tag{2.28}$$

In Laplace transform notation the process is described by

$$Y(s) = G_p(s)U(s) \tag{2.29}$$

A PID controller is described by

$$U(s) = K_c\left(\beta U_c(s) - Y(s) + \frac{1}{sT_i}(U_c(s) - Y(s)) + sT_d(\gamma U_c - Y(s))\right) \tag{2.30}$$

To simplify the algebra we are using a simplified form of the controller ($N = \infty$). Elimination of $U(s)$ between (2.29) and (2.30) gives

$$Y(s) = G_p(s)K_c\left(\beta + \frac{1}{sT_i} + \gamma sT_d\right)U_c(s) - G_p(s)G_c(s)Y(s)$$

where

$$G_c(s) = K_c\frac{1 + sT_i + s^2T_iT_d}{sT_i}$$

is the controller transfer function Hence

$$(1 + G_p(s)G_c(s))Y(s) = G_p(s)K_c(\beta + \frac{1}{sT_i} + \gamma sT_d)U_c(s)$$

The characteristic equation for the closed loop system is

$$1 + G_p(s)G_c(s) = 1 + \frac{bK_c(1 + sT_i + s^2T_iT_d)}{sT_i(s^2 + a_1 s + a_2)} = 0$$

which can be simplified to

$$\begin{aligned} s(s^2 + a_1 s + a_2) + bK_cT_ds^2 + bK_cs + \frac{bK_c}{T_i} &= \\ s^3 + (a_1 + bK_cT_d)s^2 + (a_2 + bK_c)s + \frac{bK_c}{T_i} &= 0 \end{aligned} \tag{2.31}$$

The closed loop system is thus of third order. An arbitrary characteristic polynomial can be obtained by choosing the controller parameters. Specifying the closed loop characteristic equation as

$$(s^2 + 2\zeta\omega s + \omega^2)(s + \alpha\omega) = s^3 + (2\zeta + \alpha)\omega s^2 + (1 + 2\zeta\alpha)\omega^2 s + \omega^3\alpha = 0 \tag{2.32}$$

Equating coefficients of equal powers of s in (2.31) and (2.32) gives the equations

$$\begin{aligned} a_1 + bK_cT_d &= (2\zeta + \alpha)\omega \\ a_2 + bK_c &= (1 + 2\zeta\alpha)\omega^2 \\ \frac{bK_c}{T_i} &= \alpha\omega^3 \end{aligned}$$

Solving these linear equations we get

$$
\begin{aligned}
K_c &= \frac{(1+2\alpha\zeta)\omega^2 - a_2}{b} \\
T_i &= \frac{(1+2\alpha\zeta)\omega^2 - a_2}{\alpha\omega^3} \\
T_d &= \frac{(\alpha+2\zeta)\omega - a_1}{(1+2\alpha\zeta)\omega^2 - a_2}
\end{aligned}
\tag{2.33}
$$

Notice that for large values of ω this can be approximated by

$$
\begin{aligned}
K_c &\approx \frac{(1+2\alpha\zeta)\omega^2}{b} \\
T_i &\approx \frac{1+2\alpha\zeta}{\alpha\omega} \\
T_d &\approx \frac{\alpha+2\zeta}{(1+2\alpha\zeta)\omega}
\end{aligned}
\tag{2.34}
$$

The controller parameters then do not depend on a_1 and a_2. The reason for this is that the transfer function (2.28) can be approximated by $G(s) = b/s^2$ for high frequencies.

Notice that with parameter values (2.34) the polynomial

$$1 + sT_i + s^2 T_i T_d$$

has complex zeros. This means that the controller can not be implemented as a PID controller in series form. Compare with Equation (2.12).

Command signal response

It has thus been shown that he closed loop poles of a system can be changed by feedback. For the PID controller they are influenced by parameters K_c, T_i and T_d. The zeros of the closed loop system will now be considered. With controller parameters K_c, T_i and T_d given by Equation (2.33) the transfer function from the command signal to the process output is

$$G(s) = \frac{\alpha\omega^3(1 + \beta s T_i + \gamma s^2 T_i T_d)}{(s^2 + 2\zeta\omega s + \omega^2)(s + \alpha\omega)} \tag{2.35}$$

The transfer function thus has a zero at the zeros of the polynomial

$$1 + \beta s T_i + \gamma s^2 T_i T_d$$

The zeros of the transfer function can thus be positioned arbitrarily by choosing parameters β and γ. Changing the zeros influences the response to command signals. This is called zero placement. Parameter γ is normally zero. The transfer function (2.35) then has a zero at

$$s = -\frac{1}{\beta T_i}$$

By choosing

$$\beta = \frac{1}{\alpha\omega T_i} = \frac{\omega^2}{(1+2\alpha\zeta)\omega^2 - a_2} \tag{2.36}$$

The zero will cancel the pole at $s = -\alpha\omega$ and the response to command signals is thus a pure second order response. For large values of ω (2.36) reduces to

$$\beta \approx \frac{1}{(1+2\alpha\zeta)}$$

More Complex Models

Chapter 7?

It has thus been demonstrated that it is straight forward to design PI and PID controllers for first and second order systems by using pole-zero placement. From the examples we may expect that the complexity of the controller increases with the complexity of the model. This guess is indeed correct as will be shown in Chapter 7. In that chapter we will also give precise conditions for pole placement.

In the general case the transfer function from command signal to process output has two groups of zeros, one group is the process zero and the other group is zeros introduced by the controller. Only those zeros introduced by the controller can be placed arbitrarily. The case discussed is special in the sense that the process has no zeros.

Summary

In this section we have discussed several methods to determine the parameters of a PID controller. Some empirical tuning rules, the Ziegler-Nichols step response method, and the Ziegler-Nichols ultimate sensitivity method was first described. They are the prototypes for tuning procedures that are widely used in practice. Tuning based on mathematical models were also discussed. A simple method called pole-zero placement was applied to processes that could be described by first and second order models.

If process dynamics can be described by the first order transfer function (2.20) the process can be controlled well by a PI controller. With PI control the closed loop system is of second order and its poles can be assigned arbitrary values by choosing parameters K_c and T_i of the PI controller. The closed loop transfer function from command signal to output has one zero which can be placed arbitrarily by choosing parameter β of the PI controller.

If process dynamics can be described by the second order transfer function (2.28) the process can be controlled well by a PID controller. The closed loop systems of third order and its poles can be placed arbitrarily by choosing parameters K_c, T_i, and T_d of the PID controller. The closed loop transfer function from command signal to process output has two zeros. They can be placed arbitrarily by choosing parameters β and γ of the PID controller.

2.5 Standardized Control Loops

In process control applications there are many standardized control loops. Examples are control of flow, pressure, level, temperature, and concentration. In the following some guidelines are given to facilitate the choice of the controllers and some general rules for controller parameter settings are given. The rules should be used with some caution since there are large variations within the different classes of control loops.

Flow and Liquid Pressure Control

These types of processes are, in general, fast. The time constants are in the order of 1–10 seconds. Measurement disturbances can be considerable and usually of high frequency, but low amplitude. For these types of processes PI controllers are generally used. The derivative term is not used due to the high frequency measurement noise. The controller gain should be moderate.

Gas pressure control

Gas pressure control does in general only require P control or PI control with small amount of integral action. The gain in the controller can be high.

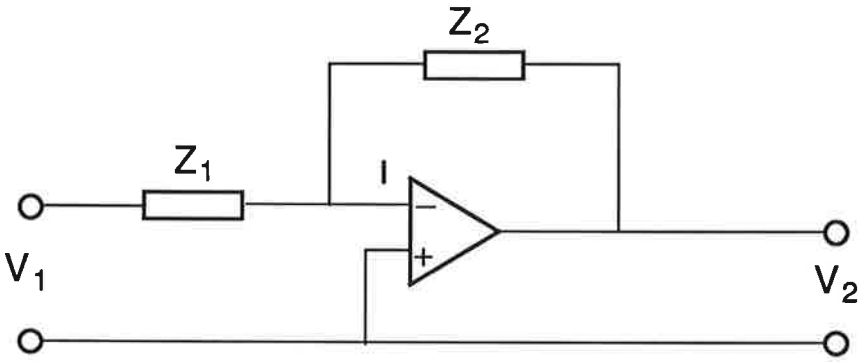

Figure 2.18 Standard circuit with an operational amplifier.

Level Control

Due to splashing the measurements are usually noisy and PI controllers are used. Level control can be used for different purposes. One use of a tank is to use it as a buffer tank. In those cases it is only important to limit the level to the capacity of the tank. This can be done with a low gain P controller or a PI controller with long reset time. In other occasions, such as reactors and evaporators, it can be crucial to keep the level within strict limits. A PI controller with a short reset time should then be used.

Temperature Control

Temperature control loops are in general very slow with time constants ranging from minutes to hours. The noise level is in general low, but the processes often have quite considerable time delays. To speed up the temperature control it is important to use a PID controller. The derivative part will introduce a predictive component in the controller.

Concentration Control

Concentration loops can have long time constants and a variable time delay. The time delays may be due to delays in analyzing equipment. PID controllers should be used if possible. The derivative term can sometimes not be used due to too much measurement noise. The controller gain should in general be moderate.

2.6 Implementation of Controllers

Controllers can be implemented in many different ways. In this section it is shown how PID controllers can be realized using analog and digital techniques.

Analog Implementation

The operational amplifier is an electronic amplifier with a very high gain. Using such amplifiers it is easy to perform analog signal processing. Signals can be integrated, differentiated and filtered. Figure 2.18 shows a standard circuit. If the gain of the amplifier is sufficiently large, the current I is zero. Using Ohm's law, we then get the following relation between the input and output voltages

$$\frac{V_2}{V_1} = -\frac{Z_2}{Z_1} \tag{2.37}$$

By choosing the impedances Z_1 and Z_2 properly, we obtain different types of signal processing. Choosing Z_1 and Z_2 as resistors, i.e. $Z_1 = R_1$ and $Z_2 = R_2$, we get

$$\frac{V_2}{V_1} = -\frac{R_2}{R_1} \tag{2.38}$$

Check Td and N

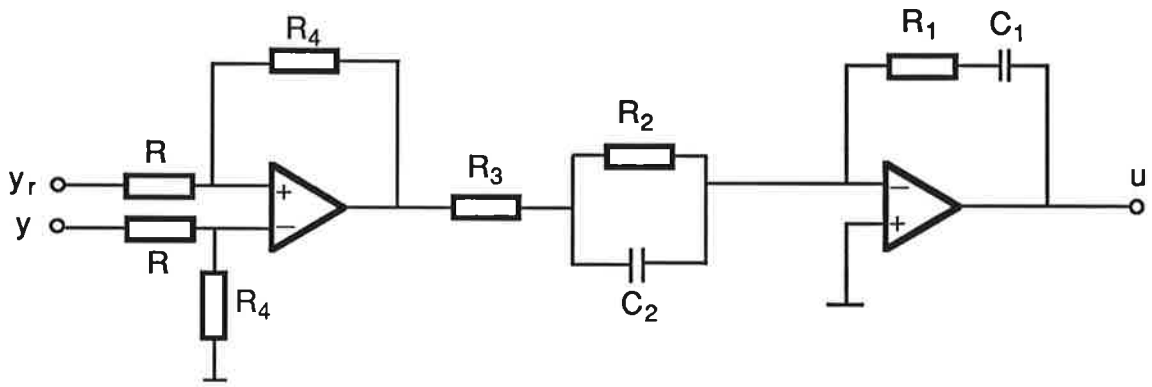

Figure 2.19 Analog circuit that implements a PID controller.

Hence we get an amplifier with gain R_2/R_1. Choosing Z_1 as a resistance and Z_2 as a capacitance, i.e. $Z_1 = R$ and $Z_2 = 1/sC$, we get

$$\frac{V_2}{V_1} = -\frac{1}{RCs} \tag{2.39}$$

which implements an integrator. If Z_1 is chosen as a series connection of a resistor R_1 and a capacitor C_1 and Z_2 is a resistor R_2, we get

$$\frac{Z_2}{Z_1} = \frac{R_2}{R_1 + 1/sC_1} = \frac{R_2C_1s}{1 + R_1C_1s} \tag{2.40}$$

This corresponds to the approximate differentiation used in a PID controller.

By combining operational amplifiers with capacitors and resistors, we can build up PID controllers, e.g. as is shown in Figure 2.19. This circuit implements a controller with the transfer function

$$G(s) = K_c\frac{(1 + sT_i)(1 + sT_d)}{sT_i(1 + sT_d/N)} \tag{2.41}$$

where the controller parameters are given by

$$\begin{aligned} K_c &= R_1R_4/(R(R_2 + R_3)) \\ T_i &= R_1C_1 \\ T_d &= R_2C_2 \\ N &= R_2/R_3 + 1 \end{aligned}$$

There are many other possibilities to implement the controller. For large integration times the circuit in Figure 2.19 may require unreasonable component values. Other circuits that use more operational amplifiers are then used.

Digital Implementations

A digital computer can neither take derivatives nor compute integrals exactly. To implement a PID controller using a digital computer, it is therefore necessary to make some approximations. The proportional part

$$P(t) = K_c(\beta u_c(t) - y(t)) \tag{2.42}$$

requires no approximation since it is a purely static relation. The integral term

$$I(t) = \frac{K_c}{T_i}\int^t e(s)ds \tag{2.43}$$

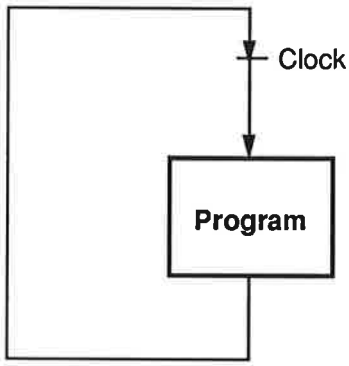

Figure 2.20 A simple operating system.

is approximated by a rectangular approximation, i.e.

$$I(kh+h) = I(kh) + \frac{K_c h}{T_i} e(kh) \tag{2.44}$$

The derivative part given by

$$\frac{T_d}{N}\frac{dD}{dt} + D = -K_c T_d \frac{dy}{dt} \tag{2.45}$$

is approximated by taking backward differences. This gives

$$D(kh) = \frac{T_d}{T_d + Nh} D(kh-h) - \frac{K_c T_d N}{T_d + Nh}(y(kh) - y(kh-h)) \tag{2.46}$$

This approximation has the advantage that it is always stable and that the sampled data pole goes to zero when T_d goes to zero. The control signal is given as

$$u(kh) = P(kh) + I(kh) + D(kh) \tag{2.47}$$

This approximation has the pedagogical advantage that the proportional, integral, and derivative terms are obtained separately. There are many other approximations, which are described in detail in textbooks on digital control.

To introduce a digital version of the anti-windup scheme, we simply compute a signal

$$v(kh) = P(kh) + I(kh) + D(kh) \tag{2.48}$$

The controller output is then given by

$$u = sat(v, u_{min}, u_{max}) = \begin{cases} u_{min} & \text{if } v < u_{min} \\ v & \text{if } u_{min} \le v \le u_{max} \\ u_{max} & \text{if } u > u_{max} \end{cases} \tag{2.49}$$

and the updating of the integral term given by Equation (2.44) is replaced by

$$I(kh+h) = I(kh) + \frac{K_c h}{T_i} e(kh) + \frac{h}{T_r}(u(kh) - v(kh)) \tag{2.50}$$

To implement the controller using a digital computer, it is also necessary to have analog to digital converters that convert the set point u_c and the measure value y to a digital number. It is also necessary to have a digital to analog converter that converts the computed output u to an analog signal that can be applied to the process. To ensure that the control algorithm gets synchronized, it is also necessary to have a clock so that the control algorithm is executed once every h time units. This is handled by an operating system. A simple form of such a system is illustrated in Figure 2.20.

The system works like this. The clock gives an interrupt signal each sampling instant. When the interrupt occurs, the following program is executed:

Analog to digital (AD) conversion of u_c and y
Compute P from (2.42)
Compute D from (2.46)
Compute $v = P + I + D$
Compute u from (2.49)
Digital to analog (DA) conversion of u
Compute I from (2.50)
Wait for next clock pulse

When the interrupt occurs, digital representations of set point u_c and measured value y are obtained from the analog to digital conversion. The control signal u is computed using the approximations described earlier. The numerical representation of u is converted to an analog signal using the DA converter. The program then waits for the next clock signal. An example of a complete computer code in BASIC is given in Listing 2.1.

The subroutine is normally called at line 300. The precomputations starting at line 200 are called when parameters are changed. The purpose of the precomputations is to speed up the control calculations. The command PEEK(100) reads the analog signal and assigns the value u_c to it. The command POKE(200) executes an DA conversion. These operations require that the AD and DA converters are connected as memory mapped IO.

Incremental Algorithms

Equation (2.47) is called *position algorithm* or an *absolute algorithm.* In some cases it is advantageous to move the integral action outside the control algorithm. This is natural when a stepper motor is used. The output of the controller should then represent the increments of the control signal and the motor implements the integrator. Another case is when an actuator with pulse width control is used. In such a case the control algorithm is rewritten so that its output is the increment of the control signal. This is called the *incremental* form of the regulator. A drawback with the incremental algorithm is that it cannot be used for P or PD regulators. If this is attempted the regulator will be unable to keep the reference value because an unstable mode, which corresponds to the integral action is cancelled.

Selection of Sampling Interval and Word Length

The sampling interval is an important parameter in a digital control system. The parameter must be chosen sufficiently small so that the approximations used are accurate, but not so small that there will be numerical difficulties.

Several rules of thumb for choosing the sampling period for a digital PID regulator are given in the literature. There is a significant difference between PI and PID regulators. For PI regulators the sampling period is related to the integration time. A typical rule of thumb is

$$\frac{h}{T_i} \approx 0.1 - 0.3$$

when Ziegler-Nichols tuning is used this implies

$$\frac{h}{L} \approx 0.3 - 1$$

where L is the apparent dead-time or equivalently

$$\frac{h}{T_u} \approx 0.1 - 0.3$$

```
100 REM
    ******************************
    *       PID REGULATOR        *
    * AUTHOR:  K J Astrom        *
    ******************************
110 REM FEATURES
    DERIVATIVE ON MEASUREMENT
    ANTI WINDUP
120 REM VARIABLES
    UC SET DOINT
    Y  MEASUREMENT
    U  REGULATOR OUTPUT
130 REM PARAMETERS
    K  GAIN
    TI INTEGRAL TIME
    TD DERIVATIVE TIME
    TR RESET TIME CONSTANT
    N  MAX DERIVATIVE GAIN
    H  SAMPLING PERIOD
    UL LOW OUTPUT LIMIT
    UH HIGH OUTPUT LIMIT
140 :
200 REM PRECOMPUTATIONS
210 AD=TD/(TD+N*H)
220 BD=K*TD*N/(TD+N*H)
230 BI=K*H/TI
240 BR=H/TR

300 REM MAIN PROGRAM
310 UC=PEEK(100)
320 Y=PEEK(102)
330 P=K*(B*UC-Y)
340 D=AD*D-BD*(Y-YOLD)
350 V=P+I+D
360 U=V
370 IF U<UL THEN UL
380 IF U>UH THEN UH
390 U=POOKE(200)
400 I=I+BI*(UC-Y)+BR*(U-V)
410 YOLD=Y
420 RETURN

END
```

Listing 2.1 BASIC code for PID controller.

where T_u is the ultimate period.

With PID control the critical issue is that the sampling period must be so short that the phase lead is not adversely affected by the sampling. This implies that the sampling period should be chosen so that the number hN/T_d is in the range of 0.2 to 0.6. With $N = 10$ this means that for Ziegler-Nichols tuning we have

$$\frac{h}{L} \approx 0.01 - 0.03$$

or

$$\frac{h}{T_u} \approx 0.0025 - 0.0075$$

Controllers with derivative action thus require significantly shorter sampling periods than PI controllers.

Commercial digital controllers for few loops often have a short fixed sampling interval on the order of 200 ms. This implies that PI control can be used for processes with ultimate periods larger than 0.6 s but that PID controllers can be used for processes with ultimate periods larger than 25 s.

From the above discussion it may appear advantageous to select the sampling interval as short as possible. There are, however, also drawbacks by choosing a very short sampling period. Consider calculation of the integral term. Computational problems, such as *integration offset* may occur due to the finite precision in the number representation used in the computer. Assume that there is an error, $e(kh)$. The integrator term is then increased at each sampling time with

$$\frac{K_c h}{T_i} e(kh) \tag{2.51}$$

Assume that the gain is small or that the reset time is large compared to the sampling time. The change in the output may then be smaller than the quantization step in the DA-converter. For instance, a 12-bit DA converter (i.e., a resolution of 1/4096) should give sufficiently good resolution for control purposes. Yet if $K_c = h = 1$ and $T_i = 3600$, then any error less than 90% of the span of the DA converter gives a calculated change in the integral part less than the quantization step. There will be an offset in the output if the integral part is stored with the same number of digits as used in the DA converter. One way to avoid this is to use higher precision in the internal calculations that are less than the quantization level of the output. Frequently at least 24 bits are used to implement the integral part in a computer, in order to avoid integration offset. It is also useful to use a longer sampling interval when computing the integral term.

2.7 Conclusions

In this chapter we have discussed PID control, which is a generic name for a class of controllers that are simple, common, and useful. The basic control algorithm was discussed in Section 2.2 where we discussed the properties of proportional, integral, and derivative control. The proportional action provides the basic feedback function, derivative action gives prediction and improves stability, integral action is introduced so that the correct steady state values will be obtained. Some useful modifications of the linear behavior of the control algorithm. Large signal properties were discussed in Section 2.3. A particular phenomena called integral windup was given special attention. This led to another modification of the algorithm by introducing the anti-windup feature. In Section 2.4 we discussed different ways of tuning the algorithm both empirical methods and methods based on mathematical models. The design method introduced was pole-zero placement. The controller parameters were simply chosen to give desired closed loop poles and zeros. In Section 2.5 we finally discussed implementation of PID controllers. This covered both analog and digital implementation.

This chapter may be viewed as a nutshell presentation of key ideas of automatic control. Most of the ingredients are here. More sophisticated controllers differ from PID controller mainly in the respect that they can predict better. The pole placement design method can also be applied to more complex controllers. The implementation issues are also similar.

3

Implementation Structures

GOAL: To introduce typical program structures used in basic cases.

The previous chapter described the typical steps that are necessary to transfer a control algorithm into computer code. The result is of the form,

```
PreCompute
Loop
  ADIn
  ComputeOutput
  DAOut
  UpdateStates
```

where much work have resulted in only few lines of code. We will now proceed by describing how this code can be embedded in a program resulting in a complete controller.

3.1 Means of Implementation

Any control system has at least two parallel activities namely the asynchronous man-machine interface and the periodic control algorithm, but there may be more. There are two major means available to obtain parallel processes on a computer. They are real-time operating systems (RTOS) and real-time programming languages (RTPL). The two principles are illustrated in Figure 3.1. An RTOS acts as a layer between the user program and the hardware, and handles e.g. the time-sharing automatically without explicit statements in the user program. The real-time behavior may then vary between different runs of a program (compare with any conventional operating system like Unix). An RPTL is a language containing explicit notations for real-time behavior within the

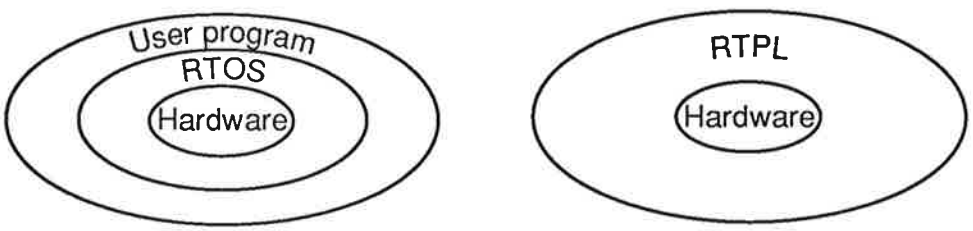

Figure 3.1 Comparison between real-time operating systems (RTOS) and real-time programming languages (RTPL).

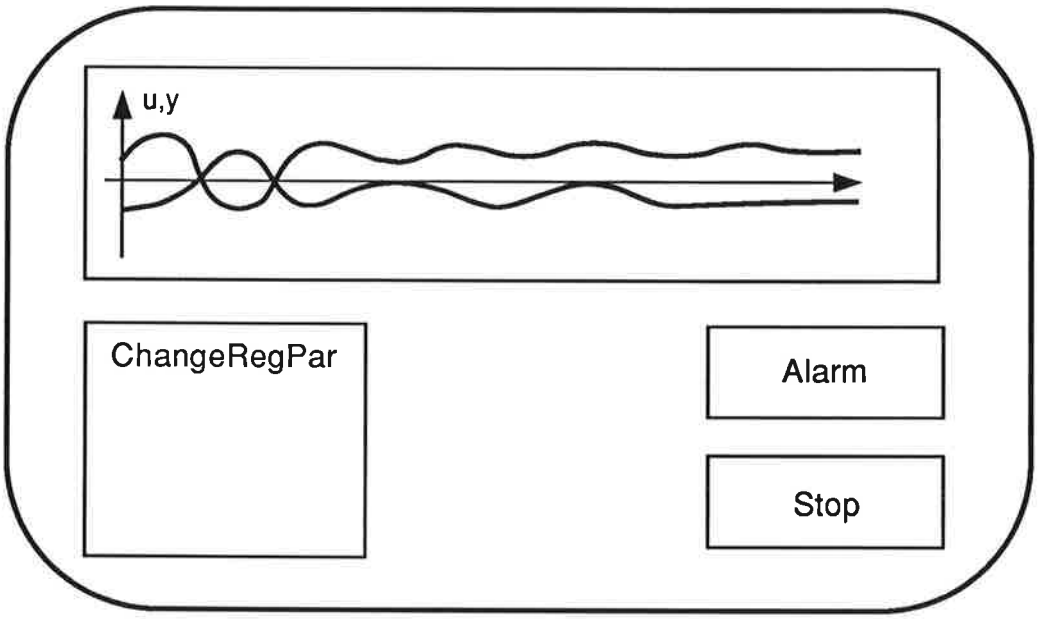

Figure 3.2 An example of a graphical layout on the computer screen.

language itself. An RTPL acts directly on the hardware usually giving a more direct control at the real-time behavior of the system. We will from now on assume that an RTPL is used for the implementation. This does not mean that it has to be a language already equipped with real-time facilities like Ada or Modula-2. It may be any language extended with a real-time kernel as described in Chapter 6.

3.2 Program Specification in a Basic Example

A program specification typically requires that a program is viewed from different perspectives. A specification of a control system requires that the following aspects are treated.

