A Model of Process Interaction in Real-Time Distributed Computer Control Systems

Jack Errol Cohen

A thesis submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg, for the degree Doctor of Philosophy.

Johannesburg, 1991

Declaration

I declare that thesis is my own unaided work. It is being submitted for the degree of Doctor of Philosophy in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any other degree or examination in any other university.

2 day of SEPTEMBER 19 91

Abstract

Real-time computer control is characterized by the need for a high degree of interaction between some machine or physical process, its controlling computer, and the human operator. Recently there has been a trend towards the use of distributed real-time computer systems which potentially offer greater functional flexibility, better maintainability and better reliability than centralized systems. The increasing demands that are being placed on real-time computer control systems have highlighted the deficiencies of current heuristic design techniques and emphasised the need for solid theoretical design precepts.

The approach adopted in this thesis recognizes that the issues of sampling and data representation, real-time control, and information ownership and flow control are all central to any theory of real-time dynamic systems. The unique characteristics of computer control systems are used to develop a theory which is directly applicable to real-time control applications. The *p_var* model is derived which describes process interaction in real-time systems and an abstract representation of the properties of the physical plant, its environment and its control system. It is shown how this model provides a powerful framework for solving practical and theoretical problems in computer control.

Included in this theory is a treatment of sampling and data representation in computer control systems. The plant-control system boundary is examined in order to determine the role of front-end data acquisition subsystems. The theory incorporates a definition of real-time which does not rely on any notion of speed, and it establishes a quantitative link between plant dynamics and the validity lifetimes of property representations. A set of guidelines for real-time design is proposed,

and techniques are presented for ensuring state-conformity in the representation of Boolean-valued properties and for improving reliability and fault-tolerance in the representation of analog properties. The proposed model requires that each plant property be reflected in the computer system and that ownership of each property representation be assigned to a single, local process. The treatment of real-time information as quantities with limited temporal validity is a pervasive theme in this work.

The methodology used has allowed the issues affecting process interaction in real-time systems to be analyzed, and it has enabled the effects of the simultaneous application of different concepts to control systems to be qualified. Consequently it has possible to restate many of these concepts in a precise manner which expresses their applicability to computer control applications. The resulting theory and proposed model of process interaction result from the unification of the principles underlying real-time distributed computer control.

Acknowledgements

I am deeply grateful to my wife Carol Lynne for her encouragement, for the patience she has shown over the past few years, and for her help with the checking of this manuscript.

The assistance of my supervisor, Professor Ian MacLeod, has also been invaluable. His enthusiasm and interest in my research have allowed me to learn a great deal from him. Professor MacLeod also arranged for my employment during my studies and for the funding of my research.

I am also grateful to the Electrical Engineering Department of the University of the Witwatersrand, where I have been employed as a research assistant since 1988. The financial support of the Foundation for Research Development (FRD)—the national research funding body—is also gratefully acknowledged.

I would like to thank my friend and mentor, Dr Vernon Lun, for his valuable input. Many of the concepts presented in this thesis emerged as a result of our stimulating discussions. I am also indebted to Dr Brian Wigdorowitz, Mr Lawrence Levin and Mr Shaun Zagnoev, for their help.

Finally, I would like to thank my parents, parents-in-law and sister for their continual support and encouragement.

To

Carol Lynne,

Ruchama Leah

and

Yehuda Leib

Contents

List of Figures

Chapter 1

Introduction

1.1 A theory of process interaction

Programmable digital computers have been used in control applications for more than two decades. They allow greater flexibility in implementing different control strategies than do traditional analog controllers. Early computer control systems consisted of multiprocessing minicomputers which performed the acquisition of data from sensors, computation of control algorithms, and the communication of control actions back to actuators. The low cost of powerful microprocessors and effective inter-computer communication systems (networks) has changed the original scenario considerably. Distributed systems, consisting of multiple co-operating computers (which we call *processing modules*), communicating via message passing over high-speed local area networks, now provide cost-effective and practical platforms for implementing large scale industrial control systems. The potential advantages in using distributed computer systems in control applications have been well documented by Farber [1], and by Kramer, Sloman and MacGee [2]. In order for these advantages to be realized, it is imperative that software for distributed computer control systems (DCCSs) be able to overcome the problems introduced by resource distribution and real time operating constraints. Software must also be capable of dealing with failures of individual processing modules and communication links.

Little exists in the way of theory which describes the behaviour of real-time distributed computer control systems. Current design considerations are based largely on practical experience and heuristics which cannot be freely applied to all systems.

With increasing demands being placed on real-time control systems, the need for a theory describing the distributed control of real-time dynamic systems is becoming more apparent.

Other workers have focused on three main areas: the development of a software architecture for DCCSs (Kramer, Sloman and MacGee, [2]), the development of a sound set of programming primitives for interprocess communication (MacLeod [3,4] and Liskov [5]), and the development of techniques for improving reliability and tolerance of communication and processor faults (Kopetz [6,7] and MacLeod [8]). All of this work is aimed at improving the general reliability and programmability of computer control systems.

Despite the success achieved in much of this work, a single, all-encompassing theory of real-time dynamic systems and their control has not emerged. Distributed computer control systems represent a synthesis of technologies from several different disciplines. The theories of distributed computing and real-time systems must be combined with digital control theory, instrumentation, data sampling and data communications technology. Each of these disciplines impacts the application of other disciplines. Although each of these disciplines is well-understood in its own right, the result of combining all these concepts into a single synergistic theory has not been thoroughly researched and modelled.

The complexity of the synthesis of these technologies, and the absence of an all-encompassing theory, is clearly illustrated by two major problems which are typically encountered in the design and implementation of distributed computer control systems.

The first problem concerns the difficulty encountered in implementing mechanisms which support concurrency in distributed systems. The widespread availability of cheap microprocessors and interconnection systems has been accompanied by a move away from large and medium scale centralized multiprocessing systems towards distributed systems. This is particularly true in the field of computer control, as most industrial plants are geographically distributed, and are comprised of functional process units which are easily distinguished from one another. Distributed systems are therefore ideally suited to solving process control problems.

One of the major challenges in distributed computing has been to develop mech-

anisms which facilitate the interaction of concurrently executing tasks, that is, task synchronization and interprocess communication. In traditional centralized (uniprocessor) systems which support pseudo-concurrency, shared memory primitives such as semaphores [9,10,11] and monitors [12] are used for interprocess communication and synchronization. Distributed systems support true concurrency, and each processing module has its own local memory. The physical architecture of distributed computer systems invariably implies that no shared memory is available for interprocess communication. Message-passing protocols which support interprocess communication in distributed systems are therefore required. Computer scientists, who have been addressing the problem of interprocess communication in generalized distributed systems, have proposed various solutions to the problem. Many of these solutions are based on assumptions which may not be valid in computer control applications. Careful examination of many of the proposed solutions shows that in most instances they do not *solve* the problem at all. Lamport [13,15,14] discusses these proposed solutions. He concludes that many of the proposed solutions merely replace a high level problem with a lower level one. We refer to the difficulties involved in providing support for interprocess communication and synchronization in DCCSs as the *interprocess communication problem.*

Another problem has arisen with the availability of cheap, powerful microprocessors. Over the past few decades, advanced numerical and statistical control algorithms have been developed. Whilst this body of modern control theory may have had some practical spin-offs, its value has been largely academic. This is primarily due to the computational intensity of the algorithms. The previous generation of 8 (and 16) bit microprocessors was not powerful enough to perform the required computation in real-time on-line situations. Newer computer technology however, makes the application of modern control techniques feasible [16]. Software is becoming increasingly complex, and this makes it difficult to produce systems which exhibit a high degree of programmability, maintainability and reliability. Consequently, system design and integration has become a most formidable task. The primary solution to this problem is a methodology which facilitates the successful decomposition of a complex system into a number of simpler, more manageable functional units. Many methods for system decomposition have been proposed, and the advantages and dis-

advantages of each is currently the subject of much debate [17,18,19,20,21,23]. Most of these techniques are not aimed at real-time control applications. The techniques which do incorporate methods for dealing with time-constraints are essentially software system design techniques, and they do not relate to the physical structure of systems external to the computer (Gomaa, [22]). We refer to the decomposition of a large computer control system into a number of concurrent distributed processes as the *decomposition problem*.

Although these two problems are usually dealt with in isolation, they both express the difficulty that is experienced in modelling process interaction in real-time distributed systems. The IPC problem can be described as the difficulty of trying to ensure semantically correct behaviour of multiple communicating processes executing concurrently on different processing modules in a real-time distributed system. The decomposition problem refers to the difficulty of decomposing a complex system into multiple processes which may, or may not, execute concurrently in a centralized or distributed system.

We believe that it will only be possible to deal with these and other problems in the practical and theoretical domains of real-time distributed control, when a consistent, unified theory, which describes process interaction in real-time distributed computer control systems, is developed [25].

This thesis deals with the development of a theory of real-time dynamic systems and the computer control of such systems. The need for such a theory has been explicitly stated by Stankovic [24] in his survey of real-time systems technology. *"Real-time computing is a wide-open research area of intellectually challenging computer science problems with direct payoffs to current technology. But results have been few, and designers presently have little that would enable them to handle the timing constraints of real-time systems effectively. Furthermore, not enough emphasis is being placed on building the proper scientific underpinnings to achieve the needed results."*

He also notes the difficulty involved in developing the required theory. *"To develop a scientific underpinning for real-time systems, we face the difficult scientific challenge of creating a set of unified theories and technologies which will allow us to reason about correctness, timeliness, and reliability at each level of abstraction, and*

to combine the results of each level into results for the integrated system."

1.2 Scope of the research

The aim of this thesis is:

- to identify and analyze the issues affecting process interaction in real-time distributed computer control systems,
- to qualify the effects that different concepts have on one another when they are simultaneously applied to control applications,
- to restate these concepts in a precise manner which expresses their applicability to computer control applications (where necessary),
- to develop a theory of real-time dynamic systems and the computer control of such systems,
- to develop a model for the interaction of processes in real-time distributed computer control systems from this theory,
- to show how this model, and the theory, can be applied to problems in both the practical and theoretical domains of real-time distributed computer control.

No attempt is made to develop any particular interprocess communication primitive or real-time system analysis, design and specification tool. The emphasis is placed on developing a theory which could be used to solve any problem in real-time distributed computer control.

To define the scope of research and the contribution of the thesis, it is appropriate to examine some of the issues one would expect to influence process interaction in real-time distributed computer control systems. Therefore, the following sections present motivations for pursuing this line of research. We then show how the concepts of sampling and data representation, real-time control, and data ownership and flow-control can be unified, leading to a theory for the computer control of real-time dynamic systems.

The approach followed in this study is to make use of the operating constraints and architectural and configurational limitations pertaining to computer control

systems to develop a theory of real-time dynamic systems and a model for process interaction in real-time distributed computer control systems.

1.2.1 Quantification of the state of the plant

A computer control system is required to quantify the state of a physical plant and represent this state in the computer. A control algorithm then calculates the control actions which are required to drive the plant from this state to the desired state. These control actions are then applied to the physical plant by means of actuators. In a computer controlled system, the communication links which are used to convey information between the plant and the control system, and between processing modules, cannot be assumed to be reliable [4,26,27]. The nature of the sampling operation [28] and the physical characteristics of sensors and data acquisition equipment [29] prevent the exact quantification of the physical properties of a plant. Despite these problems, the correctness of information in both the value and time domains must be ensured. An analysis of these problems and their effect on sampling and data representation techniques is therefore essential if the interaction of a physical plant and its control system is to be modelled effectively.

1.2.2 Real-time response

Computer control systems are required to provide a real-time response to conditions and stimuli from within both the plant and the control system. The term "real-time" has been the source of much confusion in engineering and transaction-processing applications. Ramamritham [30], Stankovic [24] and Kopetz [27] all emphasize the time-critical nature of process control. Nevertheless, most existing definitions of the term "real-time" are unsatisfactory. Real-time constraints can be used to specify temporal performance requirements other than deadlines for the completion of operations. These issues need clarification if management of time is to be incorporated into distributed control systems.

Although incorporating the management of time into a distributed system represents an added dimension of complexity, once it has been accomplished, correctness can be enforced and judged through the use of temporal mechanisms. Kopetz [7,6] and MacLeod [4] have examined the use of time-stamps to provide data with a lim-

ited lifetime (validity time). They note that the reliability and fault tolerance of a DCCS can be improved by attaching some sort of temporal significance to each message. Kopetz and MacLeod conclude that information must be both *correct* and *timely*.