The man-machine interface

It should be possible to switch between manual and auto, to change parameters, to change reference values, and to plot signals. Alarm and emergency stop is usually required.

Both the graphical layout and the command structure should be specified. The graphical layout may be described as in Figure 3.2. The command structure requires a specification of the commands, possible subcommands in different situations, and of active/inactive commands. Subcommands should appear only when applicable, and inactive commands should be dimmed or deleted from the screen. The graphical layout may get complicated if the presentation requires several alternative looks of the screen, if some parts should be changed in various situations, and so on.

The real-time structure

The real-time structure shown in Figure 3.3 describes the principle behavior of many control systems. There are three processes and two monitors. The process `OpCom` (Operator Communication) contains the man-machine interface. The monitor `RegulMonitor` contains the controller parameters and the monitor `PlotMonitor` is a buffer for values to be plotted. The plot process `Plot` is synchronous with the control process `Regul`, but it is usually not natural to combine them into one process. One reason is efficiency of the control process. Another reason is that the control process would be obscured if it should handle different plot commands like change of plot mode, switching plotting on/off,

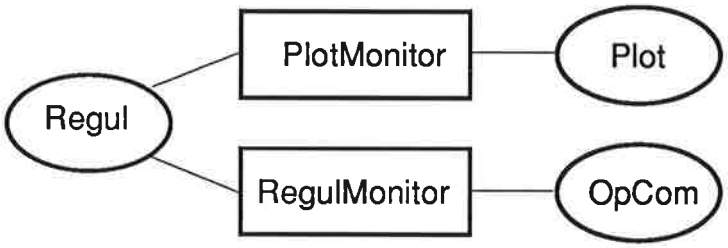

Figure 3.3 A process graph for a control system.

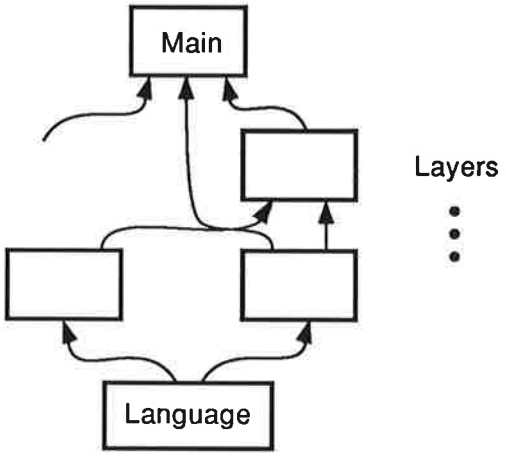

Figure 3.4 A module structure with export/import indication.

changing plot interval, and so on. This real-time structure is usually sufficient with some minor additions e.g. an alarm process.

The control structure

The logical control of the program modes, e.g., the regulator modes, should be specified. It essentially requires an implementation of a sequential net, and possible solutions can be based on Boolean variables in `RegulMonitor`, or on enumeration types in `RegulMonitor`, or by message passing between processes, or on other methods.

The data structure

The data structure is simple in this basic case. `RegulMonitor` contains the parameters used in OpCom, like k, T_i, T_d. Some default rules may be used e.g., that the tracking time constant equals the integration time, $T_t = T_i$.

The module structure

The structuring of the code in modules should be specified. An export/import analysis can be presented as in Figure 3.4. A rule of thumb is that such a figure should not be too messy.

Implementation

An example of an implementation of the system specified above is given in Chapters 6 and 7.

3.3 An automation system

The basic example will now be extended to a principle solution for a complete automation system, where the system should be able to control several loops e.g., in a large and complex process. It should be possible to create and delete controllers, to connect them together, and to connect them to the physical process. The same real-time structure as in Figure 3.3 will be used, but the data-structure will be modified, and the controller process and the man-machine interface accordingly.

The data structure

The data structure will now be a linked list, where each node represents a controller. An example of a node is

```
name
samplingrate
timer
input        ("address")
reference    ("address")
outport      ("address")
outvalue
<regpar>
<intpar>
ulimits      (Extra variables
.             for anti-windup,
manual        bumpless transfer,
.             ...
oldstate      ...)
```

The controller process

The list of controllers in the data structure is scanned periodically by `Regul` by using the minimum sampling rate. For each node in the list the following operations are done

```
count down timer
if timer=0 then
  timer=samplingrate
  outvalue=PID(,,)
  if (outport not nil) then DAOut(outvalue,outport)
  UpdateStates
```

The last if-statement ensures that the outvalue is not sent out if it is only used as the input to another controller.

The man-machine interface

The creation or deletion of a controller implies that a node should be created or deleted. Modifying the connections means that the input and/or reference addresses should be changed. This case and the case of changing other parameters in a node is solved by localizing the relevant node (nodes) in the list by using the `name`.

The order of computation should be possible to specify, so that the outer loop in a cascade can be specified to be computed before the inner loop. `OpCom` should then insert the nodes in the right order in the list.

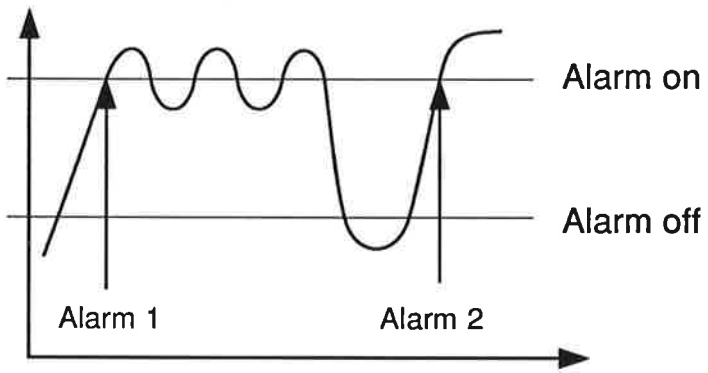

Figure 3.5 Alarm with hysteresis

Interesting problems

The previous chapter treated anti-windup and bumpless transfer only for single loops. We now have freedom to connect controllers in complicated ways. Therefore, one has to reconsider the non-linear behavior. Just to mention one problem: How should the integrator in the outer loop of a cascade be treated if the inner loop controller saturates?

3.4 Further extensions

The automation system can be extended further. For example, digital filtering possibly faster than the control rate can be introduced. A natural step then is to introduce data-structures for input signals and output signals containing for example: scale factors to engineering units, filter constants, limits on both the value and the derivative, and selection of analog or pulsed output.

Alarms

Alarm handling is non-trivial. A by now classical example is the Harrisburg accident. In that case the operator was overwhelmed by numerous alarms and was not able to sort out the relevant information about the original cause to the problems.

There are two problems concerning alarm handling. One is easy to solve, and the other is difficult. The first problem is that one error source may create several alarms by itself. This case is solved by using alarm with hysteresis as shown in Figure 3.5. The other more difficult problem is that an error may propagate and create alarms from several other sources. A complicating factor when trying to isolate the original cause, is that we are dealing with parallel activities and the alarms do not need to arrive to the alarm handler in order of occurance. Methods for alarm analysis is a research topic.

Shutdown procedures

An example of a shutdown procedure is

```
if stop then u:=0
```

This example can of course be extensively elaborated.

Function block programming

Change the line `outvalue=PID(,,)` to `outvalue=function(,,)` and supply a set of functions. Each node in the list has to be extended with a specification of what `function` to be called. `OpCom` needs to be extended with the choice of the function for each node.

Graphical interaction

Point and click to define a node and put it in the list. Automatic generation of connections (values of input and reference addresses), and of computation order (sorting) can support the user of the system.

Sequencing control and reference values

Additional desired functionalities are discussed in Chapters 4 and 5.

4

Sequencing Control

B. Wittenmark

GOAL: To give a brief overview of sequencing control and to introduce GRAFCET.

4.1 Introduction

The history of logical nets and sequential processes starts with batch processing, which is very important in many process industries. This implies that control actions are done in a sequence where the next step depends on some conditions. Simple examples are recipes for cooking and instructions or manuals for equipment. Washing machines, dish washers, and batch manufacturing of chemicals are other examples. The recipes or instructions can be divided into a sequence of steps. The transition from one step to the next can be determined by a logical expression. In this chapter logical and sequencing controllers are presented.

Sequencing control is an integrated part of many control systems, and can be used for batch control, start-up and shut-down procedures, interlooks, and alarm supervision. When implementing such systems it is useful to be aware of the notations and traditions in sequence control as it developed before it started to be integrated with automatic control. This chapter will start by giving a brief background starting with logic or *Boolean algebra*, that plays an important role and is presented in Section 4.2. The axioms and basic rules for evaluation of logical expressions are given. Logical expressions are sometimes also called combinatory networks. Using combinational networks and a memory function it is possible to build so called *sequential nets*. Section 4.3 is devoted to sequencing controllers. One way to present function procedures, that is becoming a popular industrial notation, is to use GRAFCET, which is described in Section 4.4.

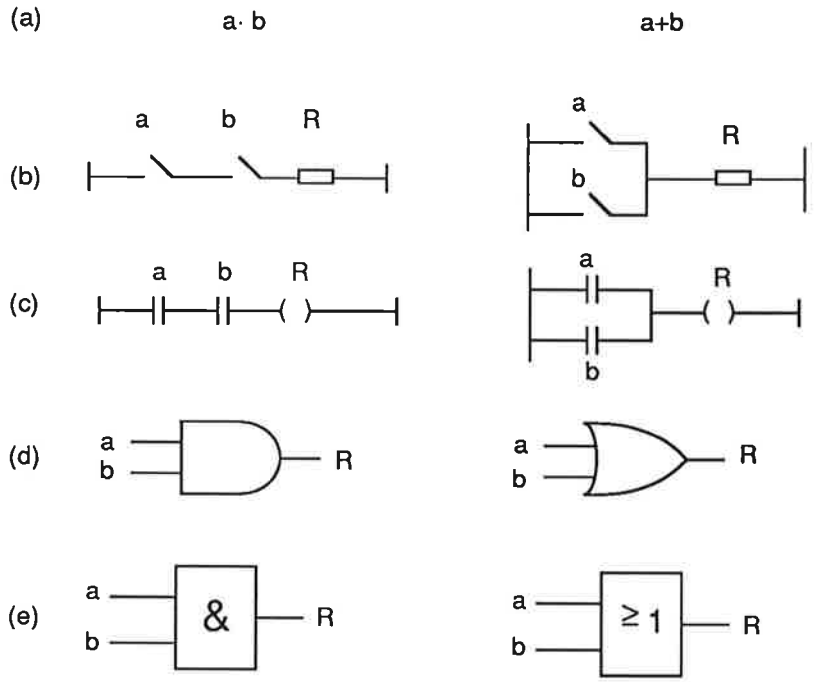

Figure 4.1 Different ways to represent the logical expressions $a \cdot b$ and $a + b$: (a) Boolean algebra; (b) Relay symbols; (c) Ladder symbols; (d) Logical circuit symbols (American standard); (e) Logical circuit symbols (Swedish standard).

4.2 Background in Logical Nets and Boolean Algebra

Logical nets can be implemented in many different ways, for instance, using clever mechanical constructions, relays, transistors, or computers. The computer implementations are often called PLC (Programmable Logical Controller) systems. An earlier notation was PC (Programmable Controller) systems. This was used before PC became synonymous with personal computer. Logical systems are built up by variables that can be true or false, i.e. the variables can only take one of two values. For instance, a relay may be drawn or not, an alarm may be set or not, a temperature may be over a limit or not. The output of a logical net is also a two-valued variable, a motor may be started or not, a lamp is turned on or off, a contactor is activated or not. The mathematical basis for handling this type of systems is Boolean algebra. This algebra was developed by the English mathematician George Boole in the 1850's. It was, however, not until after about a century that it became a widely used tool to analyze and simplify logical circuits.

Logical variables can take the values true or false. These values are often also denoted by 1 (true) or 0 (false). Boolean algebra contains three operations `or`, `and`, and `not`. We have the following notations:

and:	$a \cdot b$	a and b	$a \wedge b$
or:	$a + b$	a or b	$a \vee b$
not:	$\bar{a}$	not a	$\neg a$

The expression $a \cdot b$ is true only if both a and b are true at the same time. The expression $a + b$ is true if either a or b or both a and b are true. Finally the expression $\bar{a}$ is true only if a is false. In the logical circuit symbols a ring on the input denotes negation of the signal and a ring on the output denotes negation of the computed expression. The `and` and `or` expressions can also be interpreted using relay symbols as in Figures 4.1 and 4.2. The `and` operator is the same as series connection of two relays. There is only a connection if both relays are drawn. The `or` operator is the same as parallel connection of two relays. There is a connection whenever at least one of the relays are drawn. The relay

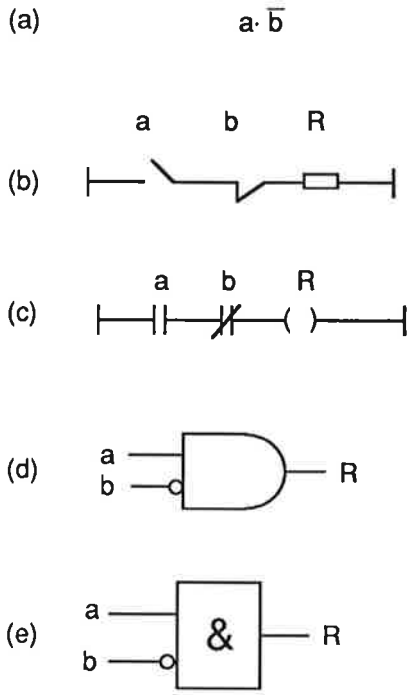

Figure 4.2 Different ways to represent the logical expression $a \cdot \bar{b}$: (a) Boolean algebra; (b) Relay symbols; (c) Ladder symbols; (d) Logical circuit symbols (American standard); (e) Logical circuit symbols (Swedish standard).

or ladder representation of logical nets is often used for documentation and programming of PLCs. We will use the more programming or computer oriented approach with $\cdot$ and $+$. This usually gives more compact expressions and is also more suited for algebraic manipulations. Exactly as in ordinary algebra we may simplify the writing by omitting the and-operator, i.e. to write ab instead of $a \cdot b$.

To make the algebra complete we have to define the *unit* and *zero elements*. These are denoted by 1 and 0 respectively. We have the following axioms for the Boolean algebra:

$$\bar{0} = 1$$
$$\bar{1} = 0$$
$$1 + a = 1$$
$$0 + a = a$$
$$1 \cdot a = a$$
$$0 \cdot a = 0$$
$$a + a = a$$
$$a + \bar{a} = 1$$
$$a \cdot \bar{a} = 0$$
$$a \cdot a = a$$
$$\bar{\bar{a}} = a$$

We further have the following rules for calculation

$a + b = b + a$	Commutative law
$a \cdot b = b \cdot a$	Commutative law
$a \cdot (b + c) = a \cdot b + a \cdot c$	Distributive law
$a \cdot (b \cdot c) = (a \cdot b) \cdot c$	Associative law
$a + (b + c) = (a + b) + c$	Associative law
$\overline{a + b} = \bar{a} \cdot \bar{b}$	de Morgan's law
$\overline{a \cdot b} = \bar{a} + \bar{b}$	de Morgan's law

A logical net can be regarded as a static system. For each combination of the input signals there is only one output value that can be obtained.

In many applications it can be easy to write down the logical expressions for the system. In other applications the expressions can be quite complicated and it can be desirable to simplify the expressions. One reason for making the simplification is that the simplified expressions give a clearer understanding of the operation of the network. The rules above can be used to simplify logical expressions. One very useful rule is the following

$$a + a \cdot b = a \cdot 1 + a \cdot b = a \cdot (1 + b) = a \cdot 1 = a \tag{4.1}$$

One way to test equality between two logical expressions is a *truth table*. The truth table consists of all combinations of the input variables and the evaluation of the two expressions. Since the inputs only can take two values there will be 2^n combinations, where n is the number of inputs. The expressions are equal if they have the same value for all combinations.

EXAMPLE 4.1—Truth table
For instance (4.1) is proved by using the table:

a	b	$a + ab$	a
0	0	0	0
0	1	0	0
1	0	1	1
1	1	1	1

To the left we write all possible combinations of the input variables. To the right we write the value of the expressions of the left and right hand sides of (4.1). The last two columns are the same for all possible combinations of a and b, which proves the equality. □

There are systematic methods to make an automatic reduction of a logical expression. The methodologies will only be illustrated by an example.

EXAMPLE 4.2—Systematic simplification of a logical network
Consider a logical network that has three inputs a, b, and c and one output y. The network is defined by the following truth table:

	a	b	c	y
v_0	0	0	0	0
v_1	0	0	1	0
v_2	0	1	0	0
v_3	0	1	1	1
v_4	1	0	0	1
v_5	1	0	1	1
v_6	1	1	0	1
v_7	1	1	1	1

The different combinations of the inputs are denoted v_i, where the index i corresponds to the evaluation of the binary number abc. I.e. the combination $abc = 101$ corresponds to the number $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5$. The expression for the output y can be expressed in two ways: either as the logical or of the combinations when the output is true or as the negation of the logical or of the combinations when the output is false. Using the first

representation we can write

$$\begin{aligned} y &= v_3 + v_4 + v_5 + v_6 + v_7 \\ &= \bar{a}bc + a\bar{b}\bar{c} + a\bar{b}c + ab\bar{c} + abc \\ &= bc(\bar{a} + a) + a\bar{b}(\bar{c} + c) + ab(\bar{c} + c) \\ &= bc + a\bar{b} + ab = bc + a(\bar{b} + b) \\ &= a + bc \end{aligned}$$

The first equality is obtained from the truth table. The second equality is obtained by combining the terms v_3 with v_7, v_4 with v_5, and v_6 with v_7. It is possible to use v_7 two times since $v_7 + v_7 = v_7$. The simplifications are then given from the computational rules listed above.

The second way to do the simplification is to write

$$\begin{aligned} \bar{y} &= v_0 + v_1 + v_2 \\ &= \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + \bar{a}b\bar{c} \\ &= \bar{a}\bar{b}(\bar{c} + c) + \bar{a}\bar{c}(\bar{b} + b) \\ &= \bar{a}\bar{b} + \bar{a}\bar{c} = \bar{a}(\bar{b} + \bar{c}) \end{aligned}$$

This gives

$$y = \bar{\bar{y}} = \overline{\bar{a}(\bar{b} + \bar{c})} = a + bc$$

which is the same as before. □

Using the methodology described in the example above it is possible to reduce a complicated logical expression into its simplest form. A more formal presentation of the methods are outside the scope of this book.

PLC implementation

Most PLC units are implemented using microprocessors. This implies that the logical inputs must be scanned periodically. A typical block diagram is shown in Figure 4.3. The execution of the PLC program can be done in the following way:

1. Input-copying. Read all logical input variables and store them in I/O memory.
2. Scan through the program for the logical net and store the computed values of the outputs in the I/O memory.
3. Output-copying. Send the values of output signals from the I/O memory to the process.
4. Repeat from 1.

The code for the logical net is executed as fast as possible. The time for execution will, however, depend on the length of the code. The I/O-copying in Step 1 is done to prevent the logical signals to change value during the execution of the code. Finally all outputs are changed at the same time. The programming of the PLC-unit can be done from a small programming module or by using a larger computer with a more effective editor.

The programming is done based on operations such as logical `and`, logical `or`, and logical `not`. Also there are typically combinations such as `nand` and `nor`, which are defined as

$$a \,\mathrm{nand}\, b = \overline{a \cdot b} = \bar{a} + \bar{b}$$

$$a \,\mathrm{nor}\, b = \overline{a + b} = \bar{a} \cdot \bar{b}$$

Further there are operations to set timers, to make conditional jumps, to increase and decrease counters etc. The specific details differ from manufacturer to manufacturer.

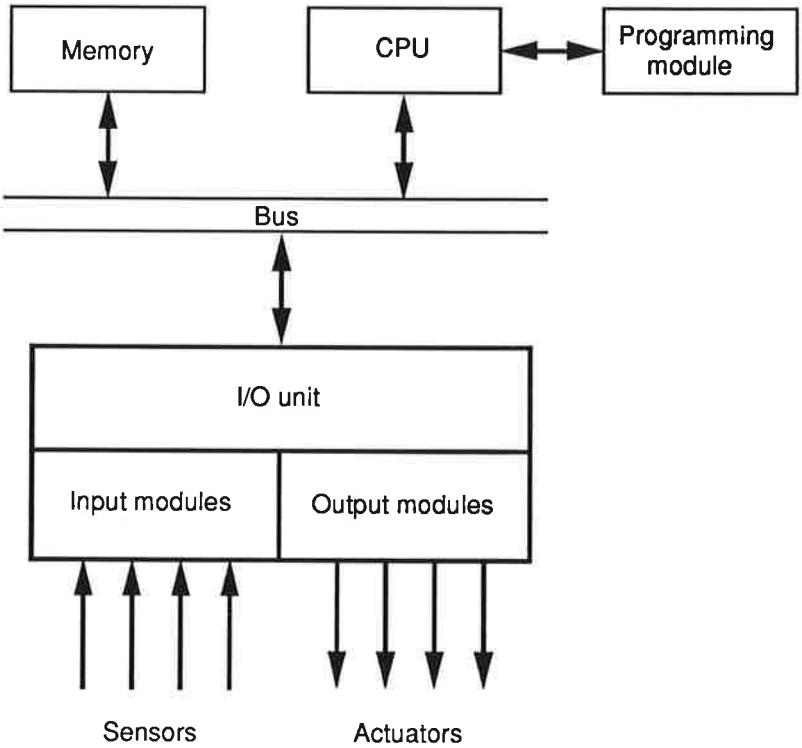

Figure 4.3 Block diagram for a PLC system.

4.3 Sequencing Controllers

The logical nets in the previous section are static systems in the sense that the same values of the input signals all the time give the same output signals. There are, however, situations where the outputs should depend on the previous history of the input signals. To describe such systems we introduce *memory* or *states*. We then obtain a *sequential net* or a *sequential process*. Sequences can either be serial or parallel. As the name indicates the serial sequences are run through one step at a time in a series. In parallel sequences we allow several things to happen in parallel with occasional synchronization between the different lines of action. A typical example of a parallel sequence is the assembly of a car. Different subparts of the car, for instance, motor and gearbox, can be assembled in parallel, but the final assembly cannot be done before all subparts are ready.

A sequence can be driven in different ways, by time or by events. A typical time driven sequence is a piece of music. Time driven sequences can be regarded as feedforward control. The chain of events is trigged by the clock and not by the state of the system. An event driven sequence is a feedback system, where the state of the system determines what will happen next.

EXAMPLE 4.3—Simple event driven sequential net

Consider the system described in Figure 4.4. It can describe a school day. It has two states 'break' and 'lesson'. When the bell rings we get a transition from one state to the other. Which output we get depends on the current state and the input signal. If starting in the state 'break' and there is no bell ringing we stay in the state break. When the bell calls we get transition to the state 'lesson'. □

A sequential net can be described by a *state graph* such as in Figure 4.4. The state graph shows the transitions and the outputs for different input signals. The sequential net can also be described by a truth table, which must include the states and also the

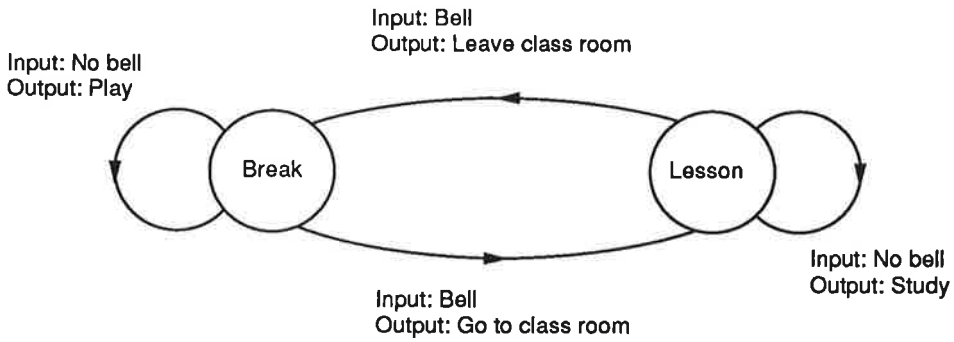

Figure 4.4 Simple sequential net for describing a school day.

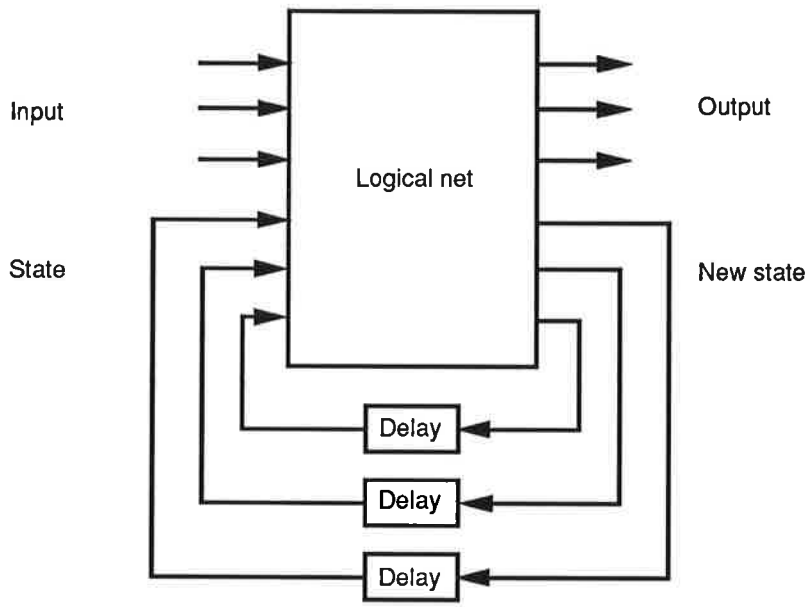

Figure 4.5 Synchronous sequential net as combination of logical net and delay or memory elements.

conditions for transitions. The sequential net is thus described by the maps

$$\text{new state} = f(\text{state, inputs})$$
$$\text{output} = g(\text{state, inputs})$$

Notice the similarity with the state equations for continuous time and discrete time systems. The difference is that the states, inputs, and outputs only can take a finite number of values. Sequential nets can be divided into *synchronous* and *asynchronous* nets. In synchronous nets a transition from one state to another is synchronized by a clock pulse, which is the case when the nets are implemented in a computer. A synchronous sequential net can be implemented as shown in Figure 4.5. In asynchronous nets the system goes from one state to the other as soon as the conditions for transition are satisfied. The asynchronous nets are more sensitive to the timing when the inputs are changing. In the sequel we will only discuss the synchronous nets.