1.2.3 Flow-control

In distributed computer systems and transaction processing systems, the sender and receiver are capable of exercising explicit information flow control over one another by means of a PAR (positive acknowledgement and re-transmission on error), or some other end-to-end protocol, or by buffering messages. Although the use of PAR protocols in computer control systems has been questioned (and discounted) by LoLann [26], Kopetz [27], MacLeod [4] and others, a thorough treatment of the use of well-accepted data communications flow-control techniques in control applications is not available.

Control applications differ from other distributed computing applications in that the flow of information from the plant to the control system cannot be regulated by the control system. In other distributed computing applications, a computer which is receiving information may block the sender if it is unable to buffer additional information. In control applications, the sender is usually the physical plant, which "sends" information, regardless of whether the receiver can cope with the flow of data. The assumptions made regarding flow control in generalized distributed systems are not necessarily valid for flow control in distributed computer control systems. This is a further constraint which applies to control systems.

1.3 Summary of the thesis

The thesis commences in chapter 2 with a review of distributed systems theory. Issues which influence process interaction in distributed systems are highlighted, and the relevance of these issues in computer control applications is discussed.

In chapter 3 a detailed analysis of interprocess communication in distributed systems is presented. The development of interprocess communication is traced from shared memory techniques, which were first used in early centralized multiprocessing systems, through to the most recent distributed message-passing proposals.

In chapter 4 principles of sampling and data representation are analysed. A comparison of various sampling methods and data representation techniques, and their use in computer control applications, is given. The philosophical question of where to draw the plant-control system boundary is addressed in this chapter. We also develop the *property* and *p_var* concepts, on which our model of process interaction and theory of real-time computer control are based.

Chapter 5 takes an in-depth look at real-time systems, and at the incorporation of time-management facilities into distributed systems. A number of guidelines and techniques which can be used to ensure correct real-time system behaviour are proposed, and the relationship that exists between message validity times and the dynamic behaviour of the physical plant is examined. A distinction is made between real-time computer systems and real-time control systems. Temporal and state-conformity problems which are encountered in sampling both analog and Boolean-valued properties are examined. The use of real-time as a metric of temporal progression is discussed, and the difficulties which are encountered in the synchronization of clocks in a distributed system are described. These concepts are then integrated into the *p_var* model.

Chapter 6 examines the use of different flow-control techniques in computer control applications. An analysis of the relationship that exists between the properties of the physical plant and their data structure and message representation in the distributed computer system is provided. Assumptions about the ownership of data based on this relationship are stated and related to theories of process interaction which have been postulated by computer scientists. The results of these studies are then incorporated into our theory.

Chapter 7 presents a summary of our model for process interaction in real-time distributed computer control systems. This model is an abstraction of our theory of real-time dynamic systems and their control.

In chapter 8 case studies are presented which show how both our model and theory can be applied to both practical and theoretical problems in distributed computer control. It is shown how the model can be used to develop a set of interprocess communication primitives, and the theory is used to analyse the state conformity problem which arises in distributed real-time systems.

The concluding chapter summarises the contributions of this study to the field of real-time distributed computer control. The significance of these contributions is discussed. Limitations and caveats of the model and theory are mentioned. Extensions to the model are proposed and recommendations regarding further which will further improve our understanding of process interaction in computer control systems are made.

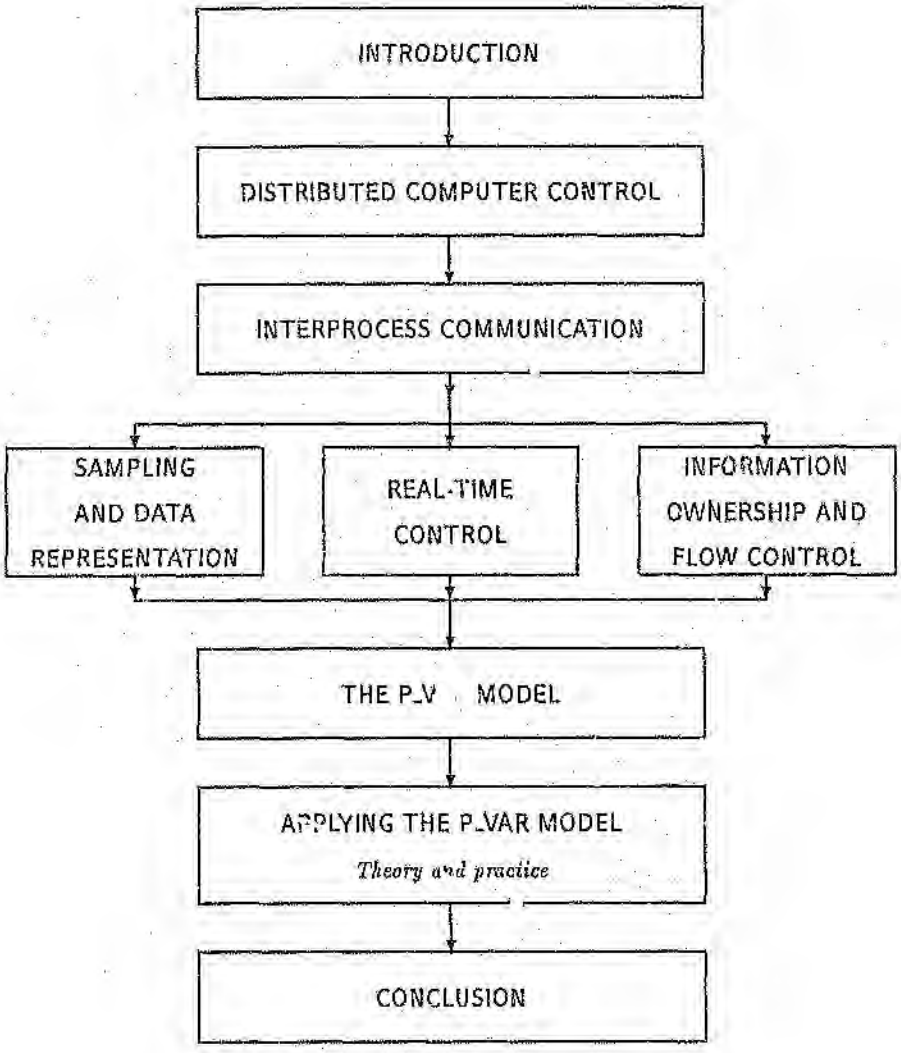


Figure 1.1: Outline of the thesis

Chapter 2

Distributed Computer Control

2.1 Introduction

In this chapter we introduce the theory of distributed computer systems and examine the application of this theory to computer control systems. Terminology and concepts relevant to distributed systems are introduced, and a brief overview of some of the central issues in distributed computing is given. Neither the list of topics nor the discussion of these topics is meant to be exhaustive. Only issues which impinge on the contents of subsequent chapters are covered. More comprehensive discussions of general issues arising in distributed computing can be found in [31,32,33,34].