There are many ways to describe sequences and sequential nets. A standard is now developing based on GRAFCET, developed in France. GRAFCET stands for "Graphe de Commande Etape-Transition" (Graph for Step-Transition Control). GRAFCET with minor modifications is passed as an IEC (International Electrotechnical Commission) standard, IEC 848. The way to describe sequential nets is called *function charts*. GRAFCET is a formalized way of describing sequences and functional specifications. This can be done without any consideration of how to make the hardware implementation. The functional

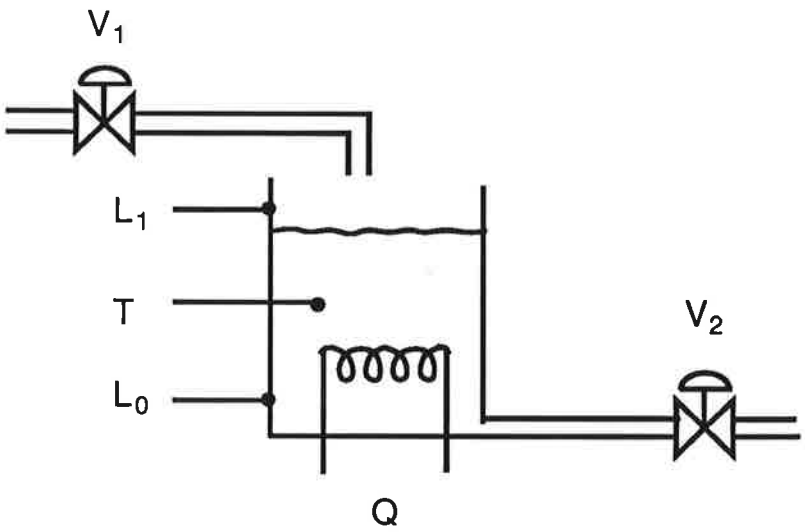

Figure 4.6 Process for batch water heating.

specifications are easy to interpret and understand. Computerized aids to program and present sequences have also been developed.

Another way to describe sequences and parallel actions is *Petri nets*. Petri nets are directed graphs that can handle sequential as well as parallel sequences. Sometimes the formalism for Petri nets makes it possible to investigate for instance reachability. It is then possible to find out which states that can be reached by legitimate transitions. This knowledge can be used to test the logic and to implement alarms.

4.4 GRAFCET

The objective of GRAFCET is to give a tool for modeling and specification of sequences. The main functions and properties of GRAFCET will be described in this section. A simple example is used to illustrate the concepts.

EXAMPLE 4.4—Heating of water
Consider the process in Figure 4.6. It consists of a water tank with two level indicators, a heater, and two valves. Assume that we want to perform the following sequence:

0. Start the sequence by pressing the button B. (Not shown in Figure 4.6.)
1. Fill water by opening the valve V_1 until the upper level L_1 is reached.
2. Heat the water until the temperature is greater than T. The heating can start as soon as the water is above the level L_0.
3. Empty the water by opening the valve V_2 until the lower level L_0 is reached.
4. Close the valves and go to Step 0 and wait for a new sequence to start.

From the natural language description we find that there is a sequence of waiting, filling, heating, and emptying. Also notice that the filling and heating must be done in parallel and then synchronized, since we don't know which will be finished first. □

GRAFCET specifications

A function chart in GRAFCET consists of *steps* and *transitions*. A step corresponds to a state and can be *inactive*, *active*, or *initial*. See Figure 4.7(a)–(c). Actions associated with a step can also be indicated, see Figure 4.7(d). A transition is denoted by a bar and a condition when the transition can take place, see Figure 4.7(e). A step is followed

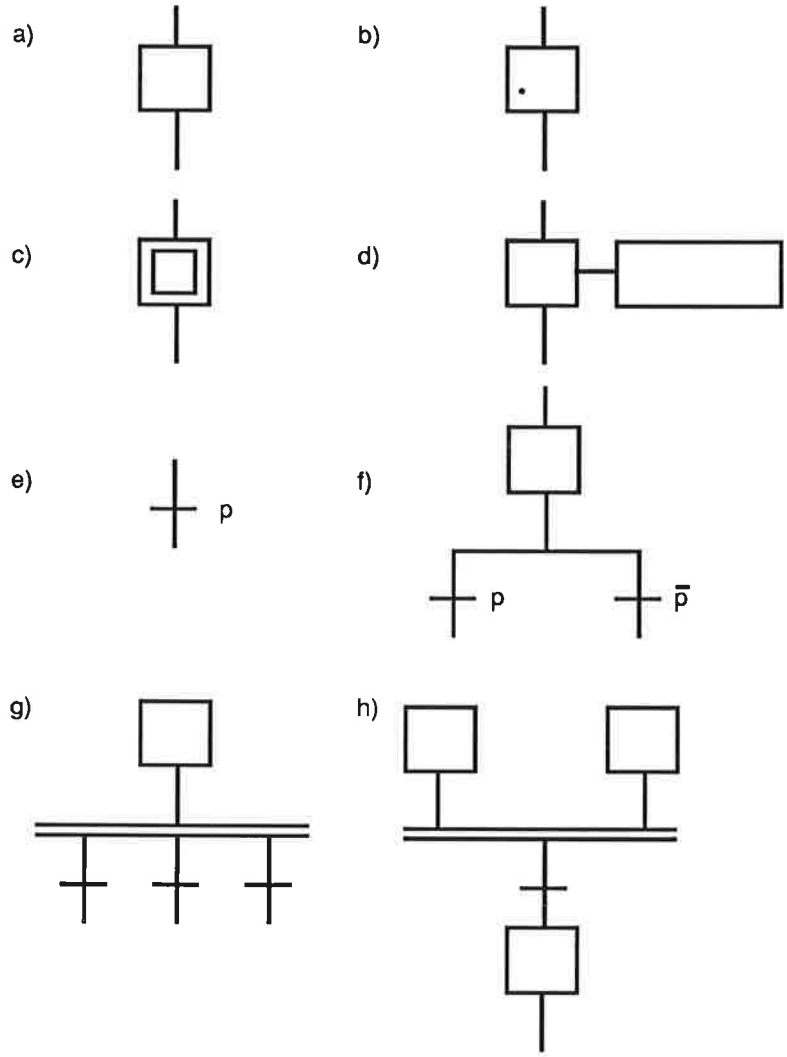

Figure 4.7 GRAFCET symbols. (a) Step (inactive); (b) Step (active); (c) Initial step; (d) Step with action; (e) Transition; (f) Branching with mutually exclusive alternatives; (g) Branching into parallel paths; (h) Synchronization.

by a transition, branching with mutually exclusive alternatives, or branching into parallel sequences. Parallel sequences can be synchronized, see Figure 4.7(h). The synchronization takes place when all the preceding steps are active and when the transition condition is fulfilled. The function chart in GRAFCET for the process in Example 4.4 is shown in Figure 4.8. The sequence starts in the step Initial. When $B = 1$ we get a transition to Fill 1, where the valve V_1 is opened until the level L_0 is reached. Now two parallel sequences starts. First the heating starts and we get a transition to Temp when the correct temperature is reached. At this stage the other branch may be finished or not and we must wait for synchronization before the sequence can be continued. In the other branch the filling continues until level L_1 is reached. After the synchronization the tank is emptied until level L_0 is reached thereafter we go back to the initial state and wait for a new sequence to start.

The example can be elaborated in different ways. For instance, it may happen that the temperature is reached before the upper level is reached. The left branch is then in step Temp. The water may, however, become too cool before the tank is full. This situation can be taken into account making it possible to jump to the step Heat if the temperature is low. In many applications we need to separate between the normal situation, and emergency situations. In emergency situations the sequence should be stopped at a

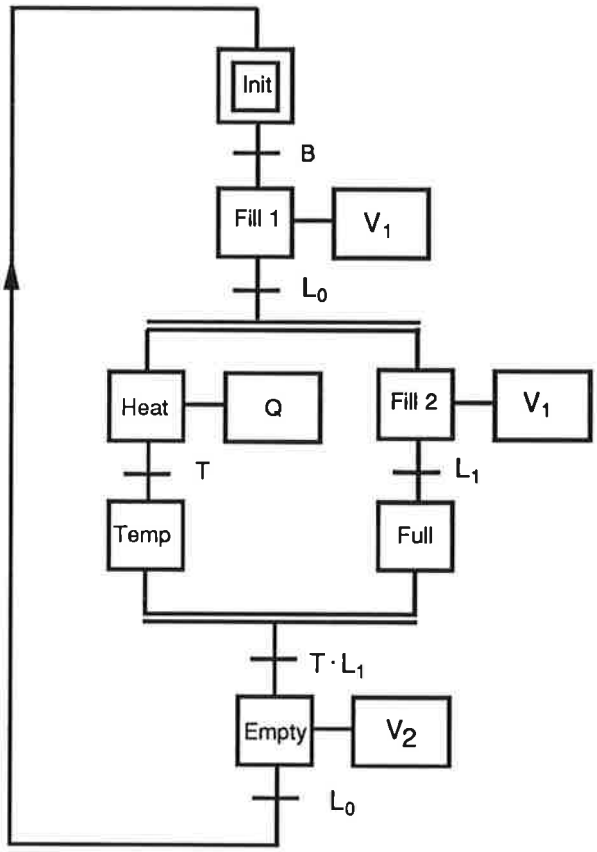

Figure 4.8 GRAFCET for the process and sequences in Example 4.4.

hazardous situation and started again when the hazard is removed. In simple sequential nets it can be possible to combine all these situations into a single function chart. To maintain simplicity and readability it is usually better to divide the system into several function charts.

GRAFCET rules

To formalize the behavior of a function chart we need a set of rules how steps are activated and deactivated etc. We have the following rules:

Rule 1: The initialization defines the active step at the start.

Rule 2: A transition is *firable* if:
- i: All steps preceding the transition are active (enabled).
- ii: The corresponding receptivity (transition condition) is true.

A firable transition must be fired.

Rule 3: All the steps preceding the transition are deactivated and all the steps following the transition are activated.

Rule 4: All firable transitions are fired simultaneously.

Rule 5: When a step must be both deactivated and activated it remains active without interrupt.

For instance, Rule 2 is illustrated in Figure 4.9. One way to facilitate the understanding of a functional specification is to introduce *macro steps.* The macro step can represent a new functional specification, see Figure 4.10. The macro steps make it natural to use a top-down approach in the construction of a sequential procedure. The overall description is first broken down into macro steps and each macro step can then be expanded. This

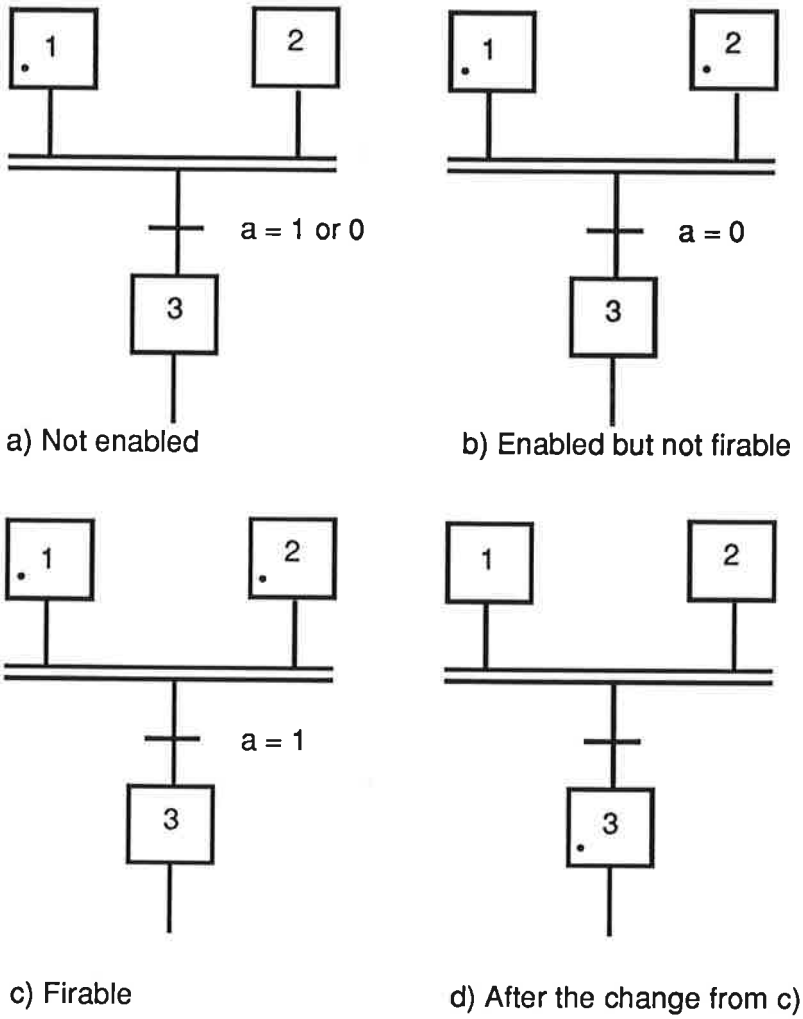

Figure 4.9 Illustration of Rule 2.

gives well structured programs and a clearer illustration of the function of a complex process.

4.5 Summary

To make alarm, supervision, and sequencing we need logical nets and sequential nets. Ways to define and simplify logical nets have been discussed. The implementation can be done using relays or special purpose computers, PLCs. GRAFCET or function charts is a systematic way of representing sequential nets that is becoming a popular industrial notation and thus important to be aware of. The basic properties of function charts are easy to understand and have been illustrated in this chapter. The analysis of sequential nets have not been discussed and we refer to the literature.

A characteristic of sequencing control is that the systems may become very large. Therefore, clarity and understandability of the description are major concerns. Boolean algebra and other classical methods are well suited for small and medium size problems. However, even though they look very simple, they have an upper size limit where the programs become very difficult to penetrate. There is currently a wish to extend this upper limit further by using even simpler and clearer notations. GRAFCET is an attempt in this direction.

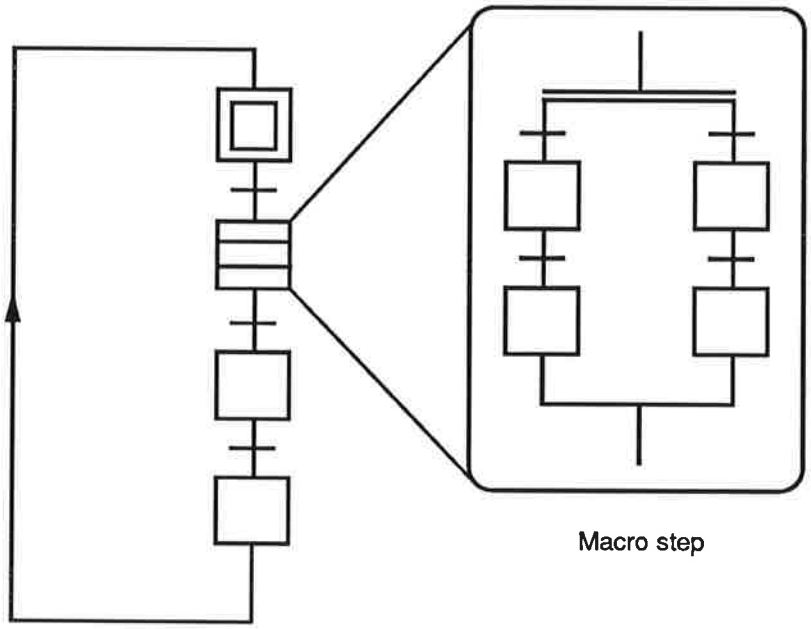

Figure 4.10 Zooming of a macro step in GRAFCET.

5

Reference Values, Trajectories, and Paths

GOAL: To introduce different aspects of reference values, which describe the desired behavior of a system.

Reference values, trajectories, and paths are different ways to describe what we expect of a system. It is obvious that we can't expect more from a system than what is physically realizable, and it is also possible to improve performance by giving well behaved reference signals to the controllers. These aspects are general but are well illustrated in robotics, and the present chapter intends to give to give some additional comments compared to a textbook in robotics, in our case Craig's book.

Before going into different methods, a comment about the needed software complexity will be made. If the user is allowed to *describe* the desired behavior of the system, then the software gets more complicated. Trajectories and paths typically may require list handling, and perhaps a graphical editor or a specific notation language for sequences of reference values. This is more complicated than having only cyclic algorithms. Advanced handling of reference values is therefore seldom used if it is not absolutely necessary, like in robotics. However, the needed software techniques will mature, and it is likely that these techniques will spread from fields like robotics to a much wider class of control systems.

5.1 Reference values

A change in a reference value should in an ideal world just be sent to the controller, which then would converge without any problems. However, we saw already in Chapter 2 that the parameters β and γ improved the performance. They were used to introduce the reference value into the control loop in a way, heuristically motivated, so that step changes have less transient effect on the behavior of the control signal. A closer analysis reveals that these parameters, β and γ, change the zeros of the closed loop transfer function from reference to output.

Model reference methods and Splines

There are different methods to describe reference values as function of time. Two methods, model reference methods and splines, will be mentioned here. Model reference methods

imply that the reference is introduced to the closed loop via a dynamic system, the reference model. The resulting reference trajectory is thus described in terms of parameters in the reference model. If both the closed loop and the reference model are computed at the same sampling rate, then the interpretation of the closed loop system is straightforward and treated in the control courses. There are for instance several ways to rewrite and structure the controller in terms of polynomials, and to make interpretations in terms of changing poles and zeros. However, there may be advantages in having a nonlinear reference model, or to have a large and computationally complicated model even if it then is necessary to compute the model at a slower sampling rate. In these cases the analysis is harder, and it is usually not natural to rewrite the controller as in linear theory, but instead to keep the structure.

Splines is another method for generation of time functions. The basic aspects are treated in Chapter 7 in Craig.

Mathematically speaking, both reference models and splines are just methods to generate functions. Nevertheless, they have different historical backgrounds. The model reference method comes from linear theory and has thus its background in descriptions of the linearized small signal behavior. Splines have been most used in applications with non-linearities e.g. with saturations on the control signal. If a double integrator (a mass) is exposed to the maximum control signal (force), then it will follow a parabola (a polynomial of 2nd degree). These historical backgrounds still influence the typical reasoning when designing the handling of the reference values. When using model reference methods one is usually concerned about poles, zeros, and e.g. what a reasonable achievable bandwidth would be. On the other hand, when choosing parameters using splines one is usually asking questions about what the available maximum torque is etc.

5.2 Trajectories and paths

Trajectories and paths are two different concepts. Trajectories describe variables as functions of time, whereas paths describe variables as functions of each other. In robotics, trajectories typically describe joint variables as functions of time, and a path may be a curve in space, e.g. a contour to be followed in a gluing application.

Path programming can be done in different ways. The spline method described in the previous section is easily extended to path programming. It is in fact a standard method in computer graphics to describe curves using splines. Another method is GRAFCET. Note that the GRAFCET-program in Figure 4.8 defines a path in T-L-space. The path is discrete and consists of the points $(T_0, L_0), (T, L_1)$. It is easy to introduce more points by introducing additional steps in Figure 4.8. A third method is the type of robot languages described in Craig in Chapters 7.7, 12, and 13. These languages are conceptually similar to GRAFCET since they define a discrete sequence of points, but they also have notations for choosing velocities and usually also built-in semantics on how the motion between the points of the path should be done. These three methods have their respective advantages. There seems so far to have been little attempt to combine the ideas. One idea that has been somewhat explored is to use convenient programming techniques based on splines, sample the path curve to obtain a discrete path, and then to transfer it to a suitable discrete description.

Paths and velocities

Assuming that we have a system supporting path programming, then it must also support notations for velocities or other ways to describe the time behavior along the path. A general problem (not limited to robotics) is then that the kinematics of the system imply that we can not define arbitrary velocities along a path. The rest of this chapter will

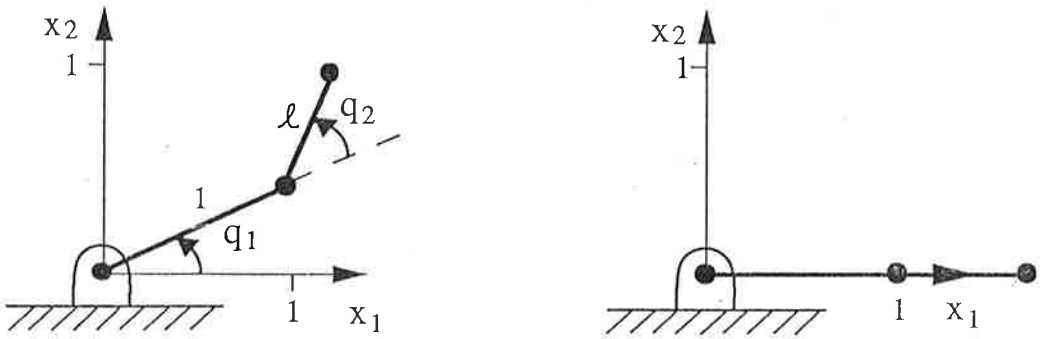

Figure 5.1 (a). A planar two link robot. (b). The two-link robot in a singular configuration. The robot can move along the x_1-axis, but there are restrictions on the possible velocities, $\dot{x}_1$. Nevertheless, it is reasonable to talk about two degrees of freedom, at least for *position* control.

illustrate these aspects as well as correct the oversimplified statements made in Craig pp. 173-174.

Kinematics

The kinematics of a system is defined as a relation between two different sets of variables.

$$x = F(q) \tag{5.1}$$

This is a general relation. One possible example, relating to Figure 4.8, would be to let q include T, whereas x instead includes e.g. a chemical reaction variable depending on T. In such an application, the chemical reaction variable would be what we really would like to control, but temperature is easier to measure and control. These variables typically have a non-linear (exponential) relation which can be written as (5.1). In robotics, we use the Cartesian representation, $x \in R^n$, as a function of the joint angles, $q \in R^m$.

The derivative of the kinematics, the Jacobian, is denoted

$$J(q) = \frac{dF(q)}{dq} \tag{5.2}$$

The Jacobian is thus the first order term in a Taylor expansion of the kinematics $F(q)$. The higher order terms usually do not vanish. The Jacobian also gives the relation between velocities e.g. in Cartesian space, $\dot{x}$, and joint rates, $\dot{q}$,

$$\dot{x} = J(q)\dot{q} \tag{5.3}$$

The singularities of the system are defined as those q, for which $J(q)$ loses rank. Degrees of freedom in the motion are thus lost in some sense. However, a closer investigation, here in the robotics case, will show that the position may be controlled arbitrarily but that there may follow requirements on the velocities. One way of anticipating such a possibility is to view the Jacobian as a first term in a Taylor expansion of the kinematics. If the first term is zero then higher order terms will, if they exist, determine the kinematic behavior.

A Familiar Kinematics Example

Consider the planar two-link, $m = n = 2$, manipulator in Figure 1 a. Without loss of generality one of the link lengths is normalized to 1. The kinematics is

$$\begin{cases} x_1 = \cos q_1 + \ell \cos(q_1 + q_2) \\ x_2 = \sin q_1 + \ell \sin(q_1 + q_2) \end{cases} \tag{5.4}$$

leading to the Jacobian, $J(q)$,

$$\begin{pmatrix} -\sin q_1 - \ell \sin(q_1 + q_2) & -\ell \sin(q_1 + q_2) \\ \cos q_1 + \ell \cos(q_1 + q_2) & \ell \cos(q_1 + q_2) \end{pmatrix} \tag{5.5}$$

The Jacobian loses rank if $\det J(q) = -\ell \sin q_2 = 0 \Leftrightarrow q_2 = n\pi$. The singular configurations are thus those where the arm is fully stretched or completely folded back.

The inverse kinematics, i.e. q as function of x, is derived using the notational conventions in Craig. Start e.g. from the Cosine-theorem to first yield q_2 as $\cos q_2$ and $\sin q_2$ from $\cos q_2 = \frac{1}{2\ell}(x_1^2 + x_2^2 - 1 - \ell^2) = c(x)$ and $\sin q_2 = \pm\sqrt{1 - c(x)^2} = s(x)$. The final result is

$$\begin{aligned} q_1 &= \mathrm{Atan2}(x_2, x_1) - \mathrm{Atan2}(\ell s(x), 1 + \ell c(x)) \\ q_2 &= \mathrm{Atan2}(s(x), c(x)) \end{aligned} \tag{5.6}$$

Consider the robot in the singular configuration where the robot is fully stretched along the x_1-axis, as seen in Figure 1 b. We will now study motions inward along the x_1-axis from this singular configuration. Such a motion is obtained if $x_2 = 0$ i.e. if the joint angles are related as:

$$\sin q_1 + \ell \sin(q_1 + q_2) = 0 \tag{5.7}$$

An approximate analysis Before developing explicit expressions for a motion along the x_1-axis, an approximate analysis based only on the leading terms in a Taylor expansion of the kinematics (5.4) is enlightening. The analysis clearly indicates how certain time functions, i.e. certain trajectories, $(x_1(t), x_2(t))$ may describe possible motions.

Close to the singularity $(x_1, x_2) = (1 + \ell, 0)$ the angles (q_1, q_2) are small. A Taylor expansion of (5.4) up to order two gives

$$\begin{cases} x_1 = 1 - q_1^2/2 + \ell\left[1 - (q_1 + q_2)^2/2\right] + O(q^4) \\ x_2 = q_1 + \ell(q_1 + q_2) + O(q^3) \end{cases} \tag{5.8}$$

Now requiring motion along the x_1-axis and neglecting higher order terms gives the constraint

$$q_1 = -\frac{\ell}{1 + \ell} q_2 \tag{5.9}$$

and

$$\begin{cases} x_1 = 1 + \ell - \dfrac{\ell}{1 + \ell} q_2^2/2 + O(q_2^4) \\ x_2 = 0 + O(q_2^3) \end{cases} \tag{5.10}$$

We thus see that it is possible to move approximately along the x_1-axis close to the singularity, since we have $\frac{\Delta x_2}{\Delta x_1} \to 0$, when $q_2 \to 0$.

The analysis so far has been without regard to velocities. Assume now motions along the x_1-axis away from the singularity

$$x_1 = 1 + \ell - p(t) \tag{5.11}$$

where $p(0) = 0$ and $p(t) \geq 0$. Note that

$$p(t) = \frac{\ell}{1 + \ell} q_2^2/2 + O(q_2^4) \tag{5.12}$$

We see that linear motion in x, i.e. $p(t) = t$ implies $q_2 \sim \sqrt{t} \Rightarrow \dot{q}_2 \sim \frac{1}{\sqrt{t}} \to \infty$ as $t \to 0$. However, quadratic motion, $p(t) = t^2$ implies $q_2 \sim t \Rightarrow \dot{q}_2 \sim 1$. The conclusion for

this particular example is that constant speed, $\dot{x}_1$, leads to infinite joint rates, but that there exist softer starts that make the motion possible. To be more formal on the latter statement: The kinematics (5.4) and the motion

$$\begin{cases} q_1 = -\dfrac{\ell}{1+\ell} \cdot t \\ q_2 = t \end{cases} \tag{5.13}$$

gives $\frac{\Delta x_2}{\Delta x_1} \to 0$ as $t \to 0$, and the derivatives are bounded.