We examine the following issues in this chapter:

- Architecture
- The ISO's Open Systems Interconnect model
- The transmission medium
- Topology
- Availability and reliability
- Fault tolerance
- Redundancy
- Task migration
- Synchronization of distributed clocks

A discussion of interprocess communication in distributed systems has purposefully been omitted from this chapter. Interprocess communication characterizes process interaction in distributed systems, and it is necessary to examine the subject in detail. Chapter 3 is therefore devoted to a study of interprocess communication. The issues listed above are discussed in most texts which deal with distributed computing, however in this chapter they are analyzed from a control engineering perspective.

The second part of this chapter begins with the development of a simple conceptual model of a computer control system. The issues discussed in the first part of the chapter are then examined in the context of the model.

2.2 Distributed systems

2.2.1 Architecture

Processing modules

A distributed computer system consists of a set of autonomous but co-operating computers which may communicate with one another only by exchanging messages over a local area network subsystem. We call these computers *processing modules*. There is no shared memory available for inter-computer communication in such a system.

Each processing module is capable of executing a process or set of processes in the system. A *process* is defined as a sequence of events or steps which aims to achieve some purpose which is specified at a chosen level of abstraction [26]. Hence a process may consist of other processes or be a component of another process. A sequential program which executes in a computer constitutes a process [35,26]. A process is normally cyclical or non-terminating.

A system composed of multiple microprocessors which share a common physical memory store cannot be regarded as a distributed system. A microprocessor performs operations on data. Since both the sequence of operations and the data are stored in memory, the sharing of memory implies that no microprocessor would be functional in the event of a memory failure. Thus we say that multiprocessor systems which share memory are centralized but allow for parallel processing. When each

microprocessor has its own local memory (we call this unit a *processing module*), it becomes possible for the processing module to function autonomously. Although a fully functional distributed system requires that several processing modules communicate with one another, each processing module is nevertheless an autonomous, fully functional computer in its own right.

Communication subsystem

Processing modules in a distributed system communicate with one another by means of some interconnection system. For this reason, each processing module is equipped with an interface to the interconnection system. The physical transmission medium could be twisted pair, co-axial cable, or fibre optic cable. Processing modules can be arranged in a ring, star, bus or hybrid topology. In DCCS applications, we advocate the use of either ring or bus topologies, because they offer increased reliability. In this thesis we are not concerned with either the physical aspects of network communication or with the provision of a low-level datagram service. It is merely assumed that an unreliable datagram service is operating over unreliable communication links. These subjects are discussed briefly later in the chapter, but a more comprehensive treatment can be found in Gee [36] and in Tannenbaum [31].

Interaction of processing modules

In a distributed system, the autonomy of the processing modules affects the fault tolerance and reliability of the entire system. In multiprocessor computer systems which are used in parallel computing applications, the microprocessors exhibit little autonomy. They are said to be *tightly coupled*. This tight coupling is usually manifest in hardware in the form of a parallel bus interconnection system which has precise timing specifications, bus parity and other features. In tightly-coupled systems, the microprocessors (or processing modules) are prevented from having any significant autonomy by the hardware architecture. Autonomy of processing modules depends on the degree of interaction between the distributed processes and the coupling of the processing modules.

Loose coupling and reduced process interaction contribute to autonomy of processing modules. Decoupling the processing modules in a distributed system does

not necessarily mean that the interaction between processing modules *has* to be low. In a loosely coupled distributed system there may be no need for processes executing on a different processing modules to communicate with one another. Such a system could be implemented by a number of computers not connected to one another. In such a system there is no interaction between processes, and the processing modules are functioning autonomously (despite the fact that they are interconnected).

On the other hand, however, the processes executing on different processing mod- may be required to exchange data and synchronize with one another frequently there is this high degree of process interaction, the processes are said to be ting with little autonomy. This is despite the fact that physically the system is loosely coupled.

Thus the coupling of processing modules in a distributed system gives no indication of the process interaction in the system. Processing modules are only said to function with a degree of autonomy when they are configured in a loosely-coupled system in which the degree of process interaction is low.

2.2.2 Networking

Networking in loosely-coupled distributed computer systems is usually based on packet-switching technology [3,4,26]. (This is as compared to other networking systems which may use circuit-switching, and with closely-coupled systems which are often based on a parallel bus.) Wide area networks such as those used in commercial systems enable data to be exchanged between heterogeneous systems which may even be located on different continents. In such systems, data usually has to pass through gateways, and it may have to be converted from one format to another many times. These systems usually introduce non-deterministic time delays, and they are seldom owned by one organisation. Local area networking is a technology which is widely used in office-automation environments. As is the case with wide-area networks, distributed systems use networks to distribute processing, databases and resources. LANs are usually owned by single organisations, and their extent is usually limited to a distance of a few hundred metres.

The Open Systems Interconnect model

In the late 1970s it became obvious that if the full benefit of computer networking was to be realised, then international standards would be required. With this in mind, the International Standards Organisation set about developing standards for the interconnection of open systems. The resulting set of standards was formally adopted in 1983 [38]. The Open Systems Interconnect (OSI) model uses a layered architecture to break up the interconnection problem into more manageable units. The OSI model consists of seven layers, and protocols are developed to conform to the architecture of each individual layer. Thus any system which conforms to the OSI model inevitably uses a hierarchy of inter-layer protocols. Whilst this architecture is certainly flexible, its suitability to certain applications has been questioned [31,4,39] (see section 2.3.6).

Although our research is at a more fundamental and theoretical level, we have decided to include details of physical network implementation. Since we are trying to develop an engineering solution to an engineering problem, we feel that it is important to describe the physical systems underlying our proposed protocols and primitives. Hardware and software components co-operate with one another to achieve a single objective. A detailed discussion of network hardware implementation and of the provision of a low-level datagram service is, however, beyond the scope of this research.

The transmission medium

The physical transmission medium is not part of the OSI model; rather it is the hardware to which the Physical Layer (lowest layer of the OSI model) relates directly. The transmission medium consists of transmission cables, repeaters, and interface hardware. (In active topologies the interface hardware, besides being a component of the processing module, also forms an integral part of the network.)