Path following The approximate analysis gives a motion that locally starts out along the x_1-axis. We will now give a motion that really follows a path with $x_2 = 0$. We will use the inverse kinematics, eq. (5.6), with $x_2 = 0$. To give such an example, it is enough to study $q_i \in (-\pi/2, \pi/2)$ and hence to use arctan instead of Atan2. Introduce the time dependence $t = 2\tan q_2/2$, which yields $\sin q_2(t) = \frac{t}{1+t^2/4}$ and $\cos q_2(t) = \frac{1-t^2/4}{1+t^2/4}$. Let the motion be such that it is defined over a time-interval and so that the singularity $t = 0$ is included, i.e. use e.g. $t \in [0, 1]$. Introduce this into the inverse kinematics, eq. (5.6), to obtain

$$\begin{aligned} q_1(t) &= -\arctan \frac{\ell t}{1+\ell+(1-\ell)t^2/4} \\ q_2(t) &= \arctan \frac{t}{1-t^2/4} \end{aligned} \tag{5.14}$$

Note that (5.14), for small t, simplifies to (5.13). It is straightforward to verify that (5.14) has bounded derivatives and hence is a possible motion along the x_1-axis.

Summing Up

The two-link robot is a standard object of study, and it is usually claimed that the only motions possible in the singular configuration are those perpendicular to the arm. The motion (5.14) is a counterexample. The interpretation of a singularity for control purposes is thus nontrivial. The kinematics is a non-linear function and the fact that the first order term, the Jacobian, loses rank means that degrees of freedom are lost only in some sense. Usually there are higher order terms that determine the behavior. In particular, in the specific example treated here, we find it reasonable to talk about two degrees of freedom for position control, although one of these degrees of freedom cannot be extended outside the reachable space of the robot and has restrictions on the shape of the velocity profile. A wider class of motions is thus possible than at first sight. One may consider preshaping of velocity profiles, or of closed loop velocity filters activated only in directions where the Jacobian loses rank, or of other possibilities, instead of just giving up and saying that control in certain directions is impossible at a singularity.

The observations on possible motions made here must be considered when designing the man-machine interface of a control system. The results are of immediate relevance to path programming, to velocity selection, and to path following, especially if the operator is allowed to specify paths (behaviors) in the variables, x, we really would like to control.

6

The Real-Time Kernel Layer

L. Nielsen and L. Andersson

GOAL: To point out some design considerations when implementing and using a real-time kernel.

A general knowledge of Modula-2 and of principles for a real-time kernel layer is assumed. Our specific solution will therefore be briefly presented and commented. The modules presented here also include modules not directly related to real-time, but we have found it conceptually natural to group them in this basic layer of software.

6.1 Function and Implementation of the Modules

The function and implementation of the kernel is very similar to the version presented in the basic course in real-time programming. The kernel uses pre-emptive scheduling, and there is one single ready queue, where all processes ready to run are sorted in priority order. Other queues are associated with semaphores, events, and so on, where processes are waiting. Calls to the primitives, such as `Wait` result in that the process record may be moved from one queue to another. The processes are always sorted on insertion in a queue. The scheduler transfers the first process in the ready queue to running.

Some comments on the use of the primitives

One should distinguish between primitives that implement a concept and primitives that can be used to implement a concept. An example of the first type is the `Semaphores` module, which really implements the semaphore concept. An example of the second type is the module `Monitors`. The monitor concept can be implemented by programming discipline by having a call to `EnterMonitor` first in each monitor procedure and to `LeaveMonitor` last, and by using the `MonitorEvent` to implement conditional critical regions. Note that `EnterMonitor` and `LeaveMonitor` operate on a `MonitorGate`, which is conceptually a semaphore, but it has the added feature that the process occupying the monitor is raised to the priority of the highest of the processes waiting to get in.

Graphics

The module `Graphics` gives an example of the central ideas in real-time graphics. The basic data structure is `VirtualScreen` (accessed via a variable of type `handle`). A virtual screen consist of two objects; a `Window` and a `ViewPort`, both of type `rectangle`. A window is a rectangle where the x- and y-axis represent user variables e.g. physical quantities, whereas a viewport is a rectangle where the x- and y-axis represent coordinates on the computer screen. The graphics system automatically transforms coordinates between the window and the viewport coordinate systems in a virtual screen once they have been defined. The key idea is thus that the user of the module only has to think in user coordinates in the window. Read and write commands are done in window coordinates and the system automatically transforms to viewport coordinates so that the result appears on the computer screen.

Modula-2

Three short comments will be made about developing software in general in Modula-2. A module can be regarded as a language concept, not just as a fix used for separate compilation. Sometimes it may be advantageous to use internal modules in other modules to structure the code. This trend may be compared with the history of the procedure concept. In early Fortran a procedure was a separate compilation unit, but in languages following thereafter like Algol or Pascal, procedures were used in the same compilation unit, even nested etc. The second and third comments are about hiding or leaving implementation details open. The technique to hide information is to use hidden types. See the declarations in `Graphics`, `Monitors`, and `Semaphores` for examples. This means that the internal structure of the these objects are inaccessible to a user of the modules, and that the only way to operate on the objects are via the routines declared in the definition modules. A dual to hiding details is to leave them out of a module. The technique to do this in Modula-2 is to use procedure-type parameters. The general idea is to provide more universal units than if all details were included in the module. One example is hardware dependent procedures in the kernel itself. The machine dependent clock procedure is installed using a procedure-type parameter, and the kernel can be kept clean and easier to port to other computers. Another example, on a higher level, is to write a complete controller framework except for the control algorithm. The control algorithm is then installed by the user without having to change anything else in the system. This idea is extended further in Chapter 8.

Modula-2 provides coroutines and primitives like `TRANSFER` and `IOTRANSFER` to handle concurrent activities. In other languages, like for example Pascal, C, or C++, similar primitives have to be implemented. An earlier version the present kernel was done in Pascal for the LSI-11 computer. In that case, the necessary basic nucleus software for interrupt disabling/enabling, and procedures analogous to `TRANSFER` and `IOTRANSFER` consisted of four pages of assembler code. The fact that Modula-2 already contains such primitives is thus not crucial, since the work to extend other languages with similar capabilities is not overwhelming.

Portability

The real-time kernel modules are one example of a software layer. When designing such a layer one should try to find units or layers that have a long life time. The present kernel was first developed for an LSI-11 computer. It was later transferred to an IBM PC, and has recently been ported to a Sun-VME system. It has also been used on other machines outside the Department. The efforts to transfer the kernel to new machines have been limited. The major part is written in a high level language, and the machine dependent

parts can be well isolated e.g. by the use of procedure parameters as described above. The present kernel has thus survived three hardware generations.

Implementation details

Processes are declared as procedures. Such procedures should not be declared inside other procedures. Processes never terminates, so the last `END` of the procedure should never be reached. There is a minor difference between the IBM PC version and the Sun-VME version in that LONGREAL is used instead of REAL on Sun-VME.

6.2 Research topics

A current trend in the research community is to study what is called hard real-time problems. One example will be presented to give a flavor of that field. Consider the well known dining philosophers problem, which is an idealized problem in scheduling. A solution is feasible if every philosopher eventually will eat. The hard real-time version of this problem is called the dying dining philosophers problem. The new element is to consider time, and to say that a philosopher dies if he is not able to eat within certain time limits. Two main versions of the problem are if there is a waiter or not, i.e. to consider centralized or decentralized scheduling. There are also other extensions treating also two bottles of soy sauce, one curry, and so on. The research field is new and there seems to be few practical results so far, but the questions asked are relevant and one should be aware of this type of work.

6.3 The definition modules

The definition modules are listed on the following pages. One should also remember that the Modula-2 system is delivered with a number of modules e.g. mathlib for numerical functions.

The modules common both to the IBM PC system and to the Sun-VME system are: Console, Conversions, Events, Identifiers, IntConversions, Kernel, LexicalAnalyzer, Messages, Monitors, Semaphores, Strings. Note that LONGREAL is used instead of REAL on Sun-VME.

Modules specific to IBM PC are: AnalogIO, Graphics, RTGraph, RTMouse. Overlapping windows are not supported. The user must check the graphical layout.

Modules specific to Sun-VME are: AnalogIO, DigitalIO, MatComm, ServoIO, SensorBall.

`DEFINITION MODULE Console;`
Simple terminal input and output.

`PROCEDURE GetChar(VAR ch : CHAR);`
Reads one character from the Console.

`PROCEDURE GetString(VAR s: ARRAY OF CHAR);`
Reads a string, up to carriage return or line feed.

`PROCEDURE CharAvailable(): BOOLEAN;`
Returns `TRUE` IF a character is available, `FALSE` otherwise.

`PROCEDURE PutChar(ch : CHAR);`
Writes one character to the Console.

`PROCEDURE PutString(s : ARRAY OF CHAR);`
Writes a string to the Console.

`PROCEDURE PutLn;`
Writes a newline to the Console.

`PROCEDURE Trap(errortext: ARRAY OF CHAR);`
Writes a string to the Console and halts.

`END Console.`

```
DEFINITION MODULE Conversions;
```

Converts between numbers and their string representations. The routines for conversion between strings and integers or cardinals also occur separately in the module `IntConversions`. Only one of the modules `IntConversions` and `Conversions` is thus necessary.

```
PROCEDURE IntToString(VAR string: ARRAY OF CHAR;
                      num: INTEGER;
                      width: CARDINAL);
```

Converts `num` to its string representation in `string`, right justified in a field of at least `width` characters. If `string` is too small to hold the representation then it is asterisk filled instead.

```
PROCEDURE CardToString(VAR string: ARRAY OF CHAR;
                       num: CARDINAL;
                       width: CARDINAL);
```

Converts `num` to its string representation in `string`, right justified in a field of at least `width` characters. If `string` is too small to hold the representation then it is asterisk filled instead.

```
PROCEDURE StringToCard(string: ARRAY OF CHAR): CARDINAL;
```

Converts a string representation to a cardinal. Leading spaces are skipped. A leading + is allowed. If no legal cardinal can be found in the string then `MAX(CARDINAL)` is returned.

```
PROCEDURE StringToInt(string: ARRAY OF CHAR): INTEGER;
```

Converts a string representation to an integer. Leading spaces are skipped. A leading + or − is allowed and interpreted. If no legal integer can be found in the string then `-MAX(INTEGER) - 1` is returned.

```
PROCEDURE StringToReal (string: ARRAY OF CHAR): REAL;
```

Decodes a real value from an input string of characters. The syntax is permissive in the sense that the string '6' is interpreted as 6.0. Leading whitespace is permitted in the string. If no acceptable real number can be found then the value `badreal` (see below) is returned.

```
PROCEDURE RealToString(VAR string: ARRAY OF CHAR;
                       num: REAL;
      width: CARDINAL);
```

Converts a real number to a fixed point or exponent representation with `width` characters. The number is converted such that maximum accuracy is obtained. If there is enough space then a suitable fixed point representation is used. If there is not enough space then an exponent representation is used. If `width > HIGH(s)+1` then s is asterisk filled. If an exponent representation must be used and `width` is too small then the field is asterisk filled.

```
CONST badreal = 10.0E+307;

END Conversions.
```

```
DEFINITION MODULE Events;
```
Free events for the Real Time Kernel

```
TYPE
  Event;


PROCEDURE InitEvent(VAR ev: Event; name: ARRAY OF CHAR);
```
Initialize the event ev. name is for debugging purposes.

```
PROCEDURE Await(ev: Event);
```
Blocks the current process and places it in the queue associated with ev.

```
PROCEDURE Cause(ev: Event);
```
All processes that are waiting in the event queue associated with ev are unblocked. If no processes are waiting, it is a null operation.

```
END Events.
```

```
DEFINITION MODULE Identifiers;
```
Module to decode identifiers.

```
TYPE identset;
```

```
PROCEDURE NewIdentSet(VAR id: identset);
```
Initializes an ident set and returns a reference to it.

```
PROCEDURE BuildIdentSet(id: identset; name: ARRAY OF CHAR;
                        key: CARDINAL);
```
Inserts an identifier in an ident set and assigns a key to it.

`id` The ident set to be used.

`name` The identifier.

`key` The key to be associated with the identifier `name`. The value of `key` should be in the range `[2..255]`. This makes it easy to map an identset to an enumaration such as `e = (unknown, ambiguous, ...)`.

```
PROCEDURE SearchIdentSet(id: identset;
                         name: ARRAY OF CHAR): CARDINAL;
```
Searches for an identifier and returns its key if it is found and 0 otherwise.

`id` The ident set to be used.

`name` The identifier

```
PROCEDURE SearchIdentSetAbbrev(id: identset;
                         name: ARRAY OF CHAR): CARDINAL;
```
Searches for an identifier. Any nonambiguous abbreviation of the identifier is acceptable. If the identifier is found, its key is returned. If the abbreviation is ambiguous then 1 is returned and if the identifier is not found then 0 is returned.

`id` The ident set to be used.

`name` The identifier

```
PROCEDURE GetIdent(id: identset; VAR name: ARRAY OF CHAR;
                   key: CARDINAL);
```
Gets an identifier with specified key from an ident set.

`id` The ident set to be used.

`name` The identifier associated with the key `key`.

`key` The key. The value of `key` should be in `[2..255]`. See also `BuildIdentSet`.

```
END Identifiers.
```

```
DEFINITION MODULE IntConversions;
```

Conversions between strings and cardinals or integers. The routines in this module are duplicated in the module `Conversions`, that also converts between strings and reals. Only one of the modules `IntConversions` and `Conversions` is thus necessary.

```
PROCEDURE IntToString(VAR string: ARRAY OF CHAR;
                      num: INTEGER;
                      width: CARDINAL);
```

Converts `num` to its string representation in `string`, right justified in a field of at least `width` characters. If `string` is too small to hold the representation then it is asterisk filled instead.

```
PROCEDURE CardToString(VAR string: ARRAY OF CHAR;
                       num: CARDINAL;
                       width: CARDINAL);
```

Converts `num` to its string representation in `string`, right justified in a field of at least `width` characters. If `string` is too small to hold the representation then it is asterisk filled instead.

```
PROCEDURE StringToCard(string: ARRAY OF CHAR): CARDINAL;
```

Converts a string representation to a cardinal. Leading spaces are skipped. A leading + is allowed. If no legal cardinal can be found in the string then `MAX(CARDINAL)` is returned.

```
PROCEDURE StringToInt(string: ARRAY OF CHAR): INTEGER;
```

Converts a string representation to an integer. Leading spaces are skipped. A leading + or − is allowed and interpreted. If no legal integer can be found in the string then `-MAX(INTEGER) - 1` is returned.

```
END IntConversions.
```

```
DEFINITION MODULE Kernel;
```
A Real Time Kernel.

```
IMPORT KernelTypes;

TYPE
  Time = KernelTypes.Time;

CONST
  MaxPriority = MAX(CARDINAL);

PROCEDURE Init;
```
Initializes the kernel and makes a process of the main program.

```
PROCEDURE CreateProcess(processa: PROC; memReq: CARDINAL;
                        name: ARRAY OF CHAR);
```
Makes a process of the procedure `processa`. `memReq` is the number of bytes needed for local variables, stack and heap. Typical numbers are in the range 1000..10000. `name` is the name of the process for debugging purposes.

```
PROCEDURE Terminate;
```
Terminates the calling process.

```
PROCEDURE SetPriority(priority: CARDINAL);
```
The priority of the calling process is set to `priority`. High numbers mean low priority. Use numbers in the range 10..1000. Numbers higher than 1000 will cause an error halt. Numbers less than 10 may conflict with predefined internal priorities.

```
PROCEDURE CurrentTime(VAR t: Time);
```
Returns current time.

```
PROCEDURE IncTime(VAR t : Time; c: CARDINAL);
```
Increments the value of `t` with `c` milliseconds.

```
PROCEDURE CompareTime(VAR t1, t2 : Time): INTEGER;
```
This procedure compares two time-variables. Returns -1 if `t1 < t2`. Returns 0 if `t1 = t2`. Returns +1 if `t1 > t2`. The `VAR`-declaration is for efficiency only; the actual parameters are not touched.

```
PROCEDURE WaitUntil(t: Time);
```
Delays the calling process until the system time `>= t`.

```
PROCEDURE WaitTime(t: CARDINAL);
```
Delays the calling process for `t` milliseconds.

```
END Kernel.
```

`DEFINITION MODULE LexicalAnalyzer;`

The routines in this module are used to decode a string. The function `LexScan` decodes the next item in the string and returns a value indicating the type of the decoded item. A call to one of the procedures `LexCardinal` through `LexString` will then return the decoded value.

`TYPE LexHandle;`

A pointer type defined internally in the LexicalAnalyser.

```
TYPE LexTypes =
  (CardLex, CardIntLex, IntLex, RealLex, IdentLex, DelimLex,
   StringLex, EolnLex, EofLex, ErrorLex, RealErrorLex,
   StringErrorLex);
```

The possible results from `LexScan`. `RealErrorLex` corresponds to real overflow or underflow. `StringError` is given when the end of a string is missing.

`PROCEDURE LexInit(VAR lh: LexHandle);`

Initializes the internal data structures and returns a handle.

`PROCEDURE LexInput(lh: LexHandle; s: ARRAY OF CHAR);`

Makes the string `s` ready to be decoded.

`PROCEDURE LexScan(lh: LexHandle): LexTypes;`

Decodes the next item in the string which is connected with the LexHandle `lh`. The items must obey the following syntax.

⟨number⟩ :== [+|-]{⟨digit⟩}*[.{⟨digit⟩}*][⟨exponent⟩]

⟨exponent⟩ :== e|E[+|-]{⟨digit⟩}*

⟨digit⟩ :== 0| .. |9

⟨identifier⟩ :== ⟨letter⟩ {⟨letter⟩|⟨digit⟩}*

⟨letter⟩ :== a| .. |z|A| .. |Z

⟨string⟩ :== '{⟨character⟩}*'|"{⟨character⟩}*"

⟨character⟩ :== ⟨all characters defined in the ASCII table⟩

`PROCEDURE LexCardinal(lh: LexHandle): CARDINAL;`

Returns the decoded value if the result from `LexScan` is either `CardLex` or `CardIntLex`.

`PROCEDURE LexInteger(lh: LexHandle): INTEGER;`

Returns the decoded value if the result from `LexScan` is either `CardIntLex` or `IntLex`.

`PROCEDURE LexReal(lh: LexHandle): REAL;`

Returns the decoded value if the result from `LexScan` is in `CardLex .. RealLex`.

`PROCEDURE LexIdent(lh: LexHandle; VAR s: ARRAY OF CHAR);`

Returns the identifier in `s` if the result from `LexScan` is `IdentLex`. All the letters `('a'..'z')` are converted to `('A'..'Z')`.

`PROCEDURE LexDelim(lh: LexHandle): CHAR;`

Returns the delimiter if the result from `LexScan` is `DelimLex`.

`PROCEDURE LexString(lh: LexHandle; VAR s: ARRAY OF CHAR);`

Returns the delimiter if the result from `LexScan` is `DelimLex`.

`END LexicalAnalyzer.`

```
DEFINITION MODULE Messages;
```
Message Passing Routines.

```
FROM SYSTEM IMPORT ADDRESS;

TYPE
  MailBox;

PROCEDURE InitMailBox(VAR Box: MailBox; maxmessages: CARDINAL;
                      name: ARRAY OF CHAR);
```
Initializes Box. The maximum number of messages that the box can contain is maxmessages.

```
PROCEDURE SendMessage(Box: MailBox; VAR MessAdr: ADDRESS);
```
Sends the message referenced by MessAdr to Box. If the mailbox already contains the maximum number of messages then the calling process will wait. On return MessAdr = NIL.

```
PROCEDURE ReceiveMessage(Box: MailBox; VAR MessAdr: ADDRESS);
```
Receives a message from Box. The calling process is delayed if Box is empty.

```
PROCEDURE AcceptMessage(Box : MailBox; VAR MessAdr : ADDRESS);
```
Receives a message from Box. If there is a message in the box then MessAdr points to the message. If there is no message in the box then MessAdr = NIL. AcceptMessage does not delay the calling process.

```
END Messages.
```

```
DEFINITION MODULE Monitors;

TYPE MonitorGate;
TYPE MonitorEvent;

PROCEDURE Init;
```

Initializes the Monitors module.

```
PROCEDURE InitMonitor(VAR mon: MonitorGate;
                      name: ARRAY OF CHAR);
```

Initializes the monitor guarded by mon. name is for debugging purposes.

```
PROCEDURE EnterMonitor(mon: MonitorGate);
```

Try to enter the monitor mon. If no other process is within mon then mark the monitor as busy and continue. If the monitor is busy, then block the calling process in a priority queue AND raise the priority of the blocking process to the priority of the blocked process.

```
PROCEDURE LeaveMonitor(mon: MonitorGate);
```

Leave the monitor mon. If the priority was raised then lower it to the original value. If there is one or more processes waiting, then unblock the first one in the queue, else mark the monitor as not busy.

```
PROCEDURE InitEvent(VAR ev: MonitorEvent; mon: MonitorGate;
                    name: ARRAY OF CHAR);
```

Initialize the event ev and associate it with the monitor mon. name is for debugging purposes.

```
PROCEDURE Await(ev: MonitorEvent);
```

Blocks the current process and places it in the queue associated with ev. Also performs an implicit LeaveMonitor(mon).

```
PROCEDURE Cause(ev: MonitorEvent);
```

All processes that are waiting in the event queue associated with ev are moved to the monitor queue associated with mon. If no processes are waiting, it is a null operation.

```
END Monitors.
```

`DEFINITION MODULE Semaphores;`

Semaphores for the Real Time Kernel. Note that `Kernel.init` must be called before any of these procedures.

```
TYPE Semaphore;
```

```
PROCEDURE InitSem(VAR sem: Semaphore; InitVal: INTEGER;
                  name: ARRAY OF CHAR);
```

Initializes the semaphore `sem` to `InitVal`. `name` is for debugging purposes.

```
PROCEDURE Wait(sem: Semaphore);
```

If the value of the semaphore `Sem` > 0 then decrement it, else block the calling process. If more than one process is waiting, then queue them first in priority and then in FIFO order.

```
PROCEDURE Signal(sem: Semaphore);
```

If there is one or more processes waiting, then unblock the first one in the queue, else increment the semaphore.

`END Semaphores.`

```
DEFINITION MODULE Strings;
```

String handling routines. For all these routines the general principle is that if a target string is too short, then the result is silently truncated. There are no run time errors.

```
PROCEDURE Length(str: ARRAY OF CHAR): CARDINAL;
```

Returns the number of characters in `str`.

```
PROCEDURE Compare(astring, bstring: ARRAY OF CHAR): INTEGER;
```

Compares `astring` and `bstring` Returns `-1`, `0` or `+1` indicating less than, equal or greater than.

```
PROCEDURE Position(pattern, source: ARRAY OF CHAR;
                   start: CARDINAL): CARDINAL;
```

Returns the position of `pattern` within `source`. The comparison starts at position `start`

```
PROCEDURE ReversePosition(pattern, source: ARRAY OF CHAR;
                          last: CARDINAL): CARDINAL;
```

Returns the position of the *last* occurence of `pattern` within `source`. The comparison starts `last` characters from the end and proceeds backwards.

```
PROCEDURE Assign(VAR target: ARRAY OF CHAR;
                 source: ARRAY OF CHAR);
```

Assigns `source` to `target`

```
PROCEDURE Insert(VAR target: ARRAY OF CHAR;
                 string: ARRAY OF CHAR;
                 pos: CARDINAL);
```

Inserts `string` into target at position `pos`

```
PROCEDURE Substring(VAR dest: ARRAY OF CHAR;
                    source: ARRAY OF CHAR;
                    index, len: CARDINAL);
```

Returns in `dest` a substring from `source` starting at `index` and containing `len` characters.

```
PROCEDURE Append(VAR target: ARRAY OF CHAR;
                 string: ARRAY OF CHAR);
```

Appends `string` to `target`.

```
PROCEDURE AppendC(VAR target: ARRAY OF CHAR; c: CHAR);
```

Appends the character `c` to `target`

```
PROCEDURE Delete(VAR target: ARRAY OF CHAR;
                 index, len: CARDINAL);
```

Deletes `len` characters FROM `target`, starting at position `index`

```
END Strings.
```

`DEFINITION MODULE AnalogIO;`
Analog input/output.

`PROCEDURE ADIn(Channel : CARDINAL) : REAL;`
Returns a value in the interval [-1.0..1.0], corresponding to [-10.0 V..10.0 V], from channel number `Channel`. Allowed channel numbers are 0 through 3 for IBM and 0 through 7 FOR Tandon.

`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 FOR IBM and 0 through 3 for Tandon.

`END AnalogIO.`

```
DEFINITION MODULE Graphics;
```

Warning Do not use this module together with the module Terminal.

```
TYPE
  handle;
```

A pointer type defined internally in the graphics system

```
  point = RECORD
    h: REAL;     Horizontal coordinate
    v: REAL;     Vertical coordinate
  END;

  rectangle = RECORD
    CASE BOOLEAN OF
    TRUE:
       loleft: point;                Lower left corner
       upright: point;|              Upper right corner
    FALSE: xlo,ylo,xhi,yhi: REAL;    Alternate representation
    END;
  END;

  color=(black, blue, green, cyan, red, magenta, brown, white,
         grey, lightblue, lightgreen, lightcyan, lightred,
         lightmagenta, yellow, intensewhite);
  buttontype=(LeftButton, RightButton);   For the mouse buttons
  buttonset = SET OF buttontype;
```

```
PROCEDURE VirtualScreen(VAR h: handle);
```

Initializes the data structures for a virtual screen and returns a handle. The default window and viewport are `(0.0 < x < 1.5)` and `(0.0 < y < 1.0)`. The default color for line and text is white; for fill it is black.

```
PROCEDURE SetWindow(h: handle; r: rectangle);
```

Defines the window coordinates.

`h` The virtual screen handle.

`r` The rectangle specifying the window. All real numbers are permitted as window coordinates.

```
PROCEDURE SetViewPort(h: handle; r: rectangle);
```

Positions the viewport on the screen.

`h` The virtual screen handle.

`r` The rectangle specifying the window.