Twisted pairs, co-axial cables and optical fibres are all commonly used transmission cables. Twisted pairs typically support data rate of up to 10 Mbps. Cheap network implementations are possible with twisted pair cables, as the cabling, repeaters and interface hardware are inexpensive. Co-axial cables also offer cheap network implementation possibilities; however they support higher transmission rates

and they provide lower signal attenuation and high noise immunity. Optical fibres can support data rates of up to 50 Mbps, and signal attenuation is much lower than with either twisted pair or co-axial cable. Optical fibre is essentially a one-way transmission medium, making it ideal for use in ring topologies. Repeaters, connectors and optical fibres are expensive to produce and difficult to install.

Topology

The physical arrangement of processing modules on the network is referred to as the network topology. Three common topologies are the star, ring and bus topologies. In star topologies, routing is performed by means of circuit switching. Expansion costs are low and routing algorithms are simple. The communications processing that has to be performed by each processing module is minimal, as all the data on the communications link is destined for the processing module. Poor reliability is one of the major drawbacks of a star topology. Whenever the central routing switch (or hub) fails, the entire network fails.

In a ring topology, each node is connected to each of its two neighbouring nodes by a unidirectional link, and so communication only occurs in one direction around the loop. Routing algorithms for ring topologies are trivial, but the possibility of several nodes transmitting data simultaneously makes media access control a non-trivial problem. In ring topologies the repeaters form an integral part of the transmission system, and so the failure of a single repeater amounts to the failure of the entire network.

Bus topologies use a single linear transmission medium. All processing modules are attached to this bus, and whenever a single processing module transmits, the data is automatically available to all other processing modules on the network. Media access control is also difficult in bus topologies. Bus topologies have several significant advantages. The repeaters and interfaces are not part of the network, and reliability is much better because of this. Bus topology implementations minimize the required length of cable, and adding a new processing module is a trivial procedure. Both ring and bus topologies use packet-switching for message delivery. A more detailed discussion of network topologies and transmission media can be found in Gee [30] and in Tannenbaum [31].

2.2.3 Long-term operation

Distributed computer systems are required to satisfy a number of stringent requirements. One of these requirements is that the system be capable of providing long-term service. Distributed systems facilitate increased reliability and availability by allowing the use of fault-tolerant and redundancy techniques.

Reliability

The reliability of a system is defined as the probability that a system will function according to specification under specified environmental conditions for the entire duration of specified time of operation [37].

Availability

System availability is the probability that the system is functioning according to specification at time t. Under steady-state conditions, availability can be understood to refer to the average fraction of time during which a system meets its performance specifications under specified environmental conditions if maintenance is performed in accordance with the specified procedures [37].

Fault-prevention

Fault-prevention is an approach which is used to improve system reliability by attempting to ensure that faults will not arise during system operation [37].

Fault-tolerance

Fault-tolerance is an approach which is used to improve system reliability by attempting to cope with (or eliminate) faults which arise during system operation [37].

Redundancy

Redundancy is a mechanism which improves fault-tolerance by incorporating spare components which replicate the functionality of existing components. Redundant systems can be classified into two broad categories [37]:-

Standby redundancy - The secondary component begins operating if the primary fails. This technique requires detection of the failure of the primary component and the execution of a "switch-over" procedure to activate the secondary component.

Active redundancy - Where all redundant components are fully operational at all times.

These techniques are extremely relevant in applications where long-term uninterrupted operation is important. They are seldom employed in classical general purpose distributed computer systems which mask resource distribution from the user [32,34], because any processing module can perform the function of any other. Hence the only effect of the failure of a single processing module will be a degradation in performance of the system as a whole. This is clearly not the case with distributed computer control systems, where processing modules perform a fixed function, and where long-term uninterrupted operation is of paramount importance [7].

2.2.4 Task migration

Many distributed systems aim to mask the fact that processing power and other resources are distributed (e.g. The V Kernel [32]). The distributed operating system automatically allocates processes to different processing modules. Failure of a single processing module is detected by the operating system, and processes are no longer allocated to that processing module. The user could be quite unaware of the failure of a processing module in the system, unless he detects a slight deterioration in system performance due to the decrease in available processing power. Tasks may also be migrated from one processing module to another in order to balance computing load [40,31], to optimize resource allocation, or to increase throughput.

2.2.5 Synchronization of clocks

In systems of co-operating processes there is usually a need to synchronize processes and to order events which occur in the system [41]. Processes may delay for a fixed period of time, or they may be suspended and restarted by certain temporal conditions [4,41,27]. For this to be possible, all processing modules must have time references which are synchronized with one another to within a specified accuracy.

A similar problem exists in uniprocessor multiprocessing systems, except that it is considerably simplified by the availability of shared memory and a single real-time clock which can be read by all processes. One of the major difficulties encountered in the design of distributed systems is the synchronization of clocks over a network. In real-time systems, clock synchronization is even more critical. In subsequent chapters we explore this subject in more detail, as it is fundamental to our research. One point which must be raised at this stage is the non-deterministic time delay which results from the transmission of data over a network [26]. Non-deterministic media access protocols, buffering of messages, and varying network traffic patterns all introduce elements of non-determinism into the message transmission delay. Centralized systems may also exhibit non-deterministic delays due to various scheduling algorithms which may be in use. Mechanisms do exist for reducing these effects. (For example, software interrupts.) The consequences of non-deterministic delays complicate the design of real-time systems [24] — this is discussed in chapter 5. Non-deterministic delays can often be tolerated in non-real-time systems.

2.3 Control systems

In order to specify interprocess communication requirements for control systems, the issues raised above must be carefully analyzed in the context of the physical plant and its control system. We begin by building up a simple model which describes plant–control system interaction. This model is developed further in chapter 4, however it is used in this chapter to introduce concepts and terminology which are used in subsequent analysis. This model is used to highlight the differences between distributed computer *control systems* and other distributed computer systems.

2.3.1 Model of a computer controlled system

A physical system is composed of one or more subsystems which co-operate or interact with one another. In control terminology, this system is called the *plant*. The plant's dynamic characteristics and environment govern its behaviour. For existing control techniques to be of use, it must be possible to model the plant's behaviour in some way, either qualitatively or quantitatively. The factors which cause the plant to respond can be classified as either *inputs* or *disturbances*. Inputs are intentionally

manipulated to obtain a response from the plant, whereas disturbances are "unwanted inputs" which cause the plant to "misbehave." The measurable quantities and variables which constitute the plant response are referred to as *outputs*. A control system measures the state of the plant by monitoring the outputs. It calculates according to some algorithm, how the inputs must be manipulated if the plant is to be driven to the desired state (*set-point*). These manipulations of inputs are then applied to the plant as *control actions*. The *state* of the plant refers to the set of variable properties of physical objects which together completely describe the plant at some point in time. The control system measures plant outputs by means of *sensors*, and applies inputs by means of *actuators*.

Although the plant environment influences the behaviour of the plant, it is not controlled by the control system. Its effect must nevertheless be modelled so that the control system can be used to compensate for environmental effects.

2.3.2 Classical computer control systems

Classical computer control systems are based on centralized multiprocessing computers. The computer typically executes three, or possibly four , concurrent processes, namely [42]:

1. A data acquisition process which obtains data directly from the sensors or data acquisition hardware, validates, scales and filters it as required, and makes it available to other processes.

2. A control algorithm process which reads the measured data and uses it, together with a set of desired plant states, p, to compute the inputs required to drive the plant to p.

3. An actuator process which communicates the control signals to the plant actuators. This process takes care of communication with the actuator hardware and it implements the required protocol.

4. A fourth process might be used to provide supervisory (and possibly user-interface) functions. It may display measured data and control actions on a screen, or it may log these values to disk or printer.

2.3.3 Geographical distribution of the physical plant

Process plants are inherently geographically distributed. An industrial process or automation system may consist of a series of operations which are performed sequentially, and where workpieces (products) are automatically transported from one work sight to another by means of some conveyancing system. Other plants may involve several centralized operations which are all carried out simultaneously. Whatever the case, the subsystems (components) of the physical plant are separate entities which could be located even several hundred metres from one another. Sensors and actuators must however be connected directly to the physical subsystems with which they wish to communicate, and hence they are geographically distributed. In the centralized computer control systems described above, sensors and actuators are geographically distributed, yet they are connected to a centralized computer. Distribution of control implies a devolution of processing power and intelligence down to the sensor and actuator level. In a horizontally-organized system (no hierarchy), this devolution of processing power and intelligence is complete, and there does not exist any centralized computer or resources. Distribution of control is desirable because physical plants and plant interfacing systems (sensors and actuators) are inherently distributed.

2.3.4 The plant-control system boundary

One issue which arises in distributed computer control systems theory is the importance of distinguishing between the plant and the control system. In centralized systems it was understood that the central computer was the control system, and that the system to which the actuators and sensors connected constituted the plant. This definition is complicated and becomes inadequate when self-regulating and "intelligent" plant components are used. Some of these components may have on-board microprocessors which they use to monitor and regulate their own states. Components such as these would inevitably represent elements of distributed control in centralized systems. Distributed control theory encourages the localization of subsystem control, making it difficult to distinguish between the plant and the control system.

It is clear that separating a distributed computer-controlled system into "plant"

and "control system" components can be a *non-trivial task*. A careful analysis of modern computer controlled systems indicates that it may be unnecessary for a clear distinction to be made. The designer of a self-regulating component with an on-board microprocessor will consider this component as a fully-functional physical plant with its own local control system. When this component is embedded in a large plant, it simply becomes part of the larger plant. Control of the larger plant does not entail the low-level control of the self-regulating component's "plant". The component's on-board controller is considered part of the larger plant. Control systems are thus nested within larger plants. This concept of *nested control* is in accordance with MacLeod's and Kopetz's cluster model [4,7], where each cluster views the rest of the plant plus the control system as its environment. Thus a logical boundary between the plant and control system can be defined at any chosen level. In our analysis, we view the plant as an independent open-loop subsystem which may or may not meet its performance specifications. We view the control system as a subsystem which is attached to the plant in to improve its performance so that it meets or betters its performance specifications. The role of the front-end data acquisition subsystem which is used to convey information about the plant to the control system is discussed in chapter 4.

2.3.5 Architecture

The cluster concept and nesting of control allows process interaction to be reduced. Processing modules which interact closely with one another are grouped together, so that the interaction between groups is minimized. The clustering policy helps to reduce the overall process interaction. Geographical distribution of processes and automation plants means that processing modules are loosely coupled. These two factors enable processing modules in distributed computer control systems to function with a high degree of autonomy.

2.3.6 Networking

Several researchers have noted the inapplicability of the ISO's OSI model to LAN-based distributed systems (Tannenbaum [31] and MacLeod, [4]). Their reasoning is that the layered protocols of the OSI model makes interaction between processing

modules slow and cumbersome. If this is indeed so, then the OSI model is even less applicable to computer control systems, which are required to meet real-time constraints. This has been stated by MacLeod [4], Kopetz [27] and Kirk [39]. (Real-time constraints are discussed in chapter 5.)

Whilst twisted-pair, co-axial and fibre-optic transmission media all have their place in distributed computer control, the bus topology has certain advantages over other topologies. A shared linear bus is passive and minimizes cable requirements. Media access control is a non-trivial problem in shared-bus networks. The debate as to whether token-passing or CSMA/CD protocols are best suited to computer control applications is on-going. These protocols are discussed in detail in [43].

2.3.7 Long-term operation

In computer control systems, plant interfacing hardware is usually connected to specific processing modules. Distributed computer control systems therefore differ from other distributed systems in that processes are assigned to fixed processing modules before system initialization. This places more demands on system availability. A general distributed computer system which performs dynamic assignment of tasks to processing modules in order to mask resource distribution from the user may only require that M-out-of-N processing modules be operational. A DCCS, on the other hand, requires that all N-out-of-N processing modules be operational since they all perform specific tasks. One way of getting around this problem is to use redundant processing modules. The fixed processor/processing module configuration makes it unnecessary to provide facilities for dynamic task migration – it makes no sense to run an actuator process on a processing module which is not connected to actuator hardware [2,44] (see chapters 4 and 6).

The physical environment which control systems are required to operate in makes demands on system reliability. Industrial conditions can cause physical damage to the transmission medium, and so dual and triple redundancy is often used in network transmission cables. The presence of intense electromagnetic fields can cause transmitted data to become corrupted. Protocols which support effective recovery from error conditions form an essential part of any distributed computer system, but DCCSs are invariably required to meet hard real-time constraints. These real-

time constraints imply that any fault-tolerant error recovery protocol which enforces retransmission-on-error would either violate restrictions that the system imposes on flow-control [45,1,27] (see chapter 6), or it would incur too much communication overhead (or both).