The viewport rectangle must satisfy the screen limits `(0.0 < x < 1.5)` and `(0.0 < y < 1.0)`.

```
PROCEDURE Shutdown;
```

Closes down the entire graphics system and sets the screen in normal text mode.

```
PROCEDURE SetLineColor(h: handle; c: color);

PROCEDURE SetTextColor(h: handle; c: color);
```

```
PROCEDURE SetFillColor(h: handle; c: color);
```

```
PROCEDURE PolyLine(h: handle; VAR polygon: ARRAY OF point;
                   npoint: INTEGER);
```

Draws a polygon.

`h` The virtual screen handle

`polygon` The points of the polygon. The start point is `polygon[0]`. Lines are drawn using the line color of the specified handle. The VAR declaration is for efficiency only, and the actual argument is not changed.

`npoint` The number of points in polygon.

```
PROCEDURE PolyMarker(h: handle; VAR polygon: ARRAY OF point;
                     npoint: INTEGER);
```

Draws markers at the specified points. The markers are plus signs at present.

`h` The virtual screen handle

`polygon` The points where the markers are drawn. The current line color is used. The VAR declaration is for efficiency only, and the actual argument is not changed.

`npoint` The number of points in polygon.

```
PROCEDURE WriteString(h: handle; p :point; s: ARRAY OF CHAR);
```

Writes a string on the screen starting at a specified point.

`h` The virtual screen handle

`p` The starting point of the text.

`s` The text string to be written. If s contains CHR(0) then this is considered the end of the string.

The text is written with the text color of the specified handle. Old text is overwritten, not erased. See `EraseChar`.

```
PROCEDURE FillRectangle(h: handle; r: rectangle);
```

Fills a rectangle on the screen.

`h` The virtual screen handle.

`r` The rectangle to be filled.

The rectangle is filled with the fill color of the specified handle.

```
PROCEDURE DrawRectangle(h: handle; r: rectangle);
```

Draws a rectangle on the screen.

`h` The virtual screen handle.

`r` The rectangle to be drawn.

The rectangle is drawn with the line color of the specified handle.

```
PROCEDURE ReadString(h: handle; p :point; VAR s: ARRAY OF CHAR);
```

Reads a string from the keyboard up to Enter, with echoing.

`h` The virtual screen handle.

`p` The point where echoing starts.

`s` The returned string. If it is less than `HIGH(s)` characters long then the string is delimited by `CHR(0)`.

The characters are echoed with the text color of the specified handle. The text background is the fill color of the specified handle.

`PROCEDURE CharacterSize(h: handle; VAR width, height: REAL);`
Returns the size of a character in the current window coordinate.

- `h` The virtual screen handle.
- `width` The horizontal size of a character.
- `height` The vertical size, i.e. the distance between the baselines of two adjacent text lines.

`PROCEDURE EraseChar(h: handle; p: point; num: CARDINAL);`
Erases characters, i.e. fills the area with the fill color.

- `h` The virtual screen handle.
- `p` The starting point of the erase.
- `num` A region corresponding to num characters is filled with the fill color.

`PROCEDURE GetMouse(h: handle; VAR p:point; VAR b:buttonset);`
Returns the mouse state.

- `h` The virtual screen handle.
- `p` The mouse position.
- `b` The button state. If `LeftButton IN b` then this button is pressed, and conversely for the right button.

`PROCEDURE WaitForMouse(h: handle; VAR p: point; VAR b: buttonset);`
Waits until at least one of the buttons get pushed, then returns the mouse state. In a real time situation it is not a busy wait.

- `h` The virtual screen handle.
- `p` The mouse position.
- `b` The button state. If `LeftButton IN b` then this button is pressed, and conversely for the right button.

`PROCEDURE SetMouseRectangle(h: handle; r: rectangle; n: CARDINAL);`
Inserts a rectangle in a list of rectangles to be tested in a `WaitMouseRectangle` or `GetMouseRectangle` operation. There is one list for each handle.

- `h` The virtual screen handle.
- `r` The rectangle to be inserted.
- `n` The number to be returned if the mouse is inside `rr`.

If the specified number `n` already exists in the list then no new entry is made, but the old entry gets a new rectangle value. Do not use the number 0, because 0 has a special meaning for `GetMouseRectangle`. No test is made, however. This routine does not draw anything. The drawing should be done with `DrawRectangle`.

`PROCEDURE WaitMouseRectangle(h: handle): CARDINAL;`
Waits until a mouse button is pressed and the cursor is inside one of the rectangles previously specified with `SetMouseRectangle`. The number associated with that rectangle is then returned. The entries are kept and tested in number order, and the first match found is returned. The cursor hot spot must be strictly inside the rectangle if it is to be considered a match.

`PROCEDURE GetMouseRectangle(h: handle): CARDINAL;`
If the cursor hot spot is strictly inside one of the mouse rectangles, then the corresponding number is returned, otherwise 0 is returned. See `SetMouseRectangle` and `WaitMouseRectangle`.

`PROCEDURE HideCursor;`

The internal hide/show counter is decremented. If the counter is zero after decrementation then the cursor is removed from the screen.

`PROCEDURE ShowCursor;`

The internal hide/show counter is incremented. If the counter is 1 after incrementation then the cursor is shown on the screen.

The following procedure types and procedures are used to define which modules should handle the mouse and the keyboard. In a real time application the modules RTGraph and RTMouse will call these procedures. They should not be referenced directly by user programs.

```
TYPE
  EchoStringProc=PROCEDURE(VAR ARRAY OF CHAR);
  MouseProcedureType=PROCEDURE(VAR buttonset, VAR INTEGER,
                               VAR INTEGER);

PROCEDURE SetEchoString(p: EchoStringProc);

PROCEDURE SetMouseProcedures(GM,WM: MouseProcedureType;
                             HC, SC: PROC);

END Graphics.
```

`DEFINITION MODULE RTGraph;`

Establishes connections between the Kernel and the graphic modules.

`PROCEDURE Init;`

Initialization. There is an implicit call to `Kernel.Init`.

`END RTGraph.`

`DEFINITION MODULE RTMouse;`

Establishes connections between mouse, Kernel and the graphic modules.

`PROCEDURE Init;`

Initialization. There is an implicit call to `RTGraph.Init` and thus to `Kernel.Init`.

`END RTMouse.`

```
DEFINITION MODULE AnalogIO;

(* Analog IO via the VDAD boards from PEP computers.

   Initialize the VDADs in one and only one of the following ways:
   1. Call InitServoIO in the module ServoIO.
   2. Call InitResolver in the module ResolverIO.
   3. Import VDAD, call InitVDAD, and set up the control registers
      on the board. Refer to VDAD reference manual for details
   Alternatives 1 and 2 configures the analog ports to operate in
   the range +/- 10V.
*)

FROM VDAD IMPORT NrOfCards;

TYPE CardType     = [1..NrOfCards];
     ChannelType  = [0..15];
     DARange      = [-2048..2047];
     IntArray     = ARRAY ChannelType OF INTEGER;
     ExpGainType  = [0..2];
     OutRangeType = [0..1];

PROCEDURE ADin  (     cardnr  : CardType;
                      channel : ChannelType;
                  VAR value   : INTEGER      );

PROCEDURE DAout (     cardnr  : CardType;
                      channel : ChannelType;
                      value   : DARange      );

PROCEDURE MultiADin (     cardnr      : CardType;
                          LowChannel,
                          HighChannel : ChannelType;
                      VAR value       : IntArray     );

PROCEDURE SetInputGain( cardnr  : CardType;
                        ExpGain : ExpGainType );
  (* ExpGain = 0, 1, 2  =>  Input gain = 1, 10, 100 *)

PROCEDURE SetOutVoltage( cardnr     : CardType;
                         channel    : ChannelType;
                         Gain,
                         UniBiPolar : OutRangeType );
  (* Outvoltage range = Vref * (1 + Gain)  *)
  (* UniBiPolar: 0 = unipolar, 1 = bipolar *)

END AnalogIO.
```

```
DEFINITION MODULE DigitalIO;

(* Digital IO via VDAD boards and the VDIN board from PEP computers.

   Initialize the VDADs in one and only one of the following ways:
   1. Call InitServoIO in the module ServoIO.
   2. Call InitResolver in the module ResolverIO.
   3. Import VDAD, call InitVDAD, and set up the control registers
      on the board. Refer to VDAD reference manual for details
   Alternatives 1 and 2 configures the digital ports to be output
   on board number one, and input on board number two.

   The VDIN board requires no initialization. To read the 16 bit
   parallell input port, call DigInput.
*)

FROM VDAD   IMPORT NrOfCards;
FROM SYSTEM IMPORT BYTE;

TYPE CardType = [1..NrOfCards];
     WireType = [0..7];
     Bit      = [0..1];
     Byte     = [0..255];

PROCEDURE DigWireOutput (     cardnr : CardType;
                              wire   : WireType;
                              value  : Bit      );

PROCEDURE DigWireInput  (     cardnr : CardType;
                              wire   : WireType;
                          VAR value  : CARDINAL );

PROCEDURE DigByteOutput (     cardnr : CardType;
                              value  : BYTE     );

PROCEDURE DigByteInput  (     cardnr : CardType;
                          VAR value  : BYTE );

PROCEDURE DigInput( VAR value : SHORTINT );
  (* value = [-32768, 32767] *)

END DigitalIO.
```

```
DEFINITION MODULE MatComm;

IMPORT SYSTEM;
(*$NONSTANDARD*)

CONST
  ProcessNameSignificance = 32;

TYPE
  Socket;
  NameString = ARRAY [1..ProcessNameSignificance] OF CHAR;
  DataType  = (notype,char,real,longreal,complex,longcomplex);
  ErrorType = (OK,notOK,Closed);


PROCEDURE OpenSocket(
  VAR sock      : Socket;
  REF myname    : ARRAY OF CHAR
  );
  (*
  A socket is returned to be used in subsequent calls to the procedures
  below. The 'myname' is required to make the request for a socket unique.
  It can be any string, but is typically the name of the process. The same
  string has to be given as the proc-argument to vmeio in matlab. It does
  not matter if the modula process or the unix process (i.e. matlab) is
  the first one to try to establish a new connection (after Init has been
  called).
  *)

PROCEDURE CloseSocket(
  VAR sock      : Socket
  );
  (*
  The socket is closed by the caller for further communication. The line
  can also be closed by the remote machine. In both cases, a Send or
  Receive request will return 'Closed'. If so, OpenSocket can be called
  again.
  *)

PROCEDURE Send(
  VAR sock : Socket;
  nrows,
  ncols     : CARDINAL;
  dtype     : DataType;
  REF data : ARRAY OF SYSTEM.BYTE
  ): ErrorType;
  (*
  Send the 'data' on the open socket 'sock'. Proper values for 'nrows',
  'ncols', and 'dtype' have to be supplied (if you don't want a dump of
  the memory following the variable supplied). 'data' is however
  allowed to be bigger than the matrix specified.
  *)
```

```
PROCEDURE GetNextType(
  VAR sock      : Socket;
  VAR nrows,
      ncols : CARDINAL;
  VAR dtype : DataType
  ): ErrorType;
  (*
  If a new data message is available, the head of the message is read and
  the type of the matrix is returned in nrown, ncols, and dtype. To
  prevent reading the head again in an additional call (without
  Receive in between), and to save some computations, some extra
  information is stored in 'sock'. To allow update this privat
  information, 'sock' is also VAR declared. If no data is available,
  an Await for data on 'sock' is performed. If data to be received is
  of fixed type (or size), Receive can be called directly.
  *)

PROCEDURE Receive(
  VAR sock  : Socket;
  VAR nrows,
      ncols : CARDINAL;
  VAR dtype : DataType;
  VAR data  : ARRAY OF SYSTEM.BYTE
  ): ErrorType;
  (*
  If not done already for the next message, GetNextType is called.
  This means 'nrows', 'ncols', and 'dtype' will be the same as if
  GetNextType were called. The 'data'-matrix is allowed to be bigger
  then required to store the data. If the 'data'-matrix is to small,
  as mutch as possible is stored in 'data'. The rest is read in and
  then deallocated. In this case 'notOK' is returned.
  *)

PROCEDURE Receive2(
  VAR sock  : Socket;
  VAR nrows,
      ncols : CARDINAL;
  VAR dtype : DataType;
  VAR data  : ARRAY OF SYSTEM.BYTE;
  dsize     : CARDINAL
  ): ErrorType;
  (*
  As Receive, but the size of the data has to be explicitly given.
  To be used for dynamic variables with size unknown at compile time.
  *)

PROCEDURE Init;
  (* Includes creation of process 'operator' handling incoming calls *)

END MatComm.
```

```
DEFINITION MODULE ServoIO;

(* Interface module to drive units and position sensors of the modified
   ASEA IRB-6 system, and joint compatible with that system.

   Upon init (by calling InitServoIO) the following happens:
   1. The scaling of RefOut is set to either MaxVel or MaxTorque
      depending on mode of operation. With the drives in PI or
      P-servo mode, the reference is a velocity reference, and
      InitServoIO should be called with MaxRef=MaxVel. With the
      drives in Ext mode, the reference is a torque reference, and
      InitServoIO should be called with MaxRef=MaxTorque.
   2. Analog or digital position sensor interface is selected by
      assigning the proper procedures to the procedure variables
      below. This way, the same procedures (i.e. the procedure
      variables absPos, incPos etc.) can be called for both analog
      and digital interface, and without any test of mode in the
      implementation module.
   3. The motor revolution counter in the resolver interface hardware is
      reset to zero.
   4. The internal counter in this module used for keeping track of
      the resolver-interface "turns" is reset to zero.
   The counters can also be reset by calling ResetServo. The zero position
   is then defined to be at motor angle zero IN THE CURRENT MOTOR TURN.
   This is needed for syncronization of the position measurement system.

   The incremental position values can be used to get the speed, or simply
   to reduce the magnitude of signals by using an incremental controller.

   The procedures with names containing a '5' can be used for fast
   sampling of joints 1..5. All values are then sampled at exactly the
   same time.

   To fully use this module, two VDAD boards and one VDIN board (PEP
   boards) is required. With fewer boards, be careful not to address
   a non-exsisting board, which will result in a bus-error exception.
*)

CONST MaxVel = 314.15; (* rad/s => 3000 rpm *)
      MaxTorque = 1.3; (* Nm => 15 A motor current. *)

TYPE JointType      = [0..7];   (* 0 for external joint. 1..5 for IRB-6 *)
     AnalogOutputs  = [0..7];
     Array5Real     = ARRAY [1..5] OF REAL;
     Array5Longreal = ARRAY [1..5] OF LONGREAL;
     ReadMode       = ( analog, digital );
     absPosProc     = PROCEDURE ( JointType ) : LONGREAL;
     incPosProc     = PROCEDURE ( JointType ) : REAL;
     abs5PosProc    = PROCEDURE ()            : Array5Longreal;
     inc5PosProc    = PROCEDURE ()            : Array5Real;
     ZeroPosProc    = PROCEDURE ( JointType );
```

```
VAR absPos     : absPosProc  ;  (* absolute motorangle in radians       *)
    incPos     : incPosProc  ;  (* incremental motorangle in radians    *)
    abs5Pos    : abs5PosProc ;  (* 5 absolute motorangles in radians    *)
    inc5Pos    : inc5PosProc ;  (* 5 incremental motorangles in radians *)
    ResetServo : ZeroPosProc ;  (* resets the motorturn counter         *)


PROCEDURE Init( AnOrDig : ReadMode);
  (* See description above. Also initializes RefScale for all joints to
     MaxVel.                                                            *)


PROCEDURE RefScale( joint : JointType; MaxRefOut : LONGREAL );


PROCEDURE RefOut ( joint : AnalogOutputs;  value : LONGREAL );
  (* Output value is limited to [-MaxRefOut, MaxRefOut].
     For joints not present, RefOut can be used as AnalogOut with
     built in scaling and saturation.                                   *)


PROCEDURE ZeroSense ( joint : JointType ) : CARDINAL;
  (* Returns a value in the range [0,1] depending on the state of the
     syncronization switches on the robot. 1 means switch is closed. *)


END ServoIO.
```

```
DEFINITION MODULE SensorBall;
(*
   Handler for the six DOF joystick, the so called sensor ball.

   The integers returned are in the range [-128,+127]. The Button
   integer has one bit set for each button pressed according to:
   0 => no buttons, 1 => button 1,  2 => button 2, 4 => button 3,
   7 => buttons 1,2,3, etc.

   The communication goes via an interrupt driven RS232 communication
   port, set to a baud rate of 9600.
*)

TYPE PositionDataType = RECORD
                              Xtransl,
                              Ytransl,
                              Ztransl,
                              Xrot   ,
                              Yrot   ,
                              Zrot   : INTEGER;
                           END;

PROCEDURE InitSensorBall;
(*
   Initiate the sensor ball, the serial communication, and data buffers.
   Warning: Several interrupts with high hardware priority is serviced
   internally in the module during init. Software processes even with
   high software priority might therefore be delayed. This means you
   should not call this procedure during very critical control tasks
   with high CPU load and fast sampling (e.g. > 200 Hz)
*)

PROCEDURE ReadSensorBall ( VAR PositionData : PositionDataType;
                           VAR Buttons      : INTEGER;
                           VAR Error        : CARDINAL          );
(*
   Sends a read request to the sensor ball, awaits interrupt, receives
   data on interrupt, and returns joystick and key data. Error = 0
   indicates no error. Error > 0 usually indicates timeout due to not
   connected cables. A suitable rate of reading is 10 Hz
*)

END SensorBall.
```

7

Real-Time Graphics Support Modules

P. Persson and L. Nielsen

GOAL: To give ideas and principles for support primitives in a layer on top of the real-time kernel layer.

An implementation of a control system almost always include a number of components regarding interaction, such as presenting values, plotting signals, creating signals, and others. Since the functions needed are similar between different applications, it is possible to provide a set of support modules to simplify the use of the basic real-time and graphics primitives, like those in the previous chapter. The advantage of having such support modules can be quantified by comparing the example given in Section 7.9 with a similar program implemented using only the real-time kernel layer. The code size is reduced four times, and the work to develop the program is reduced considerably more. The function and implementation of the library modules are described in Sections 7.1-8. An example of the use of the routines in a real-time program is given in Section 7.9, and the definition modules of the routines are listed thereafter.

Main principles

It is mandatory to realize that a man-machine interface consist of several parallel activities. For example, an emergency stop has to be active even when the operator is in the middle of parameter editing. It is also necessary to be able to synchronize actions e.g. when changing several parameters in a controller. Support modules for a man-machine interface have to provide such real-time possibilities. A simple approach that regards the computer screen as one single object is therefore not sufficient. Instead, the screen should be viewed as consisting of different areas divided between different parallel processes. A variety of functions are needed to achieve this, and this chapter presents one possible set of modules.

It is an advantage to have a layered software structure, where functions from all layers are available. Consider a case where the support modules of this chapter are used for plotting, parameter interaction and so on. Further, there is perhaps a need, in the man-machine interface, for a button having a special function requiring a specially tailored semantics of that button. There are no problems in combining the support modules presented here with such special solutions implemented using the module `Graphics` in the

previous chapter.

7.1 Function and Implementation of the Modules

Modules for handling of lists, signal generation, bargraphs, plot windows, screen buttons, numerical menus, and logical menus are available. All objects created by these modules should be considered as *monitors,* i.e., they contain data and procedures and guarantee mutual exclusion when used from different processes. When the routines of these modules are used there is thus no need for semaphore protection of objects created by the use of the routines. The system is written in Logitech's Modula-2 and is intended to be used on IBM-AT compatible machines with EGA and Microsoft Mouse. The real-time kernel layer described in the previous chapter is used for the implementation.

It is necessary to understand the concurrency characteristics of these modules when using them. The following example is typical. A common need in real-time applications in control engineering is to have a controller with several parameters and to be able to change all parameters of the controller at the same time. One correct way to handle this is to use a *numerical menu*, see Section 7.7. When all input to the menu is done, the user clicks in the `Done` rectangle and the new values are made available to the controller. There is no way of changing the values between the click and the time when they are available to the controller. One solution, which is wrong in principle, would be to let the monitor consist of a number of bargraphs and a button, and to let a process wait for a click in the button, and when a click is detected read the values of the bargraphs to the controller. The principle error is that since all bargraphs and buttons work *in parallel*, then it possible that other processes could be activated when only some of the controller parameters have been changed. It would even be possible to click in a bargraph after the click in the button and before the value of the bargraph had been read, which is also wrong in principle.

Implementation details

Processes and other features from the Modula-2 real-time system are used in all these modules. Therefore a call to `RTMouse.Init` must be made before any of the modules described here are initiated. The priority of the internal processes must be given in the initialization procedure of the modules. All coordinates for windows, menus etc. are given in screen coordinates, i.e., $0 \leq x \leq 1.5$ and $0 \leq y \leq 1.0$. There may be problems in changing the attributes (e.g. color, scales etc.) of the objects once the attributes have been set. Therefore, regard the first call to a routine for setting attributes as a declaration of the object, and do not try to change attributes once they have been set. There are also some slight inconsistencies between the present modules in the calling conventions of the creation of squares on the screen and in the way to hide windows. Read the definition modules carefully.

Opaque (hidden) data types have been used to implement buttons, bargraphs, lists, menus, and plot windows. This means that the internal structure of these objects are inaccessible to a user of the modules. The only way to operate on the objects are via the routines declared in the definition modules. This is the Modula-2 way to implement abstract data types.

7.2 List Handler

The `ListHandler` module handles a doubly linked non-circular list with a list head. The list handling is done so that it is independent of the type of the element that is stored in the nodes. This is achieved by using the data type `ADDRESS` in the nodes. This also

means that the elements must be referred via a pointer, see the example in the definition module. The list handling package is used in all the other modules.

7.3 Signals

The routines in the `Signals` module are used for generating time signals. Any number of signals can be generated, and each signal is identified by a text string. The signals which can be generated are of the types `Sin`, `Step`, `Pulse`, `Ramp`, and `Random`. Signals of different types can be generated at the same time. The user of this module does not have to handle time explicitly. When `GetRefValue` is called the value of the signal at that time is returned.

The implementation is simple. A new signal instance is created with `MakeRefSignal`. All signal instances are represented as nodes in a linked list. The signals are generated with the time interval `Delta`, by a loop in the process `Generator`, which is an process internal to the module. The value of all signals are in the interval [0 1]. There is no way of changing the amplitude and offset of the signal within the module, so that must be taken care of by the user program.

7.4 Bargraphs

The module `BarGraph` has facilities for making bargraphs. Any number of bargraphs can be created. It is possible to use a bargraph as an input device where new values can be input to the bargraph by mouse clicks in the bargraph or by numerical input from the keyboard. Only one bargraph at the time can handle numerical input. It is also possible to use a bargraph as an output device to display a variable. Alarm limits can be set in the bargraph so that the bargraph changes color when it reaches certain limits. If `HideBarGraph` is called while numeric input from the keyboard is active then the bargraph will be hidden *after* the numeric input has been completed.

The implementation consists of two processes. One process waits for a mouse click in one of the bargraphs, this process then sends a message to another internal process which interprets the messages and changes the bargraphs.

7.5 Plots

The `Plot` module can create plot windows for plotting signals as functions of time. Any number of plot windows can be created. In each plot window a number of signals can be plotted. The signals to be plotted can be chosen via a menu on the frame of the plot window. The user does not have to take care of time explicitly, the plot module does this automatically. Signals are plotted via calls to `WriteValue` and `SynchronizeSignals`, and everything else is handled internally in the plot module.

The implementation of this module mainly consists of two processes, the process `InteractiveProcess` and `PlotProcess`. `InteractiveProcess` handles all input from the screen to the plot window. `PlotProcess` handles all output to the screen. It gets its input from `InteractiveProcess` via a monitor, and from a ring buffer. Procedure calls from the user program places commands in a ring buffer, the commands are then executed by `PlotProcess`, examples of such commands are commands for showing and updating the plot window.

7.6 Buttons

The `Button` module handles buttons sensitive to mouse clicks. It is possible to create a button with a number of states, change the states via clicks of the mouse, read the states from the program, and wait for the state of a specific button to be changed.

The implementation consists of two processes, one mailbox, and one monitor. The process `InteractiveProcess` handles all input from the screen. When a click is detected, a message of this is put in a mailbox to be read by `UpdateProcess`. Calls from the user program also puts requests in the mailbox for `UpdateProcess` to read and handle. `UpdateProcess` handles all output to the screen. Note that a button must have more than one state to be able to make use of `WaitChangeState`. In `Button` there is no way to wait for a click in *any* button; the programmer must know in which button he expects a click. Waiting for clicks in more than one button is solved by having one process for each button in the user program.

7.7 Numerical Menus

The `NumMenu` module handles one or several menus for inputting real values to a real-time system. Internally there are two sets of values, one which is shown in the menu and one set which is available via a call to `GetNumMenuState`. When the `Done` square is clicked the values shown in the menu are copied so that they are available via a call to `GetNumMenuState`. The routine `GetNumMenuStateWait` makes the calling process wait for a click in the `Done` rectangle, and then calls the second argument (a procedure parameter) with the state of the menu as parameter. By doing so, there is no way of changing the state of the menu until the procedure has completed its execution. The example below gives an example of the use of `GetNumMenuStateWait`.

The implementation consist of a list of one or several numerical menus and a process which waits for a click in any of the visible menus.

7.8 Logical Menus

The `LogMenu` handles menus for inputting logical values to a real-time system, and has a structure which is almost identical to the structure of `NumMenu`. The implementation of `LogMenu` is analogous to the implementation of `NumMenu`.

7.9 An Example

The implementation of a PI controller is used as an example of the use of the modules. The program starts on the next page. It is possible to change the parameters (k and T_i) of the controller. The reference signal is a square wave. The amplitude and frequency of the reference signal can be changed. All parameters are changed by clicking in an numerical menu. The reference value, the control signal, the process value, and the value of the integrator are plotted in the plot window during the operation of the controller. The program halts when the exit button is clicked. Note that the procedure `SetRegPar` is used as a call-back procedure in `GetNumMenuStateWait`. A similar program that was implemented using only the real-time kernel layer in the previous chapter was 16 pages long.

```
MODULE Regul;

IMPORT DebugPMD;
IMPORT RTMouse;
FROM AnalogIO IMPORT ADIn, DAOut;
FROM Graphics IMPORT point, handle, VirtualScreen, ReadString,
                     WriteString, EraseChar, color, buttonset,
                     ShowCursor;
FROM Kernel IMPORT Time, CreateProcess, SetPriority, IncTime,
                   WaitUntil, CurrentTime;
FROM Monitors IMPORT MonitorGate, InitMonitor, EnterMonitor,
                     LeaveMonitor;
FROM Signals IMPORT InitSignals, ChangeOmega, ChangeDelta,
                    ChangeFunction, GetRefValue, FunctionType,
                    MakeRefSignal, ChangeDirection;
FROM Plot IMPORT PlotType, InitPlot, MakePlot, MakeSignal,
                 SynchronizeSignals, WriteValue, ShowPlot,
                 HidePlot, SetPlotScale, SetTimeScale,
                 SetBackgroundColor, SetHidePlotColor;
FROM BarGraph IMPORT MakeBarGraph, ShowBarGraph, HideBarGraph,
                     PositionBarGraph, SizeBarGraph,
                     SetBarGraphLimits, ReadBarGraph,
                     WriteBarGraph, SetBarGraphColors,
                     ShowBarGraphValue, ActivateBarGraphArrows,
                     InitBarGraph, BarGraphType,
                     SetBarGraphHeader, ShowBarGraphLimits,
                     ActivateBarGraphHighAlarm,
                     ActivateBarGraphLowAlarm,
                     ActivateBarGraphNumericInput;
FROM Button IMPORT MakeButton, ShowButton, InitButton,
                   SetButtonAlt, GetStateWait, ButtonType,
                   GetState, ChangeButtonState;
FROM NumMenu IMPORT MakeNumMenu, SetNumMenuEntry, ShowNumMenu,
                    InitNumMenu, NumMenuType, GetNumMenuState,
                    HideNumMenu, GetNumMenuStateWait;

CONST
   KInit = 5.0;
   TiInit = 10.0;
   AmpInit = 0.1;

VAR ExitButton : ButtonType;
    NM : NumMenuType;
    Plt : PlotType;
    Pos : point;
    State : ARRAY [0..10] OF CHAR;
    RegPar : RECORD
               Mutex : MonitorGate;
               K, Ti, Amp : REAL;
             END;
(*-------------------------------------------------------*)
```

```
PROCEDURE GetRegPar(VAR p1, p2, p3 : REAL);
BEGIN
  WITH RegPar DO
    EnterMonitor(Mutex);
    p1 := K;
    p2 := Ti;
    p3 := Amp;
    LeaveMonitor(Mutex);
  END;
END GetRegPar;
(*-----------------------------------------------------------*)
PROCEDURE SetRegPar(p : ARRAY OF REAL);
BEGIN
  WITH RegPar DO
    EnterMonitor(Mutex);
    K := p[0];
    Ti := p[1];
    Amp := p[2];
    LeaveMonitor(Mutex);
  END;
  ChangeOmega("Ref", p[3]);
END SetRegPar;
(*-----------------------------------------------------------*)
PROCEDURE InitRegPar;
BEGIN
  WITH RegPar DO
    InitMonitor(Mutex, "Mutex");
    K := KInit;
    Ti := TiInit;
    Amp := AmpInit;
  END;
END InitRegPar;
(*-----------------------------------------------------------*)
(* Process *) PROCEDURE RegulProcess;

CONST h = 20; offset = 0.5;

VAR t : Time;
    amp, v, u, y, yref, e, i, k, ti : REAL;
    index : CARDINAL;
BEGIN
  SetPriority(10);
  CurrentTime(t);
  i := 0.0; index := 0;
  LOOP
    GetRefValue("Ref", yref);
    GetRegPar(k, ti, amp);
    yref := 2.0*amp*(yref - 0.5) + offset;
    y := ADIn(1);
    e := yref - y;
    i := i + k*e*FLOAT(h)/(1000.0*ti);
```

```
    v := k*e + i;
    u := v;
    IF v > 1.0 THEN
      u := 1.0;
    ELSIF v < 0.0 THEN
      u := 0.0;
    END;
    DAOut(1, u);
    i := i + FLOAT(h)/(1000.0*ti)*(u - v);
    IF index < 10 THEN
      INC(index);
    ELSE
      index := 0;
      WriteValue(Plt, "i", i);
      WriteValue(Plt, "u", u);
      WriteValue(Plt, "yref", yref);
      WriteValue(Plt, "y", y);
      SynchronizeSignals(Plt);
    END;
    IncTime(t, h);
    WaitUntil(t);
  END;
END RegulProcess;
(*-----------------------------------------------------------*)
(* Process *) PROCEDURE Opcom;
VAR i : CARDINAL; a : ARRAY [1..4] OF REAL;
BEGIN
  SetPriority(20);
  GetNumMenuState(NM, i, a);
  SetRegPar(a);
  LOOP
    GetNumMenuStateWait(NM, SetRegPar);
  END;
END Opcom;
(*-----------------------------------------------------------*)
BEGIN
  RTMouse.Init;

  InitPlot(20);
  Pos.h := 0.05;
  Pos.v := 0.55;
  MakePlot(Plt, "The Plot Window", Pos, 1.4, 0.4);
  MakeSignal(Plt, "u", lightblue);
  MakeSignal(Plt, "yref", lightcyan);
  MakeSignal(Plt, "y", blue);
  MakeSignal(Plt, "i", cyan);
  SetPlotScale(Plt, -1.0, 1.0);
  SetTimeScale(Plt, 30);
  SetBackgroundColor(Plt, white);
  ShowPlot(Plt);
```

```
  InitNumMenu(20);
  Pos.h := 0.05;
  Pos.v := 0.2;
  MakeNumMenu(NM, Pos, 4);
  SetNumMenuEntry(NM, "K", KInit);
  SetNumMenuEntry(NM, "Ti", TiInit);
  SetNumMenuEntry(NM, "Amplitude", AmpInit);
  SetNumMenuEntry(NM, "Frequency", 0.5);
  ShowNumMenu(NM);

  InitButton(20);
  Pos.h := 1.35;
  Pos.v := 0.05;
  MakeButton(ExitButton, Pos, 0.1, 0.1);
  SetButtonAlt(ExitButton, "Exit", red);
  SetButtonAlt(ExitButton, "Exit", red);
  ShowButton(ExitButton);

  InitSignals(20);
  MakeRefSignal("Ref", Step, 0.5);
  ChangeDelta(50);

  InitRegPar;
  CreateProcess(Opcom, 1000, "Opcom");
  CreateProcess(RegulProcess, 1000, "Regul");
  ShowCursor;

  GetStateWait(ExitButton, State);
END Regul.
```

7.10 The definition modules

The definition modules are given on the following pages in the following order: ListHandler, Signals, Bargraph, Plot, Button, NumMenu, LogMenu.

```
DEFINITION MODULE ListHandler;

FROM SYSTEM IMPORT ADDRESS;

EXPORT QUALIFIED
  ListTypePtr, NodeTypePtr,
  NewList, NewNode, InsertFirst, InsertLast,
  FirstNode, LastNode, PredNode, SuccNode,
  IsEmptyList, IsFirstNode, IsLastNode,
  ElementPtr, RemoveNode, ClearList;

TYPE
  ListTypePtr; NodeTypePtr;

PROCEDURE NewList() : ListTypePtr;
(* Creates a new empty list. *)

PROCEDURE NewNode(e : ADDRESS) : NodeTypePtr;
(* Create a new node and put a pointer to the element in
   the node. *)

PROCEDURE InsertFirst(n : NodeTypePtr; VAR l : ListTypePtr);
(* Put node n first in the list l. *)

PROCEDURE InsertLast(n : NodeTypePtr; VAR l : ListTypePtr);
(* Put node n last in the list l. *)

PROCEDURE FirstNode(l : ListTypePtr) : NodeTypePtr;
(* Returns a pointer to the first element in list l. *)

PROCEDURE LastNode(l : ListTypePtr) : NodeTypePtr;
(* Returns a pointer to the last node of the list. *)

PROCEDURE PredNode(n : NodeTypePtr) : NodeTypePtr;
(* Returns a pointer to the preceeding node. *)

PROCEDURE SuccNode(n : NodeTypePtr) : NodeTypePtr;
(* Returns a pointer to the succeeding node. *)

PROCEDURE IsEmptyList(l : ListTypePtr) : BOOLEAN;
(* Returns TRUE if the list is empty. *)

PROCEDURE IsFirstNode(n : NodeTypePtr) : BOOLEAN;
(* Returns TRUE if n points to the first node in a list. *)

PROCEDURE IsLastNode(n : NodeTypePtr) : BOOLEAN;
(* Returns TRUE if n points to the last node in a list. *)

PROCEDURE ElementPtr(n : NodeTypePtr) : ADDRESS;
(* This routine is used to get the actual element from the node.
   An example:
```

```
   N1 := NewNode(Obj);
   InsertFirst(N1, List);
   N2 := FirstNode(List);
   E1 := ElementPtr(N1);
   E2 := ElementPtr(N2);
   Now E1 and E2 points to the same element, namely Obj. *)

PROCEDURE RemoveNode(n : NodeTypePtr; l : ListTypePtr);
(* Removes a node from a list *)

PROCEDURE ClearList(l : ListTypePtr);
(* Deletes an entire list. *)

END ListHandler.
```

```
DEFINITION MODULE Signals;

FROM Graphics IMPORT point;

EXPORT QUALIFIED
  InitSignals, ChangeOmega, ChangeDelta, ChangeFunction,
  GetRefValue, FunctionType, MakeRefSignal, ChangeDirection;

TYPE
  FunctionType = (Sin, Pulse, Ramp, Step, Random);

PROCEDURE InitSignals(SignalPriority : CARDINAL);
(* Initiates the signal generator module. *)

PROCEDURE MakeRefSignal(SignalName : ARRAY OF CHAR;
                        FunctionName : FunctionType;
                        Omega : REAL);
(* Creates a signal from the generator. *)

PROCEDURE GetRefValue(SignalName : ARRAY OF CHAR;
                      VAR Value : REAL);
(* The value of SignalName gets assignad to Value. *)

PROCEDURE ChangeOmega(SignalName : ARRAY OF CHAR; Omega : REAL);
(* Changes the value of omega (the frequency) of the signal
   SignalName. *)

PROCEDURE ChangeDelta(Delta : CARDINAL);
(* Changes the frequency of the generation of new values.
   Delta is given in ms. *)

PROCEDURE ChangeDirection(SignalName : ARRAY OF CHAR;
                          Direction : REAL);
(* Changes the slope of the ramp function. *)

PROCEDURE ChangeFunction(SignalName : ARRAY OF CHAR;
                         Function : FunctionType);
(* Changes the function of the signal belonging to
   SignalName. *)

END Signals.
```

```
DEFINITION MODULE BarGraph;

FROM Graphics IMPORT point, color;

EXPORT QUALIFIED InitBarGraph, MakeBarGraph, PositionBarGraph,
                 SizeBarGraph, SetBarGraphLimits,
                 SetBarGraphColors, ShowBarGraph, HideBarGraph,
                 ReadBarGraph, WriteBarGraph, BarGraphType,
                 SetBarGraphHeader, ActivateBarGraphLowAlarm,
                 ActivateBarGraphHighAlarm,
                 ActivateBarGraphArrows,
                 ActivateBarGraphNumericInput,
                 DeactivateBarGraphClick, ShowBarGraphValue,
                 ShowBarGraphLimits;

TYPE BarGraphType;

PROCEDURE InitBarGraph(Priority : CARDINAL);
(* Initializes the BarGraph module. The processes in the
   BarGraph module will get the priority Priority. *)

PROCEDURE MakeBarGraph(VAR BarGraph : BarGraphType);
(* Creates a BarGraph, only one click-active bar is generated.
   All options must be generated with the other
   functions. *)

PROCEDURE ReadBarGraph(BarGraph : BarGraphType;
                       VAR Value : REAL);
(* Reads a value from BarGraph. *)

PROCEDURE WriteBarGraph(BarGraph : BarGraphType; Value : REAL);
(* Sets a value to BarGraph. *)

PROCEDURE ShowBarGraph(BarGraph : BarGraphType);
(* Shows the BarGraph(!). *)

PROCEDURE HideBarGraph(BarGraph : BarGraphType);
(* Hides the BarGraph. This routine can be used together with
   ShowBarGraph. *)

PROCEDURE PositionBarGraph(BarGraph : BarGraphType;
                           LowerLeftPos : point);
(* Sets the position of the lower left corner of the Bargraph.
   The point given in screen coordinates (0.0 < x < 1.5 and
   0.0 < y < 1.0). *)

PROCEDURE SizeBarGraph(BarGraph : BarGraphType;
                       Width, Height : REAL);
(* Defines the size of the BarGraph. The sizes are given in
   screen coordinates. *)
```

```
PROCEDURE SetBarGraphLimits(BarGraph : BarGraphType;
                            Lower, Upper : REAL);
(* Sets the upper and lower limits of the values which can be
   shown in the BarGraph. *)

PROCEDURE ShowBarGraphLimits(BarGraph : BarGraphType;
                             LimitColor : color);
(* Writes the limits of the BarGraphs numerically. *)

PROCEDURE SetBarGraphColors(BarGraph : BarGraphType;
                            BarColor, BackGroundColor,
                            HideColor : color);
(* Sets 1) Colour of the bar
        2) Colour under the bar
        3) Colour to erase the BarGraph with. *)

PROCEDURE SetBarGraphHeader(BarGraph : BarGraphType;
                            Header : ARRAY OF CHAR;
                            HeaderColor : color);
(* Sets a text at the header of BarGraph. *)

PROCEDURE ActivateBarGraphArrows(BarGraph : BarGraphType);
(* Gives the possibility to use up and down arrows to
   manipulate the BarGraph. *)

PROCEDURE ActivateBarGraphNumericInput(BarGraph : BarGraphType);
(* Gives the possibility to input a numeric value to BarGraph
   via the keyboard. *)

PROCEDURE DeactivateBarGraphClick(BarGraph : BarGraphType);
(* Disables the possibility to set a value to BagGraph via a
   mouse click. *)

PROCEDURE ShowBarGraphValue(BarGraph : BarGraphType;
                            ValueColor : color);
(* Shows the value of BarGraph numerically. *)

PROCEDURE ActivateBarGraphLowAlarm(BarGrap : BarGraphType;
                                   AlarmLevel : REAL;
                                   AlarmColor : color);
(* Defines low alarm level. The BarGraph will change color at
   this level. *)

PROCEDURE ActivateBarGraphHighAlarm(BarGrap : BarGraphType;
                                    AlarmLevel : REAL;
                                    AlarmColor : color);
(* Defines high alarm level. The BarGraph will change color at
   this level. *)

END BarGraph.
```

```
DEFINITION MODULE Plot;

FROM Graphics IMPORT point, color;

EXPORT QUALIFIED PlotType, MakePlot, MakeSignal,
                 SynchronizeSignals, WriteValue, ShowPlot,
                 HidePlot, InitPlot, SetPlotScale, SetTimeScale,
                 SetBackgroundColor, SetHidePlotColor;

TYPE PlotType;

PROCEDURE InitPlot(PlotPriority : CARDINAL);
(* Initiates the plot module. PlotPriority denotes the priority
of the processes internal to the plot module. *)

PROCEDURE MakePlot(VAR Plot : PlotType;
                   Name : ARRAY OF CHAR; LoLeft : point;
                   Width, Height : REAL);
(* Creates a plot window with the header Name. LoLeft specifies
   the lower left corner of the plot, Width and Height denotes
   the size of the plot. *)

PROCEDURE MakeSignal(P : PlotType;
                     SignalName : ARRAY OF CHAR;
                     Colour : color);
(* Makes it possible to plot a signal named SignalName in a plot
   window P. The colour of the signal is set by Colour. *)

PROCEDURE SynchronizeSignals(P : PlotType);
(* Synchronizes the signals so that all calls to WriteValue
   which happen before the next SynchronizeSignals call, are
   internally considered to be simutaneous. *)

PROCEDURE WriteValue(P : PlotType; SignalName : ARRAY OF CHAR;
                     Value : REAL);
(* Plots the value of the signal in the plot window. *)

PROCEDURE ShowPlot(P : PlotType);
(* Shows the plot window on the screen. *)

PROCEDURE HidePlot(P : PlotType);
(* Hides the plot window. *)

PROCEDURE SetPlotScale(P : PlotType; ymin,ymax : REAL);
(* Changes the scale of the plot in y-direction. *)

PROCEDURE SetTimeScale(P : PlotType; seconds : CARDINAL);
(* Sets the time scale of the plot. *)

PROCEDURE SetBackgroundColor(P : PlotType; Colour : color);
(* Sets the background color of the window. *)
```

```
PROCEDURE SetHidePlotColor(P : PlotType; Colour : color);
(* Sets the color for erasing the window. *)

END Plot.
```

```
DEFINITION MODULE Button;

FROM Graphics IMPORT point, color;

EXPORT QUALIFIED InitButton, MakeButton, SetButtonAlt,
                 ShowButton, HideButton, GetState,
                 GetStateWait, ChangeButtonState,
                 ButtonType, SetHideButtonColor;

TYPE ButtonType;

PROCEDURE InitButton(ButtonPriority : CARDINAL);
(* Initializes the Button module. ButtonPriority denotes
   the priority of the internal processes of the Button
   module. *)

PROCEDURE MakeButton(VAR Button : ButtonType;
                     LoLeft : point;
                     Width, Height : REAL);
(* Creates a click sensitive Button area. LoLeft is the point
   of the lower left corner of the Button, and Width and Height
   are the width and height of the Button. *)

PROCEDURE GetState(Button : ButtonType;
                   VAR State : ARRAY OF CHAR);
(* Gives the current state of Button. *)

PROCEDURE ChangeButtonState(Button : ButtonType;
                            State : ARRAY OF CHAR);
(* Changes the state of Button to State. *)

PROCEDURE ShowButton(Button : ButtonType);
(* Shows the Button on the screen. *)

PROCEDURE HideButton(Button : ButtonType);
(* Hides the Button. *)

PROCEDURE SetButtonAlt(Button : ButtonType;
                       State : ARRAY OF CHAR;
                       Colour : color);
(* Adds a new state to the Button. State denotes the name
   of the state, and Colour is the background colour of
   the Button. *)

PROCEDURE GetStateWait(Button : ButtonType;
                       VAR State : ARRAY OF CHAR);
(* Waits until the state of Button changes, and then gives
   the new state in State. *)

PROCEDURE SetHideButtonColor(Button : ButtonType;
                             Colour : color);
```

```
(* Sets the erase colour of Button.This colour is used in
   HideButton. *)

END Button.
```

```
DEFINITION MODULE NumMenu;

FROM Graphics IMPORT rectangle, point, color;

EXPORT QUALIFIED SetNumMenuEntry, MakeNumMenu,
                 GetNumMenuState, ShowNumMenu, HideNumMenu,
                 InitNumMenu, SetNumMenuColors, SetNumMenuState,
                 NumMenuType, GetNumMenuStateWait,
                 GetNumMenuSize;

TYPE NumMenuType;
     CallBackProcedure = PROCEDURE(ARRAY OF REAL);

PROCEDURE InitNumMenu(Priority : CARDINAL);
(* Initializes the NumMenu system. *)


PROCEDURE MakeNumMenu(VAR NM : NumMenuType;
                          LowLeft : point;
                          inputs : CARDINAL);
(* Creates a numerical menu with inputs entries and
   its lower left corner in LoLeft. *)

PROCEDURE GetNumMenuSize(NM : NumMenuType; VAR r : rectangle);
(* Returns the rectangle on the screen which NM occupies *)

PROCEDURE SetNumMenuEntry(NM : NumMenuType;
                          text : ARRAY OF CHAR;
                          num : REAL);
(* Sets the text string and initial value of an entry in
   a NumMenu. The text string is truncated to 15 characters. *)

PROCEDURE SetNumMenuColors(NM : NumMenuType;
                           bottom, text, alert, hide : color);
(* Sets the colors of a NumMenu bottom is the background
   color in the menu, text is the text color, alert is the
   color of changed text, and hide is the color used to
   hide the menu. *)

PROCEDURE ShowNumMenu(NM : NumMenuType);

(* Shows NM. *)

PROCEDURE HideNumMenu(NM : NumMenuType);

(* Hides NM. *)

PROCEDURE SetNumMenuState(NM : NumMenuType;
                          nums : ARRAY OF REAL);

PROCEDURE GetNumMenuState(NM : NumMenuType;
```

```
                                VAR inputs : CARDINAL;
                                VAR nums : ARRAY OF REAL);
(* Returns the number of values and the values
   stored in the menu. *)

PROCEDURE GetNumMenuStateWait(NM : NumMenuType;
                              CB : CallBackProcedure);
(* Calls CB with the state of NM as argument when the Done
   rectangle is clicked. This prevents the state to change
   before the call of CB has returned. *)

END NumMenu.
```

```
DEFINITION MODULE LogMenu;

FROM Graphics IMPORT rectangle, point, color;

EXPORT QUALIFIED SetLogMenuEntry, MakeLogMenu,
                 GetLogMenuState, ShowLogMenu, HideLogMenu,
                 InitLogMenu, SetLogMenuColors, SetLogMenuState,
                 LogMenuType, GetLogMenuStateWait,
                 GetLogMenuSize;

TYPE LogMenuType;
     CallBackProcedure = PROCEDURE(ARRAY OF BOOLEAN);

PROCEDURE InitLogMenu(Priority : CARDINAL);
(* Initializes the LogMenu system. *)

PROCEDURE MakeLogMenu(VAR LM : LogMenuType;
                          LoLeft : point;
                          inputs : CARDINAL);
(* Creates a logical menu with inputs entries and
   its lower left corner in LoLeft. *)

PROCEDURE GetLogMenuSize(LM : LogMenuType; VAR r : rectangle);
(* Returns the rectangle on the screen which LM occupies. *)

PROCEDURE SetLogMenuEntry(LM : LogMenuType;
                          text : ARRAY OF CHAR;
                          log : BOOLEAN);
(* Sets the text string and initial value of an entry in
   a LogMenu. The text string is truncated to 15 characters. *)

PROCEDURE SetLogMenuColors(LM : LogMenuType; true, false, text,
                           alert, hide : color);
(* Sets the colors of a LogMenu bottom is the background
   color in the menu, text is the text color, alert is the
   color of changed text, and hide is the color used to
   hide the menu. *)

PROCEDURE ShowLogMenu(LM : LogMenuType);
(* Shows LM. *)

PROCEDURE HideLogMenu(LM : LogMenuType);
(* Hides LM. *)

PROCEDURE SetLogMenuState(LM : LogMenuType;
                          logs : ARRAY OF BOOLEAN);

PROCEDURE GetLogMenuState(LM : LogMenuType;
                          VAR inputs : CARDINAL;
                          VAR logs : ARRAY OF BOOLEAN);
(* Returns the number of values and the values
```

```
   stored in the menu. *)

PROCEDURE GetLogMenuStateWait(LM : LogMenuType;
                              CB : CallBackProcedure);
(* Calls CB with the state of LM as argument when the Done
   rectangle is clicked. This prevents the state to change
   before the call of CB has returned. *)

END LogMenu.
```

8

Incorporating a Controller Language

O. Dahl

GOAL: To introduce how an application specific language can be introduced in an automation system

Application specific languages can be used to greatly improve the efficiency of programming in several application areas. Such specific languages should be easy to use within a general programming language or easy to interface to a general programming language, so that the user is not limited to the constructs of the specific language. A solution must therefore penetrate the language, a compiler for the language, the interface of the generated code, and an environment for its use. This chapter will treat these aspects using a language for description of the control algorithm. A compiler for translation of control algorithms is used in combination with a general program for real time control. The compiler translates control algorithms, written in a design language, to an implementation language, and generates code for connecting the control algorithms to the user interface. The translated algorithms are then automatically incorporated in the real time control program. The resulting executable program have a number of interactive facilities such as interconnection of controllers, plotting and textual display of all variables, and data logging.

A control algorithm is a dynamic system, and Simnon is an already existing language for describing dynamic systems. Simnon is also a language for simulation of nonlinear systems. It is an advantage to use a simulation language, because then exactly the same code is used for the development using simulation, and for the implementation. The implementation language i.e. the code generated by the compiler is chosen to be Modula-2. These ideas can of course be used in many ways. However, from now on the focus will be on a complete system for efficient implementation of real time control systems, and an overview of these aspects are given in Section 8.1. The environment is an automation system as described in Chapter 3, and procedure variables are used to be able to link in any procedure in the list. The present chapter is a natural continuation of Chapters 6 and 7, where first the real-time aspects were simplified, and then the operator communication was simplified. Now, the coding of the control algorithm is simplified.

Section 8.2 presents an example of a session using the system. A short outline of the

system design is given in Section 8.3. The two software components in the system, the general control program, and the Simnon to Modula-2 compiler are described in Sections 8.4 and 8.5.

8.1 An Interactive Environment for Implementation of Control Systems

When a control algorithm is designed, a laboratory experiment is often desired to verify the design. This can be more or less time consuming, depending on the tools available. The experiment requires a real time implementation of the control algorithm, which typically requires programming, and consideration of real time problems, the user interface, data logging etc. Further, the risk of introducing errors in the control algorithm during programming may be significant. The motivation of the work presented here is the simplification of real time implementation of control systems. The idea is to combine a compiler for translation of control algorithms, and a general program for real time control. The requirements on the user is to write the control algorithm in a design language. A real time version of the algorithm is then automatically generated, and included in the general control program. The control algorithms are written as discrete time Simnon systems, and translated by the compiler to Modula-2. The compiler is implemented in Scheme. The generated Modula-2 code is then compiled and linked with the general control program, which is implemented in Modula-2. The result of the translation is an interactive program for real time control, containing a command driven user interface, graphics, and data logging. The control algorithm is automatically included in the program as a system type, of which instances can be created at run time. Other facilities are interconnection of controllers, plotting, graphic input via bargraphs, display of all variables, and editing of parameters. The running controllers and the user interface are separate concurrent processes, thus making it possible to interactively e.g. change parameters in the running control algorithm. This should be compared with e.g. Simnon with Real Time Capability, where there is no concurrency. The modification of a parameter in a controller then requires that the periodic execution of the controller is stopped, while modifying the parameters. Another difference compared to Simnon is that here each control algorithm is regarded as a system type. This means that e.g. a cascade controller can be implemented by having one Simnon system defining the algorithm, and then create two instances of the controller and make the connections between the controllers at run time.

By choosing a high level programming language as implementation language, it is also possible to use a combination of the two languages in the controller implementation, e.g. writing the main part of the controller in Simnon, and use Modula-2 to call e.g. an optimization routine from a Modula-2 library. The combination of Simnon and Modula-2 is also useful when the main part of the control algorithm is written in Modula-2, and the Simnon system is used to specify which variables that should be accessible from the user interface. This makes it easy to e.g. add new parameters in the controller by introducing them in the Simnon part of the controller. They are then automatically made accessible to the user interface, and can be modified on line. Using a high level language as implementation language also makes the code generation simpler than using direct translation to machine language. The full capabilities of Modula-2 can be utilized, e.g. it is possible to use Modula-2 modules with special purpose routines that are not implementable in Simnon, e.g. pole placement calculations. By using Modula-2, it is also easy to create libraries of control algorithms. The system software is currently running on IBM PC/AT computers, and on a VME-board based system using a Motorola M68030 as real time computer, and a Sun workstation for programming. The system has been used in research projects, and in laboratory experiments for education, where it has reduced the implementation time considerably.