2.4 Conclusion

This chapter has reviewed issues in the theory of distributed computing which are relevant to our research. The architecture of distributed computer systems was described, and an introduction to physical network implementations was given. Reliability and availability were discussed and the concept of task migration was introduced. The problem of synchronizing the clocks of processing modules in a distributed system was outlined.

In the second half of the chapter, a simple model of a real-time distributed computer control system was developed. The issues raised in the first half of the chapter were then analyzed in the context of this model. It was concluded that a distributed computer control system should consist of a number of nearly-autonomous processing modules which communicate via message-passing over a local area network. This network should be loosely coupled, and process interaction should be minimized as far as possible. This is in accordance with the theories of nested control and the clustering of processing modules. Several important conclusions regarding real-time distributed computer control were reached. These include:

- The provision of operating system facilities for dynamic task migration is not required. Processing modules in distributed computer control systems are usually connected to specialized hardware, and therefore process-processing module configurations are fixed.

- Fault-tolerance improvement mechanisms, such as the incorporation of redundant processing modules can help to improve the reliability and availibility of computer control systems.

- The use of a layered protocol system such as the OSI model is not ideal for use in real-time distributed computer control applications.

- Protocols which enforce retransmission-on-error may violate principles of flow-control in real-time control applications.

Chapter 3

Interprocess communication

3.1 Introduction

Process interaction in distributed computer systems is characterised by interprocess communication. Various aspects of interprocess communication are examined in order to provide some insight into the issues which affect process interaction.

The need for interprocess communication in distributed systems is examined. Various interprocess communication proposals which have been put forward since Dijkstra's first solution of 1965 [10] are then analyzed and compared with one another. After highlighting some of the basic issues in IPC primitive design, the use of these proposals in distributed real-time control applications is examined. A distinction is made between mutual exclusion protocols and interprocess communication primitives, and it is shown how a theory of real-time dynamic systems and the computer control of such systems can assist with the design and analysis of IPC primitives and mutual exclusion protocols.

3.2 The need for interprocess communication

Interprocess communication is a facility which enables concurrently executing processes to exchange information with one another. Processes which have potential for parallel execution are described as concurrent processes [46,47]. Concurrent programming refers to programming techniques that are used for exploiting potential parallelism and for resolving the resulting communication and synchronization

problems. Interprocess communication is thus a facility which enables concurrent programming problems to be solved. Concurrent programming difficulties include:

- Ordering of events both internal and external to the system
- Synchronization of processes
- Exchange of data between processes

These difficulties manifest themselves as problems with mutual exclusion and the protection of critical sections. Most problems in concurrent programming can be reduced to the mutual exclusion problem [13]. The difficulties listed above are encountered in programming both centralized multiprocessing systems and distributed systems. Before we delve into some of the proposed solutions, let us consider the abovementioned problems in a little more detail.

3.2.1 Ordering of events

Ordering of events is a problem which is limited to distributed systems. For the purposes of ordering events, Lamport defines a distributed system to be one where the message transmission delay is not negligible compared to the time between events in a single processor [41]. Humans tend to order events with reference to some global physical time. In distributed systems there is no single clock specifying global physical time (because the clock would be a centralized critical resource). Each processing module has its own local clock. The finite accuracy of physical clocks and difficulty experienced in synchronizing clocks imply that on a theoretical level it is not possible to view several clocks in a distributed system as a single physical clock. In section 5.4 we explain why physical clocks (as opposed to logical clocks) must be used in real-time distributed computer control systems.

There are many reasons as to why it is important to establish the order in which events that affect a system occur. The possibility of one event causally affecting another depends on the order in which the events occur, and therefore without a mechanism for ordering events, a distributed system will not be able to make any decisions or perform any operations which are based on cause-effect relationships. To say that event[1] a happened before event b only makes sense if events a and b

[1]The term "event" refers to an instantaneous happening which has no duration. In [13] Lamport

occurred in systems which reference a common time base. Lamport therefore defines a method for ordering events in a distributed system which is not based on physical time. This method uses logical clocks to establish the order of events. Logical clocks cannot be used to order events external to the system. In [41], Lamport presents algorithms for synchronizing clocks in distributed systems which can be used to order events external to the system. Lamport defines the precedence relation "$\longrightarrow$" to mean "happened before". Therefore if event a represents the sending of a message, and event b represents the receipt of that message, then $a \longrightarrow b$. Events a and b are said to be concurrent if $a \not\longrightarrow b$ and $b \not\longrightarrow a$.

In [13] and [14] Lamport extends this theory and makes a distinction between precedence and causality. Since a single process (operation execution) is a set of events which has an *a priori* total ordering, it is possible for one process to causally affect another if it has an event which proceeds some event in that other process or if it shares an event with that process. The relationships can be stated as follows:

For two processes A and B, a being an event in A and b being an event in B:

$$A \longrightarrow B \stackrel{\text{def}}{=} \forall a \in A : \forall b \in B : a \longrightarrow b$$

Causality is defined using the "can causally affect" operator:

$$A \dashrightarrow B \stackrel{\text{def}}{=} \exists a \in A : \exists b \in B : a \longrightarrow b \text{ or } a = b$$

Lamport derives his treatment of event-ordering, precedence and causality using the basic principles of the geometry of space-time as presented in the theory of special relativity [13].

3.2.2 Synchronization of processes

Processes may be required to initiate or terminate actions based on the status of other processes executing in the system. Communication between processes which does not involve the exchange of any data is referred to as synchronization. The mechanisms used for synchronizing processes are the same as those used for inter-process communication, except that no data is exchanged [46,35]. (For example, in a message-passing system, synchronization messages consist of a header and a null

discusses the treatment of events which have a non-zero duration. (See definitions in sections 4.2 and 5.4.1.)

body. The structure of communication and synchronization messages is nonetheless the same.)

3.2.3 Communication between processes

Processes executing concurrently in a distributed system generally express their co-operation through synchronization and communication requirements. Interprocess communication can be effected in four logical access patterns, namely:

One-to-one Process A may wish to transmit a private message to process B, or process A may be producing data which is to be consumed by process B.

One-to-N Process A may wish to broadcast a message to all N processing modules in the network (or to a subset of these processing modules - this is often referred to as a *multicast* [32]). There may also exist multiple nodes which are all able to consume data produced by a single producer. Alternatively, all N processing modules may wish to make use of the data without consuming it.