A session with the system is described in the next section. A cascade PI controller is implemented. The example includes how the translation from Simnon to Modula-2 is done, and it is described how e.g. graphical presentation of real time data, and data logging can be achieved.

8.2 A Session with the System

A simple example is given to demonstrate the system. The controller is a PI controller, implemented in Simnon as

```
Discrete System pi

input yr y
output u
state i
new ni

time t
tsamp ts

u = k*e + i

e = yr - y
ni = i + k*h/Ti*e

ts = t+h

K : 1
Ti : 100000
h : 0.1

END
```

Listing 8.1 A Simnon PI Controller

The example describes how to obtain a real time version of the PI controller, and a typical command dialogue is given, including specification of A/D and D/A channels, selection of signals to be plotted, editing of parameters in the running controller, and data logging.

Real Time Implementation

The Simnon code for the PI controller is stored in the file `pi.t`. The SIM2DDC (Simnon to DDC, Direct Digital Control) environment is implemented as a set of commands at the operating system level. There are commands for translation from Simnon to Modula-2, and for direct translation to executable code. The direct translation is done by first translating from Simnon to Modula-2, and then invoking the Modula-2 compiler and linker. The command `sim2exe pi` translates the PI controller in `pi.t` to Modula-2 and generates an executable program, containing the PI controller as a system type. The program is stored in `ddcmain.exe` and is executed by typing `ddcmain`. The program is command driven, and contains data logging and graphics for plotting.

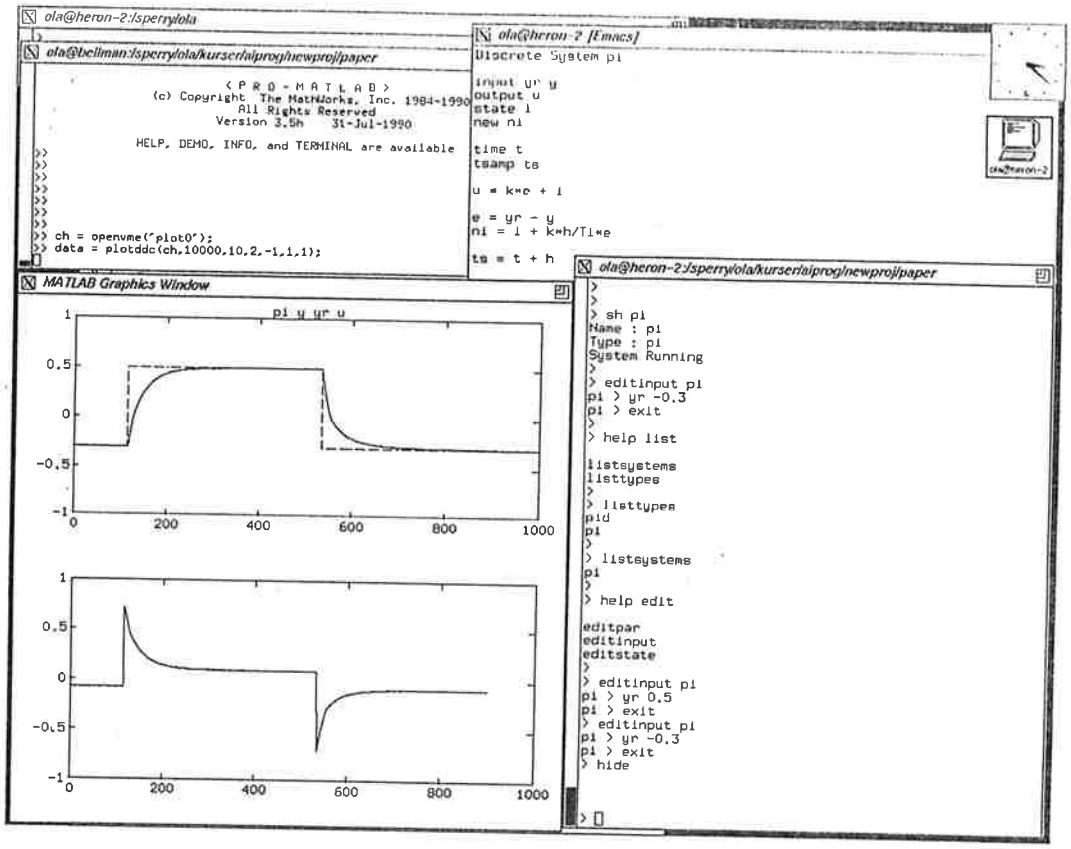

Figure 8.1 The graphical display of the real time control program.

Interacting with the Real Time Program

Initialization of a controller is typically done by using a command file, containing a list of interactive commands. The contents of a command file for initialization of the PI controller could be

```
" Initialization of PI controller
"
" File : init.mac

newsystem pi pi
adconnect pi y 0
daconnect pi u 0
plotconnect pi y 0 yr 0 u 1
```

The command `newsystem pi pi` creates a system named `pi` of the system type `pi`. The system is initially at rest and there are no connections of inputs and outputs. The commands `adconnect` and `daconnect` makes the connections to the A/D and D/A converters. The plotting facility is implemented in two diagrams on the screen, showing the plotted signals as functions of time, see Figure 1. The command `plotconnect pi y 0 yr 0 u 1` connects the signals y and yr to channel 0 (upper plot on the screen), and the signal u to channel 1 (lower plot on the screen). The command file `init.mac` is executed by typing `init` as a command. The system `pi` is then created and the connections are made. Interactive commands can now be given, e.g. to start and stop the controller, change

parameters etc. The command `runsystem pi` starts the PI controller, i.e. the algorithm will execute periodically with a sampling period determined by the default value of the associated parameter in the Simnon system. The plotted signals in the system `pi` are displayed by using the command `showsystem pi`. A typical graphical display from the Sun/VME implementation is shown in Figure 1. The plotting is here implemented in Matlab, which has been extended with an interface to real time control systems.

Further Interaction

We have shown how to go from a Simnon system description to a running real time controller. When the controller is running, further interaction may be desirable, e.g. displaying values of states and parameters, and modifying parameters. The parameters of the running controller are modified by the commands

```
> editpar pi
pi > k 4
pi > Ti 10
pi > exit
```

All parameters are changed at the same time, i.e. the command `exit` copies the modified parameters to the running controller.

The system type facility is useful e.g. for cascade PI control, which can be implemented by creating one more PI controller and connecting it to the previously created controller. The commands

```
> newsystem pi2 pi
> adconnect pi2 y 1
> plotc pi2 y 0 yr 0 u 1
```

creates a new PI controller with the same plot connections. A cascade connection is now obtained by connecting the output of `pi2` to the reference input of `pi`. This is done by the commands

```
> stopsystem pi
> intoutconnect pi2 u 0
> intinconnect pi yr 0
```

where after stopping the controller `pi`, the output `u` of `pi2` is connected to the reference input `yr` of `pi`. The cascade controller is started by the commands `runsystem pi` and `runsystem pi2`. The controllers `pi` and `pi2` are then running as two separate concurrent processes, having the same priority. The priority can be modified by the command `setpriority`. When using cascade control, it is often desirable that the controllers run synchronously. This is achieved by the command `synchronize pi2 pi`, which makes the controllers run synchronously with a sampling interval determined by `pi2`. The outer controller `pi2` is executed before the inner controller `pi`.

Data Logging

In the PC-AT implementation, data from the experiment can be stored in files by using the commands `logon` and `logoff`. The stored data can be directly loaded into Matlab for off-line analysis, e.g. system identification. In the Sun/VME implementation, since Matlab is used for graphical presentation, data can be saved on files by using commands available in Matlab.

8.3 Design Outline

The system is composed of two software components, the compiler for translation from Simnon to Modula-2, and the real time control program. The control program is general in the sense that it is not designed for a special control algorithm. Control algorithms are represented as system types, i.e. each Simnon system corresponds to a system type. Each control algorithm is written as a separate Modula-2 module, in a standardized format. The module includes both the control algorithm and procedures for connecting the algorithm to the user interface. This standardization of the format of the code makes it possible to use automatic generation of the entire module, given a high level specification of the algorithm, its inputs, outputs, states etc.

It should be noted that the algorithm itself is typically only a small part of the code. The main part of the code is for "administration", e.g. procedures for reading and writing the controller parameters. As an example, the PI controller shown in Listing 1 contains 22 lines of Simnon code, and the generated Modula-2 code is 760 lines.

8.4 A General Real Time Control Program

The control program is implemented in Modula-2, and consists of about 20 modules, with a total of about 10000 lines of source code. The program is used as a compiled library. In order to minimize the amount of compilation needed when a new control algorithm is incorporated in the system, the size of the main program is minimized. A typical main program may look like

```
MODULE DDCMain;

IMPORT pid, pi, adaptive;
IMPORT DDC;

BEGIN
  DDC.Init;
  pid.InitSystemType;
  pi.InitSystemType;
  adaptive.InitSystemType;
  DDC.Start;
END DDCMain.
```

Compiling this program, and linking with the general control program, gives an executable real time controller containing the algorithms `pid`, `pi`, and `adaptive`, each specified in separate modules. The statement `IMPORT DDC` makes the connection to the general control program. The controller modules `pid`, `pi`, and `adaptive`, are typically automatically generated from Simnon system descriptions. Note that also the main program can be

automatically generated, since each controller module is treated in the same way, i.e. it should be imported, and its procedure for initialization, `InitSystemType`, should be called. The contents of the controller module `pi` will now be described, showing the standardized format of the code. The corresponding Simnon system is shown in Listing 1.

A Standardized Format for Controller Code

Each control algorithm can be regarded as an abstract data type. There are a fixed set of procedures that operate on objects of this data type. Typical procedures are `Init` for initialization, `UpDate` for computing the controller output, and `NewState` for updating the controller state. The procedures `UpDate` and `NewState` together forms the control algorithm, as described in the Simnon system. For the PI controller in Listing 1, their structure is as follows.

```
PROCEDURE UpDate(System : SystemType);
BEGIN
  WITH System^ DO
    e := yr - y;
    u := k*e + i;
  END;
END UpDate;

PROCEDURE NewState(System : SystemType);
BEGIN
  WITH System^ DO
    ni := i + k*h/ti*e;
    i := ni;
  END;
END NewState;
```

Listing 8.2 The generated code for the PI control algorithm

The separation of the Simnon code in two procedures is done to minimize the time from A/D conversion to D/A conversion. A typical calling sequence in the real time process handling the controller is

```
ADIn(SystemInputs);
UpDate(System);
DAOut(SystemOutputs);
NewState(System);
```

The user interface to the controller is implemented in another set of procedures, also operating on the data type `SystemType`. The algorithm variables are separated in the groups *inputs*, *outputs*, *states*, *parameters*, and *auxiliary variables*. Each variable group has four procedures. For the parameters, the procedures have the following interfaces.

```
PROCEDURE WriteParVariable(
  S : SystemType;
  Name : ARRAY OF CHAR;
  Value : REAL;
```

```
    VAR NameOk : BOOLEAN);

  PROCEDURE ReadParVariable(
    S : SystemType;
    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);
```

Listing 8.3 The generated procedures for accessing the parameters in the PI control algorithm

Since the interface to the control algorithm is standardized in this form, a general user interface can be written. This interface then needs no modification when a new control algorithm is implemented. As an example, displaying the values of all parameters in a particular controller is done by first calling `GetNumberOfPars` to get the number of parameters. For each parameter number, the corresponding name and value is then obtained by first calling `GetParName` and then `ReadParVariable`.

8.5 A Simnon to Modula-2 Compiler

A compiler for generation of Modula-2 code, having the format described in the previous section, has been implemented in Scheme. The code size is approximately 5000 lines. The source text used by the compiler is a discrete time Simnon system, and the generated code is a Modula-2 module The real time facilities are implemented using the primitives in the previous chapters.

The processing of a Simnon source text starts with lexical and syntax analysis. The syntax analysis is implemented as a recursive descent parser. After that, declarations and assignments are analyzed. Multiple declarations and illegal assignments, e.g. assignments of input variables, are detected. The parameter assignments and the variable assignments are then sorted. The code generation is then done, resulting in the control algorithm procedures `Init`, `UpDate`, and `NewState`, and the access procedures, e.g. `ReadParVariable`, see Listings 2 and 3. Note that the separation of the Simnon code in the procedures `UpDate` and `NewState` is done so that `UpDate` always contains the minimum amount of computations needed for computing the output variables.

8.6 Summary

It is demonstrated how a compiler, in combination with a general program for real time control, can be used to simplify the real time implementation of control systems. The compiler translates control algorithms from a design language to an implementation language. The algorithms are written as discrete time Simnon systems, and translated by the compiler to Modula-2 modules. Besides translating the control algorithm, the compiler also generates procedures for connecting the algorithm to the user interface. The generated code has a standardized format, and can be incorporated in a general program for real time control, where the user interface is made independent of the control algorithm,

and where new control algorithms can be introduced without modifying the rest of the program.

9

Applications

T. Hägglund, K-E. Årzén, and K. Nilsson

GOAL: To present three examples of applications from different areas.

There exist both commercial systems and research projects along the lines of the previous chapters. Different applications lead to different specific designs. This chapter will present three examples of solutions for different control problems. Section 9.1 presents aspects of the development of the commercial system ECA-400. Robot control systems are discussed in Section 9.2, and expert control is discussed in Section 9.3.

9.1 Development of a commercial system

A development project involves normally several parallel activities that have to be synchronized. These can e.g. be

- Hardware development
- Software development
- Documentation and manual writing
- Marketing planning
- Education
- Production planning
- Service planning

There are normally several participants involved in each of these activities. These participants must also be synchronized in the sense that each person should know exactly what his task in the project is, and when this task should be solved. If this synchronization is not performed properly, it will result in increased development costs and a delayed completion of the project. To avoid this, it is important to have a good planning of the project. How this planning should be performed is of course varying between different projects. The principle is, however, the same, namely to divide the project into several smaller well defined sub-projects. It is also important to split the project time into several phases. This simplifies the discovery of possible delays in the project, and makes it easier for the project leaders to allocate the resources.

We will here shortly describe one way to plan a development project, with concentration on the software development part of the project. A development project may involve the following phases: Pre-investigation, Function description, System design, Detailed design, Specification of sub-systems, Programming, Integration, Product verification, and Completion. We will describe each of these phases in more detail.

Phase 1: The pre-investigation

The first and most important question to answer is: Should this project be performed at all? It is often a very hard decision to stop a project that has already been running for a while. Therefore, it is important that the project decision is taken by the right people based on relevant information.

The goal for the pre-investigation is to provide the information needed for the project decision. The pre-investigation can e.g. include a market analysis, a preliminary function description of the project, a suggested product design, estimated development costs and production costs, and a time schedule for the project.

When the pre-investigation is completed, it is decided whether to continue with the project or not. If the project is continuing, time schedules are determined, and resources in terms of personnel and equipment are allocated.

Phase 2: Function description

In this phase, a document describing the function of the product is prepared. This document should describe, *in detail*, the function of the product. If the product e.g. is a controller, the function description should include the algorithms (PID, filters etc.), the number and types of inputs and outputs, the man-machine interfaces etc. In this description it should e.g. be possible to read how to change a controller parameter; which buttons to press, which commands to write, the allowed range of the parameter etc.

The preparation of a function description document is time consuming and often more or less neglected in development projects, even though the time spent with this work is mostly saved later in the project. The function description can often be used as a first version of a user manual.

Phase 3: System design

Based on the function description, an overall system design is performed. It is e.g. decided what hardware to use and the software part of the project is divided into some major parts such as time scheduling, I/O-handling, memory and data base handling and algorithms. When this phase is completed, the hardware and software development can be separated from each other.

Phase 4: Detailed design

In the software part of this phase, the system design is divided into system functions. These functions should be as independent of each other as possible. The dividing should be such that each system function is so small that one single person can take care of its development.

If the product is a controller, the functions can be input-handling, output-handling, user-interfaces, controller algorithms, etc.

Phase 5: Specification of sub-systems

Each system function now has a well defined task and well defined interfaces to the other system functions. As said above, it is normally handled by one individual. Before the coding starts, this sub-system should also be organized into smaller functions and procedures with well-defined tasks and interfaces.

Phase 6: Programming

Now, the code generation starts. If the previous phases are performed properly, this phase consist of the writing of many but well-defined and small pieces of code. A relatively simple task.

Phase 7: Integration

The idea behind the previous phases was to divide the project into smaller and smaller well-defined sub-projects. In this phase, the different sup-systems are linked together to the final product.

Phase 8: Product verification

In this phase the product is verified. It should correspond to the function description derived in Phase 2. The tests may start in the lab but are normally continued as field tests at some customers.

Phase 9: Completion

This is an important part of the project. Even if the field tests show that the product works as expected, the project is not completed! The documentation must be e.g. be fulfilled, and people should be given responsibilities for the maintenance of the different parts of the product.

Summary

To summarize, a development project involves normally a big synchronization problem. The solution to this problem is to have a sound project planning, where the project is divided into smaller well defined sup-projects, and where the project time is divided into several phases, which should be performed in the right order.

The most common mistake in software development projects is that the coding part starts far to early, and that the function description is to sketchy or even missing.

9.2 Robot control

There are several interesting problems in robot programming and control. These problems in robot programming, sensor based motion control, and real-time aspects are to some extent treated in Chapters 12 and 13 in Craig, but further research in this area is in progress within the Department of Automatic Control, Lund. Some background and ideas in this research will be given in this section.

Background

Industrial robots have become an important production tool in a number of standard applications. The use of robots in modern production systems in general is however quite limited. The reasons for this are both technical and economical. The tasks not performed by robots are then performed by NC-machines, fixed automation, and manual work. The aim of the current research is to extend the applicability of industrial robots.

The main differences between robots and fixed automation are the combined demands for flexibility and performance from a mechanically flexible structure. The demand on performance implies use of modern control theory, both for control of the robot itself, and for control of motions in certain applications with additional external equipment and sensors. The demand for flexibility means that both reconfiguration for new applications and reprogramming for new tasks should be possible and easy. Both the programming

of the system on different levels for different types of users, and the architecture of the system, is then of major concern.

One approach for making robots easier to use is taken in research on task-level programming, off-line programming systems, production planning, etc. The aim there is to create robot independent tools, and to reason about the production process in high level terms. The issue is then how to encode and handle knowledge of the production process and its objects within some kind of system. It should be quite clear that for such an approach to be attractive in practice, the high level actions requested must be performed accurately, and with performance that utilizes the comparably expensive mechanical robot. It is the basic (i.e. non AI) robot control that is the subject, and the results are intended to provide a base for research in new applications and knowledge based robot control.

Robot programming

Manipulator oriented programming is basically performed in two different styles today. Interactive menu-based teach-in is normally used on-line and has proved to be preferable in standard applications, especially when the robot programmer has no or little experience in computer programming. An example of this style is ARLA from ABB Robotics. Languages of this type will be referred to as *on-line* languages. Another style is to program robots in languages resembling computer programming languages. The user with some experience in computer programming can then build new functions in a computer programming environment, or with the help of an off-line robot programming system if the programming is done off-line, i.e. without occupying the robot itself. Languages of this type will therefore be referred to as *off-line* languages. Normally, however, some positions must be redefined on-line, and if the robot operator also needs to modify the program on-line, programming experience is required. An example of this style is VAL-II from Unimation, or the more general robot independent language SIL from SILMA. Today's systems are designed primarily for only one of the on-line or off-line programming styles. It would be desirable to have a system suitable for both cases.

Limitations of today's industrial robots

Many applications suitable for an inexpensive, easily reconfigurable, light weight structure machine, with normal requirements on accuracy and speed, i.e. applications where robots should be used, can not be properly handled by today's industrial robots. There are also applications where robots are used today, but where improved efficiency in some sense would be desirable. Major limitations are:

- Deficient control performance resulting in inaccurate or slow motions.
- Information available in internal sensor signals can not be utilized for improved functionality or performance for a certain application.
- External sensors for motion control can not be fully utilized. The motion control system forms a closed structure, and only predefined cases of control can be used.
- Insufficient support for robot calibration. Calibration of the kinematic model is possible for some simpler robots. Calibration of dynamics, friction, or characteristics of external equipment can however not be done.
- When cycle time is critical, as in most assembly applications, most of the time consumed beyond the theoretically minimal time is typically spent in some critical parts of the task. Optimal motion in critical regions of the workspace is then of interest. Some kind of user controlled adaptive control or auto-tuning would be the solution.
- Programming the robot and its surrounding equipment is sometimes too hard, especially when trying to get around the limitations above.

- The cost of the mechanics, the control system, installation, programming, etc., is too high. Robot manufacturers are therefore optimizing the systems for the most profitable applications (today), which then also makes development for new applications harder.

The limitations are in the control system. Even the cost of the mechanical robot can probably be reduced by more advanced control.

The need for a new open system structure

The limited applicability of robots (and the interesting problems in robotics) has inspired a lot of research activity. New algorithms and technical solutions in robotics research have shown promising results in special cases. Implementations for real applications are desirable, but the industrial systems have excessively closed and rigid structure. When trying to merge technology from different fields, there is a need for a new open layered system structure because:

- Support is needed to manage the increased system complexity, that will result from adding more features.
- Different types of users will program the system and add sensors on different levels. Different abstraction barriers within the system are then needed.
- To make functions in different parts of the system compatible with each other, certain restrictions must be imposed on the technical solutions. The system structure should support this.
- The system should not be specifically on-line or off-line oriented. Advantages of both these styles should be kept.
- When solutions for new applications have been developed, the system structure should support exclusion of unnecessary parts for development of new application optimized control systems.

Several system structures have been defined within the robotics field, but mostly for mobile robots or aero-space robots. One example is the well known NASREM structure. The requirements for industrial robots are however not met, and a new system structure is needed.

9.3 Real-time expert systems

Expert systems or knowledge-based systems is an area of Artificial Intelligence (AI) that has grown rapidly during the last years. The basic definition of an expert system is a program that solves problems within a specific, limited domain that normally would require human expertise. The problem is solved by mimicing the human expert's reasoning by representing his reasoning strategies and knowledge in the program. Some of the earliest and most well-known expert system applications are in the fields of medical diagnosis and computer system configuration.

Expert systems more or less fulfill a number of different characteristics of which the emulation of human problem solving is the most prominent. Another common characteristic is that knowledge about the problem domain is explicitly represented in an identifiable, separate part of the program. This so called knowledge base is separated from the inference engine that executes the program by operating upon the knowledge base. The explicit knowledge representation gives an expert system an declarative nature in contrast to traditional procedural languages such as C and Pascal where the domain knowledge is expressed in the form of program statements.

Well-developed explanation facilities is another expert system characteristic. In order to be accepted by the user it is important that the system can automatically generate an explanation of its conclusions. The possibility to reason with uncertainty is another feature that might be present in expert systems. Knowledge might be uncertain and this uncertainty is expressed in terms of probability measures that are associated with facts or rules and which are propagated during the reasoning.

The representation of knowledge is a key issue in expert systems. Common representation forms are rules and objects. The general structure of a rule is

```
If <conditions>
 then
  <actions>
```

The condition part contains a condition that must be fulfilled for the rule to be applicable. The action part contains the actions which are taken when the rule has fired. Actions may include creating new conclusion, redraw old conclusion, etc. The inference engine controls the rule execution according to some strategy. Using the forward chaining strategy the condition parts of the rules are examined to see whether or not they are fulfilled. If so, a fulfilled rule is selected for execution. The execution of the action part of the rule causes new rules to be fulfilled and so on. Forward chaining systems are sometimes called data-driven. A backward chaining system tries to achieve a goal or verify a hypothesis be trying to fulfill rules that confirm the hypothesis. A goal could, e.g., be expressed as the need to compute a value of a certain object attribute in the knowledge base, and a hypothesis could be expressed, e.g., as a certain value of an object attribute that needs to be verified. If the goal not is immediately available in the knowledge base, the backward chainer tries to find rules whose actions establish the goal. One of these rules is selected and the conditions of this rule become new goals that must be established and so on.

Real-time applications

Expert systems were originally developed for consultative off-line applications were the non-expert user interacts with system in a question and answer dialogue where he provides the system with additional information and receives suggestions and conclusions. During the last years, however, real-time, on-line expert system applications have emerged as a promising area. The applications typically involves monitoring, diagnosis, scheduling, and control of complex technical systems of the kind found in the process industry, manufacturing industry, aerospace industry, or nuclear industry. Systems of this kind have a number of common features:

- The complexity of the systems is such that no single individual or small group of individuals can fully understand them.
- The operations and maintenance manuals may cover several tens of volumes. Maintenance of the documentation is a particular problem. It is difficult to access relevant and correct information speedily.
- Systems are continually changing and evolving since: the process and the operating environment changes; shortcomings in the original specifications come to light; bugs are discovered; and, technology advances.
- The rate of change means that old methods of training and retraining staff are no longer adequate.
- Different users of the systems need markedly different styles of interaction with the system.

- Speedy and accurate correction of faults is required. The hazards of slow or incorrect treatment are:
  - Faults not treated early enough may propagate catastrophically.
  - Dormant faults undetected or left untreated may greatly affect the overall reliability and maintainability of the system.
  - The existing control system may itself be prone to failure and thus may mask the true cause of misoperation.
  - Wrongly identified faults and consequential repair actions may make matters worse.
  - The high reliability of the systems gives problems. Some failures are so rare that it is difficult to ensure that maintenance staff are appropriately predisposed or equipped to handle them

In the majority of the applications expert systems are used as an operator assistant in order to enhance the operators problem solving capacity in hazardous, uncommon situations which the operators have little experience of or to assist him in stereotype, monotone operations. The expert system is connected on-line either directly to the process or via a conventional digital control system. The expert system reasons about the process and its behaviors on the basis of the incoming sensor measurements.

Some of the more common application types will be further described below.

Alarm analysis: A complex industrial process is normally equipped with numerous alarms. The alarms are often on a low descriptive level, e.g., *too low* or *too high*, and only give rudimentary information about the problem. A fault generating a primary alarm is often followed by numerous secondary alarms which obscure the original cause. Furthermore, due to different scan intervals alarms may be recorded in an incorrect order. An expert system for alarm analysis uses a description of the process, e.g., a process schematic, together with rules which describe causal relationships between measurements and alarms and between different alarms to sort out which alarm that was primary and which that were secondary.

Condition monitoring: A condition monitoring system can be seen as an on-line diagnosis system that monitors the process in order to reveal deviations from normal operating conditions. These deviations could be caused by slowly developing faults, badly tuned control equipment, worn-out process equipment, etc. The variations are typically quite slow and the normal way for the operator to detect them would be to study trend curves over several days or weeks.

Symptom-based diagnosis: An expert system for diagnosis is typically used handle intermittent failures and failures not causing low-level alarms. A diagnosis system is more focused on interaction with the user than an alarm analysis system. A symptom-based diagnosis system is based directly on the operators' heuristic knowledge about faults that can appear and what may have caused them. The symptoms and faults are directly represented as rules. A drawback with a purely symptom-based approach is that the knowledge is highly process specific. It is also impossible to detect unanticipated faults for which the operators have no solutions.

Model-based diagnosis: A model-based diagnosis systems is based on an explicit model of the process. The model could either describe the process, normal fault-free behavior or the different fault that may occur, a so called fault model. The models are typically described on the component level. For each process component, e.g., pump, heat exchanger, reactor, etc., a model is derived. The model for the entire process is then

derived from the component models and a process schematic that describes how the components are connected. The diagnosis procedure compares the measured process behavior with the model's expected behavior and uses the deviations to conclude the faults. The nature of the models might differ from numerical, differential equation models to qualitative, causal models that describe how process variables qualitatively influences each other.

Dynamic scheduling: Planning of a sequence of actions or operations and the time scheduling of these operations is found in many industries, e.g. the manufacturing industry and the steel industry. The scheduling problem concerns the allocation of over time of a finite set of resources in order to fulfill customer orders in a timely and cost-effective fashion. The scheduling problem is governed by a set of constraints. These are either hard constraints on, e.g., delivery times, that must be met, or soft constraints representing preferences that provide a basis for selection among possible choices. A dynamic scheduler must be able to reactively revise and maintain the schedule in response to changing conditions.

Scheduling problems have traditionally been solved by numerical, dynamic optimization methods. The drawback of these methods is that the mathematical modeling assumptions needed do not match the actual scheduling situation. Heuristic knowledge and soft constraints are difficult to include in the models and limits their practical use. Knowledge-based scheduling system concentrates on generating a practically useful schedule rather than the optimal one.

Fuzzy control: In fuzzy control, expert system techniques are used to mimic the operators' manual control strategy. The control actions are expressed as linguistic rules for how the control signal should be chosen in different situations, e.g.,

```
If the back-end temperature is somewhat low
then open slightly the exhaust gas fan damper.
```

The linguistic term, i.e., *somewhat low, slightly*, are defined as fuzzy quantities that are described by a membership function that assign a membership grade between 0 and 1 to a quantitative value.

The controller consists of a set of fuzzy control rules. In each sample, all the rules that fulfilled to some degree are weighted together to generate the resulting control signal. The indented applications for fuzzy control are control of complex processes for which either appropriate models do not exist or are inadequate, but where the human operators can manually control the process satisfactorily. The cases where fuzzy control has been used has all considered set-point control of non-linear, multi-variable processes.

Expert control: Expert control seeks to extend the range and functionality of conventional SISO controller by encoding general control knowledge and heuristics about auto-tuning, adaptation, and loop performance in a supervisory expert system. The controller consists of an "intelligent" combination of an expert system and a set of control, identification, and monitoring algorithms. The majority of the work in the field has so far concentrated on smart PID auto-tuners and heuristic safety jackets to self-tuning controllers.

Real-time aspects

Combining expert system techniques with conventional control software in on-line applications creates a set of very special problems that will be discussed here. As a result of this a special market has arisen for real-time expert systems. Several commercial systems are available on the market, e.g., G2, Cogsys, and R*Time. These systems are intended to be used as an add-on on top of the existing distributed control systems. There is also

activities going on that aim to extend the functionality of conventional control systems to make it possible to also include knowledge based applications.

The special real-time expert system problems are:

Nonmonotonicity: The expert system reason about a dynamic environment. Incoming sensor data, as well as inferred conclusions, do not remain static. Data are either not not durable and decay in validity with time, or they cease to be valid due to events which have changed the state of the system. In order to maintain a consistent view of the environment, the expert system must be able to automatically retract inferred facts. To retract facts is not allowed in predicate calculus which the underlying logic for most off-line expert systems. They are all monotonic in the sense that the set of inferred conclusions always grows.

One approach to handle the non-monotonicity is to record all the dependencies or justifications between sensor data and inferred conclusion in a dependency network. When things change the network is used to determine what should be retracted. This approach is used in the so called Truth Maintenance Systems. Another approach is to attach validity intervals to sensor data and conclusions. Here, all conclusions will eventually be retracted when their validity intervals expires as long as they are not inferred anew.

Asynchronous events: The expert system must be able to interrupt its reasoning in order accept input from unscheduled or asynchronous events. The system must be able to focus its attention on the most important event.

Temporal reasoning: Time is important in real-time systems. The system must be able to represent time and reason about past, present, and future events as well as the sequence in which the events occur.

Reasoning under time constraints: Reasoning under time constraints cover the problem where the system needs to come up with an answer within a given deadline. Furthermore, the best possible solution within that time is desired. The topic includes the problem of estimating the time needed for internal reasoning activities. Some measure of goodness of the solutions or the different reasoning mechanisms that may lead to a solution is also needed.

Several other real-time aspects are also important. The system must be able to cope with uncertain or missing sensor data. Interfaces must be provided to both conventional software and the external environment. Finally, the execution speed must be sufficiently high. Many consider this the crucial problem. In comparison with the other problems and in light of the rapid hardware development this is, however, probably the least difficult problem.

Many of the problems described are extremely difficult and very far from a solution. In other cases, however, practical approaches exist that to some degree solve the problem.

10

Computer Aided Control Engineering

M. Andersson

GOAL: To give a brief overview of tools and research issues in computer aided control engineering.

Even before computers were used for controlling processes, they were used to aid the design of control systems. Today, almost all control engineering and research in control theory is more or less computer aided. There is a wide range of available software useful in Computer Aided Control Engineering (CACE).

CACE software has developed over the years as computers have become more powerful and more accessible to control engineers. The early days type of CACE software was usually subroutine libraries most often written in FORTRAN. To solve a particular control design problem, a special purpose program had to be written, compiled, linked and executed. Today most CACE software are interactive programs that allow the user to enter commands and get immediate response. Some commonly used interactive CACE tools will be presented in this chapter. We will also take look some current results from a CACE research project at the Department of Automatic Control.

Section 10.2 contains a short overview of software for numeric and symbolic calculations commonly used in CACE. Section 10.3 is devoted to simulation which is a particular important technique in control design. A short discussion on user interaction is given in Section 10.4. Finally in Section 10.5, a new modeling language developed in a research project is presented.

10.1 Overview

Many modern design methods used in automatic control relies heavily on numerical computations performed by computers. Many design methods assume linear systems and results in linear controllers. In this case most computations are performed on matrix data. Some examples of common matrix computations are matrix inversion, eigenvalue computations and matrix factorization.

Algorithms for numerical computations on matrices is an active research area and the results are often used in control design. Reliable software implementing numerical rou-

tines is usually developed by experts. Fortunately, many standard numerical routines are implemented as subroutine packages and available to non-experts. EISPACK and LINPACK are two examples of such packages of reliable numerical subroutines implemented in FORTRAN. Matlab is an interactive program based on these subroutines. It is available for many different computers and it is, together with a similar program called Matrix-X, probably the most widely used CACE tool.

Even though development and implementation of fundamental numeric algorithms can be trusted the experts, it is important that the user of these routines has a basic understanding of the difficulties in order to avoid the pitfalls. To get accurate answers to posted numerical problems the following issues are important to consider.

- The problem should be well-conditioned. More formally this means that the answer to a slightly different problem (in some sense) should be close to the answer of the actual one.
- The algorithm used to solve the problem should be numerically stable. This means that round-off errors, that are always made when numbers are represented in computers, must not cause large errors in the solution.
- The software implementation of the used algorithm should be robust. For example, it should warn the user or produce satisfactory results also to slightly ill-conditioned problems.

Even when well behaved and reliable software is used, all computed results should be treated with a sound scepticism.

Of course there are numerical problems that are not naturally turned into basic matrix computations. Some examples of numerical techniques often used in control engineering are simulation, optimization and system identification. For these techniques there exists special purpose software. Simulation is a particular important example and there is a wide range of simulation software available on the market. Simulation will be discussed in more detail below.

Some difficulties with inaccuracy in numerical problem solving techniques can be overcome by solving the problems using algebraic manipulations. There are computer programs available that can carry out symbolic manipulations of mathematical expressions and formulas. The programs have a fairly good knowledge about the rules of algebra and of functions, series, integrals and derivatives. Symbolic manipulations can sometimes be used for deriving closed form solutions to problems, i.e, solutions that are represented explicitly as expressions in the original parameters. Different design parameter values can then, as a final step, be applied to the derived formulas. The solutions acquired by symbolic manipulations are often much more informative for the user than are the numbers obtained form a numerical routine. However, to many problems there exist no closed form solutions or the size of the computed expressions grows out of bounds. Two examples of powerful software products for symbolic algebra are MACSYMA and Maple.

10.2 Simulation

Simulation is a commonly used technique for investigating the behavior of complex dynamic systems. Simulation means experiments performed on a model of a system rather than on the system itself. There are many advantages of doing simulations instead of experiments on a real system. For example, in many cases real world experiments can not be done for safety reasons or for economical reasons. In a simulation it is easy to study the behavior of quantities that are not measurable in a real experiment. Very often, like in control design, simulation is used as a part of a design procedure to study behavior of

systems that do not yet exist. It is usually cost effective to try different design possibilities by simulation in order to avoid expensive redesigns of the real process.

Many analysis and design methods used in control design are based on a linear and time invariant model of a real world system. Even though most real systems are nonlinear and time varying, linear approximations are often used in the controller design. This is usually acceptable as long as the system is working close to the validity range of the approximation. Simulation is not limited to linear and time invariant models. To avoid unpleasant surprises when a controller is applied to the real system, simulation is often used as a part of the design process to verify that the controller works properly even outside its normal region of operation.

There are many different kinds of simulation techniques depending on what kinds of models that are used. Every model is an abstraction that describes only some aspects of a real world system, while it neglects many other aspects. Systems can be represented by continuous time models or sampled data models. These kinds of models are most common in control engineering and will be discussed more below. A rather different kind of model is a *discrete event* model that leads to a different kind of simulation technique. Discrete event models are, for example, used for representing customer–server and queue systems common in manufacturing and resource allocation.

Computer simulations can of course be performed by writing a program in some ordinary programming language like FORTRAN or Pascal. Such a simulation program contains the model as well as the numerical routines for performing the computations. This is not a very practical way in most cases, and therefore a number of special purpose simulation languages have been developed. A simulation language makes it possible to enter the models on a form that is closer to the way the model designer normally thinks about the systems. Some simulation languages, especially the early ones developed in the sixties, are using FORTRAN as an intermediate representation. That means the models are translated into FORTRAN subroutines that are then compiled and linked with a library of simulation routines. Modern simulation languages are often translated directly into a form that can be executed by the simulator. This makes it possible to create interactive simulation tools where it is easy to modify the models and where different simulation experiments can be performed without any time consuming compilations. Simnon is an example of a language and an interactive simulation tool for continuous time and sampled data models.

Continuous time and sampled data models are types of models most often used in control engineering. The former type will be discussed more below. Sampled data models are often used in computer controlled systems representing the behavior of a continuous time system that is sampled, i.e, observed at equally spaced time points.

Continuous time simulation

A continuous time model is based on ordinary differential equations (ODE) or partial differential equations (PDE). The state of an ODE model can be represented by a finite number of state variables. This is not possible for PDE models which are also called distributed parameter models. In contrast, ODE models are sometimes called lumped parameter models. PDE models result from systems involving heat transfer, fluid dynamics or flexible mechanical structures, for example. These kinds of models are not commonly used in control engineering. In many cases systems most accurately described by PDE's are approximated by an ODE model.

Most continuous time simulation programs accept ordinary differential equation models. A high order differential equation can always be turned into a set of first order differential equations. One commonly used form to represent such a model mathematically

is

$$\frac{dx}{dt} = f(x, u, p, t) \tag{10.1}$$

where x is a vector of unknown functions in time, u is a vector of known functions in time (input signals), p is a vector of constant parameters, t is time, and f is a known vector function. Simnon is an example of a simulation program where continuous time models are given on this form.

To simulate a system represented represented as (10.1) is the same as solving the equation for $x(t)$ for some interval in time t. Usually the initial state $x(t_0)$ is known. This is the same as integrating the function f from t_0 to t_1 and the procedure is called numerical integration.

There exists a large number of procedures for numerical integration. The simplest one is called the *Euler Method* or sometimes *Forward Euler*. In this method the derivative of x at time t is simply approximated by a forward difference:

$$\frac{dx}{dt} \approx \frac{x(t+h) - x(t)}{h}.$$

When this approximation is used we get

$$x(t+h) \approx x(t) + hf(x(t), u(t), p, t)$$

which can be used for computing points $x_0, x_1, \ldots$ that are close to the true solution at the time points $t_0, t_0 + h, t_0 + 2h, \ldots$. The accuracy of the computed solution depends on the step length h which has to be small enough.

The described method of integration is a one step first order method since it uses only the current step and one evaluation of the function f to compute the next step in the solution. There are many other integration methods which all have their different merits and drawbacks. A common class of integration methods is called Runge-Kutta methods which are all one step methods but of different orders.

Modeling

A difficult problem in simulation and control engineering is to obtain good and accurate models. One way to support the engineer in defining models is to provide a good simulation language. Very often models of large systems can be structured and separated into a set of interacting submodels. One way to support modeling is to provide libraries of commonly used submodels that can be reused. In a following section a modeling language that facilitates model structuring and reuse, is described.

10.3 User interaction

Many CACE tools in use today were developed in the seventies and in the early eighties when simple teletype terminals were the most common devices for user interaction. Interactive tools like Matlab and Simnon are based on a command language. The user enters a command line that is interpreted and executed when the user hits the return key. In Simnon there is one interactive command language and one language for describing the models. The models are defined and modified off-line, i.e, outside the Simnon program by an ordinary text editor.

When the user is working with an interactive tool it is very common that certain sequences of commands are issued over and over again. In this case it is convenient to be able to store a command sequence in a file and then executing the commands by just

calling that file. This is possible in Simnon where commands can be grouped together in a command procedure, called a macro, and stored in a file. Matlab has a similar facility called m-files. Command procedures and macro facilities are very convenient for documenting a particular design or experiment so that it can easily be repeated at a later time or by other persons. A common way of communicating new research results is to develop a special "tool box" of macros supporting, for example, a new control design method in Matlab.

New methods for computer interaction have emerged as the graphical capabilities of the computers have developed. The style of interaction that is normally used, for example on Macintosh computers, is called *direct manipulation*. The main idea is that entities and objects manipulated by the program and by the user have graphical representations, icons, that can be displayed on the screen. The user is then interacting with the objects by pointing at them with a mouse and by choosing commands from menus. This style of interaction is particular appealing for the unexperienced user. More and more CACE tools with direct manipulation type of interaction is emerging. Block diagram editors for structured models is an example were direct manipulation is particular convenient.

One problem with direct manipulation interfaces is the difficulty of saving sequences of user actions, such that they can be modified and repeated as command procedures and macros.

10.4 Omola

Omola is general language for representing models of dynamic systems. The language is based on ideas from object-oriented programming and the name is short for *Object-oriented Modeling LAnguage*.

Omola is one of the outcomes from a project in computer aided control engineering at the Department of Automatic Control in Lund. In this project it was realized that models are playing an essential role in engineering and in particular in the design of control systems. Most simulation languages and model representations used in various design tools are too specialized and unflexible to be used as a general modeling language. Omola has been designed to overcome these deficiencies.

Omola was designed with the following important objectives in mind:

- The language should support a number of mathematical and logical frameworks for representing model behavior. For example, differential algebraic equations, transfer functions, state space descriptions, discrete events and qualitative behavior.
- It should include concepts for structuring of large models, for example, hierarchical submodel decomposition.
- It should be modular in order to support reuse of parts of models in other models.
- It should be possible to include "redundant" information in models for the purpose of documentation and automatic consistency check.
- It should be generally useful as an input language for different control design tools and simulators. It should also be useful for model documentation and as a standardized exchange language between users and tools. That means, it must fit within an interactive CACE environment.

Omola is designed to describe structure and behavior of dynamic systems. However, it is based on a few very general concepts of object-oriented data structuring. This makes Omola generally useful as a data modeling language.

The basic entity that can be defined in Omola is called a *class*. A class defines a data type which has a name and a number of *attributes*. The attributes defines the properties

of the class and they can be ordinary named variables of a defined type (real, integer, string, etc.) or they can be other class definitions, so called *components*.

Classes are arranged in a hierarchy such that every class has one *super-class*. The inverse relation is called *subclass*. A class will inherit all attributes present in its super-class. An inherited attribute belongs to a class in the same way as if it was defined locally in the class. If a local attribute is defined with the same name as an inherited attribute, the local definition will override the inherited one.

The general form of an Omola class definition looks like:

```
<name> IS A <super class> WITH
  <body with local attribute definitions>
END
```

where the class body may contain other class definitions or variable definition on the following format.

```
<name> TYPE <type name> := <binding expression>
```

The binding expression is optional. It binds the variable to a specific value or an expression.

Model representation in Omola

We will now see how models of dynamic systems can be represented in Omola. There is a set of predefined classes in Omola that serve as super classes for the models and model components defined by the user. Some of the more important predefined super classes in Omola are `Model`, `Terminal` and `Parameter`.

A *model* is the main structural entity. A model contains a description of its interface to the environment, its dynamic or static behavior and its parameters. Models can be developed and tested, saved in libraries and reused as submodels in different contexts.

A model is an encapsulated module where the interaction with the environment is limited to certain variables called *terminals*. A model can have any number of terminals. Very often a model represents a physical component such as a pump, a valve or a regulator, and the terminals represents physical quantities like mass flow, electric voltage, etc. Models can have parameters to make them more flexible and adaptable to different circumstances. A parameter is a variable that can be changed by the model user but normally remains constant during a simulation.

Here follows an example of a simple model definition in Omola. It defines a tank model with two terminals and two parameters.

```
Tank IS A Model WITH
  terminals:
    inflow IS A Terminal;
    outflow IS A Terminal;
  parameters:
    tank_area   ISA Parameter WITH default := 5.0; END;
    outlet_area ISA Parameter WITH default := 0.05; END;
END
```

The tank model is defined as a subclass of Model with the terminals and parameters defined as local attributes. The terminals and parameters are component attributes, i.e., they are classes defined as subclasses of the predefined classes `Terminal` and `Parameter`. The parameters have default values that can be changed in a subclass of Tank or in an instance involved in a simulation.

The Tank model defines only the model interface and not the model behavior. We can use this simple tank model as a super class to several different tanks defined with different behaviors. For example, we can define a tank model with a non-linear behavior specified by two equations:

```
NonLinearTank IS A Tank WITH
  variables:
    level TYPE Real;
  equations:
    tank_area * dot(level) = inflow - outflow;
    outflow = outlet_area * sqrt(level);
END;
```

This new tank model will inherit the terminal and parameter attributes from Tank and add a variable and two equations. An advantage of separating the tank model into two different classes, one defining the interface and another defining the behavior, is that we can define alternative tank models with different behavior but with identical interfaces. When a tank is used as a part of large plant model, it is easy to exchange tank models with different behavior descriptions. This is a good example of how the object-oriented approach achieves both modularity and reusability of models.

The NonLinearTank was an example of a primitive model, i.e., its behavior was described by differential equations. Models can also get their behavior definition from a set of connected submodels. Such a model is called a *structured* model. For example, we can define a new model composed of two connected tanks:

```
TankSystem IS A Model WITH
  terminals:
    inlet  IS A Terminal;
    outlet IS A Terminal;
  submodels:
    tank1 IS A NonLinearTank;
    tank2 IS A NonLinearTank;
  connections:
    inlet AT tank1.inflow;
    tank1.outflow AT tank2.inflow;
    tank2.outflow AT outlet;
END
```

The tank system consists of terminals, submodels and connections. A connection is binary relation between two terminals indicating model interaction. The submodels are subclasses of the previously defined `NonLinearTank`.

We have seen Omola used for representing models in a modular way. Models can be defined as classes which can be specialized in various directions and used as components in other models. Also terminals can be structured in a similar way. The terminals used in the examples here have been on the very simplest form. We can also define terminals with additional attributes defining the physical quantity, unit of measure, range, etc. Interaction between model components involving a set of quantities can be represented by structured terminals. For example, a terminal modeling a pipe connection may have pressure, temperature and flow as component terminals.

An interactive environment

Omola is intended to be used together with tools that support modeling, simulation and control design. Block diagrams are convenient for displaying and manipulating structured models. One important tool is a block diagram editor which can be used for building and manipulating structured models graphically. In this case, Omola code can be generated automatically from the block diagram.

Another example where Omola is used as a general model language is a controller design tool that generates a controller on Omola format. This controller representation

can then be used as input to another tool that generates the actual real-time controller code automatically.

Omola can be viewed as a textual format for a model database in an environment of cooperative tools for control systems design. Different tools in the environment may use the models for different purposes. Here are some examples:

- Generating a graphical picture of the system structure, for example, a block diagram.
- Generating text descriptions of the system for documentation or user information.
- Generating special purpose code, for example, automatically generated regulator code.
- Generating standardized system descriptions in order to communicate with other control engineering packages.
- As input to various control design tools.

11

Bibliography

GOAL: To give references and suggestions for further reading.

Chapters 2 and 4 are reprinted from

WITTENMARK, B., K.ÅSTRÖM, and S.B. JØRGENSEN (1990): "Process Control," Kompendium.

Implementation issues for a larger class of controllers can be found in Chapter 15 in

K.J. ÅSTRÖM AND B. WITTENMARK (1990): *Computer Controlled Systems*, Prentice Hall, Englewood Cliffs N.J.

More on sequence control and GRAFCET can be found in the standard IEC 848 or in

GREPA (GROUPE EQUIPEMENT DE PRODUCTION AUTOMATISÉE RÉUNI À L'ADEPA) (1985): *Le Grafcet – de nouveaux concepts*, Cepadues - éditions, 111, rue Nicolas-Vauquelin, 31100 Toulouse, France.

Some references on robotics are

ASADA, H., and J.-J. E. SLOTINE (1986): *Robot Analysis and Control*, John Wiley & Sons, Inc., USA.

CRAIG, J. J. (1989): *Introduction to Robotics Mechanics and Control*, Second Edition, Addison-Wesley Publishing Company, Inc., USA.

SPONG, M. W., and M. VIDYASAGAR (1989): *Robot Dynamics and Control*, John Wiley & Sons, Inc., USA.

A general reference on real-time programming and a description of Modula-2 can be found in, respectively,

SILBERSCHATZ, A.,and J.L. PETERSON (1988): *Operating System Concepts*, Addison-Wesley.

WIRTH, N. (1985): *Programming in Modula-2*, Springer, New York, Berlin, Heidelberg, Tokyo.

Control systems have been implemented using computers since the early seventies at the Department of Automatic Control. The current direction using real-time kernels was initiated by Hilding Elmqvist. The first kernels developed are described in

ESSEBO, T., R. JOHANSSON, M. LENELLS, and L. NIELSEN (1980): "A facility for executing concurrent processes in Pascal," Report TFRT-7194, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ELMQVIST, H., S.-E. MATTSSON, and G. OLSSON (1981): *Datorer i Reglersystem, Realtidsprogrammering*, Institutionen för Reglerteknik, Lund.

The two references above describe real-time kernels in Pascal for LSI-11. The latter reference also includes a complete description of an implementation of a DDC-package (called an automation system in Chapter 3). The original kernel has been transferred to Modula-2 on IBM PC and on a Sun-VME system. The main person cultivating this tradition is Leif Andersson and he is responsible for the modules presented in Chapter 6. Additional related material can be found in

ANDERSSON, L. (1989): "Command Decoding in Modula-2," Report TFRT-7413, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ANDERSSON, L. (1989): "A Modula-2 Real-Time Scheduler," Report TFRT-7414, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ELMQVIST, H. (1985): "LICS–Language for Implementation of Control Systems," Report TFRT-3179, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Implementation issues have recently received an increased interest, e.g. in what is called hard real-time systems, and two recent publications are

BURNS, A., and A. WELLINGS (1990): *Real-Time Systems and their Programming Languages*, Addison-Wesley, Reading, Mass.

STANKOVIC, J.A., and K. RAMAMRITHAM (Eds.) (1988): *Hard Real-Time Systems*, IEEE Tutorial, IEEE Computer Society Press.

Chapter 7 is based on a course project in the course Computer Implementation of Control Systems. The project was supervised by Per Persson, who also wrote the first version of the chapter. The present version has been modified in collaboration. Chapter 8 is mainly written by Ola Dahl and is based his work

DAHL, O. (1989): "Generation of Structured Modula-2 Code From a Simnon System Description," Report TFRT-7416, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

DAHL, O. (1990): "SIM2DDC – User's Manual," Report TFRT-7443, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

The system has been used both for research and education

DAHL, O., and L. NIELSEN (1990): "Torque Limited Path Following by On-line Trajectory Time Scaling," *IEEE Trans. on Robotics and Automation*, **6**, 554–561.

NILSSON, B. (1990): "Computer Implementation of Control Systems, Laboratory Exercise in Process Control," Department of Automatic Control, Lund Institute of Technology, In Swedish.

Descriptions of Scheme, Simnon, and Matlab can be found in

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

ELMQVIST, H., K.J. ÅSTRÖM, T. SCHÖNTHAL and B. WITTENMARK (1990): *Simnon User's Guide for MS-DOS Computers*, SSPA Systems, Göteborg, Sweden.

MOLER, C., J. LITTLE and S. BANGERT (1987): *PRO-MATLAB User's Guide*, The MathWorks, Inc., Sherborn, MA, USA.

Section 9.1 is written by Tore Hägglund, Section 9.2 by Klas Nilsson, and Section 9.3 by Karl-Erik Årzén. Further material on the latter subject can be found in

ÅRZÉN, K-E. (1989): "An architecture for expert system based feedback control," *Automatica*, **25**, 6, 813–827.

LAFFEY, T.J., P.A. COX, J.L. SCHMIDT, S.M. KAO and J.Y. READ (1988): "Real-time knowledge-based systems," *AI Magazine*, **9**, 1, 27–45.

Chapter 10 is written by Mats Andersson, and more details on object oriented handling of dynamic systems are presented in

ANDERSSON, M. (1990): "An Object-Oriented Language for Model Representation," Licentiate Thesis TFRT-3208, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.