N-to-one Process A may be collecting data from all processing modules in the network (or from some subset of these). Alternatively, process A could be waiting for data from any M-out-of-N processing modules.

N-to-N Any one- or M-out-of-N processes could be producing data which can be used or consumed by any one- or M-out-of-N other processing modules.

3.3 The mutual exclusion problem

The mutual exclusion problem is important in the theory of concurrent programming because it epitomizes several interprocess synchronization problems [46]. Consider two processes, A and B. Processes A and B each perform some operation (a and b respectively) which affects a resource R. This resource could be shared data or some hardware device such as a printer. The nature of operations a and b is such that when process A is performing operation a, it must have exclusive access to R, and when process B is performing operation b, it must have exclusive access to R. The portions of processes A and B which execute operations a and b are called *critical sections*. For example a and b may cause output to be printed on a

line printer, R. If a and b occur simultaneously, then the data output by process A will be interleaved with the output of process B, resulting in incorrect system behaviour. Because processes A and B are executing concurrently, it is likely that some time during execution the processes will be in their critical sections at the same time. In centralized uniprocessor systems, pseudo-concurrent execution of processes is achieved by allowing the processor to allocate a portion of time to the execution of a small section of each process. This means that any pre-emptive system could swap out process A whilst operation a is only half compete, load up process B and begin executing operation b. In distributed systems which are capable of fully concurrent execution of processes, the shared resource R is usually located at a remote processing module. If the processing module does not make some effort to ensure that access to R is controlled in some way, then the system may execute incorrectly.

The problem of ensuring that processes which have critical sections, or which share resources, do not execute in their critical sections simultaneously is referred to the problem of mutual exclusion. As described above, the mutual exclusion problem arises in both centralized and distributed systems. Centralized systems may have one or more CPUs, but the CPUs share a common memory store. In distributed systems however, each processing module has its own local memory, and there is no shared memory available for interprocess communication. Processes can only communicate by message-passing. Many of the proposed solutions to the mutual exclusion problem are based on shared memory communication techniques and are only applicable to centralized systems. The difficulties involved in finding a more general solution to the mutual exclusion problem which can also be applied to distributed systems have only been realized in recent years.

3.4 Interprocess communication primitives

Before reviewing any IPC primitive proposals, it is necessary to identify the issues which arise in IPC primitive design, and to examine different modes of communication. Certain primitives have been designed for specific applications. There is no one primitive which is ideally suited to all applications. The following criteria are used by Tannenbaum [31] to categorize IPC primitives:

3.4.1 Shared memory and message-passing

Processes executing concurrently (or pseudo-concurrently) on *uniprocessor systems* or on *multiprocessor* systems which have a common shared memory store, are able to synchronize their activities or exchange information by means of this shared memory. In multiprocessor systems, where each processing module has its own local memory and there is no shared memory, message-passing is the only technique which processes can use to communicate with one another.

3.4.2 The client-server model

This model is used to describe the basic interaction of two processes. Process A sends a message to some specified destination (or destinations) using the *send* primitive, and process B receives a message from a specified source using the *receive* primitive. The client-server model can be based on either a reliable or unreliable communication model.

Reliable The *send* primitive can ensure that the receiver obtains messages in the desired format. That is, the *send* primitive handles lost and corrupted messages, network failures, retransmission-on-error, and acknowledgement. When *send* is completed, the sending process can be sure that the receiver process has received the message correctly.

Unreliable The *send* primitive transmits the message to the receiver, but makes no additional effort to ensure that it is received correctly. Thus delivery of the message is not guaranteed, and there is no retransmission-on-error.

3.4.3 Blocking and non-blocking primitives

A non-blocking *send* primitive returns control to the program immediately after the message has been queued for transmission. The program is then free to continue executing. A non-blocking *receive* primitive provides a buffer into which the incoming message is placed. The *receive* primitive may notify the process by means of an interrupt, or the process may poll the buffer periodically. Non-blocking primitives are flexible and they offer more potential for parallelism. The asynchronous nature of non-blocking primitives makes programming and debugging difficult. Blocking

primitives reduce potential for parallelism, but their synchronous nature makes process interaction more deterministic. A blocking *send* only returns control to the program when either:

1 The message has been sent - *unreliable blocking send.*

2 The receiver has acknowledged receipt of the message - *reliable blocking send.* [31]

A blocking *receive* does not return control to the program until the message has been placed in the buffer. A reliable *receive* automatically acknowledges receipt of a message. It is possible to use a blocking *send* with a non-blocking *receive*, and vice versa.

3.4.4 Buffered and unbuffered primitives

Consider two communicating processes, A and B. A wishes to send a message to B, so it executes a *send*. Process B however, has not executed a *receive*. If the *send* is blocking, then the sending process will not be able to continue with its execution until B executes a *receive*. (This is the rendezvous - ADA, CSP.) A similar problem would arise if process B executed a *receive* before A executed a *send*. If the sender and receiver could buffer messages, then the blocking would not occur. These buffers can be implemented as part of the operating system kernel as *mailboxes*.

3.5 IPC primitive proposals

A number of proposed solutions to the mutual exclusion problem are now considered. Some of the solutions are interprocess communication mechanisms which can be used to solve the problems of interprocess communication and synchronization in general. In the literature they are presented as solutions to the mutual exclusion problem, because it is the classical abstraction of the interprocess communication problem. Other solutions propose higher-level interprocess communication primitives which make it possible for programmers to use mutual exclusion protocols to solve concurrent programming problems [48].

The following solutions and proposals are considered: (Each proposal is first discussed briefly, and in section 3.6.1 the suitability of the various proposals to

computer control applications is examined.)

- Co-routines
- Semaphores
- Monitors
- Communicating Sequential Processes
- Distributed Processes
- Mascot
- ADA Rendezvous
- PLITS
- Liskov's Proposals
- Kramer's Proposal
- Accent
- Lamport's solution

Many primitives for interprocess communication have been proposed over the last few decades, and the selection of proposals discussed here may seem somewhat arbitrary. However, we have chosen to discuss proposals which either incorporated some major new innovation (at their time of publication), or which are particularly appropriate to distributed computer control systems. The proposals are presented in order of publication.

3.5.1 Co-routines

Processes executing in a system where control is explicitly passed from one process to another are called *co-routines*. A simple memory arbiter allows the processes to ascertain whether they are permitted to enter their critical sections. The memory arbiter is used in conjunction with a shared memory segment which specifies the identity of the process which is entitled to execute its critical section. When a process exits its critical section, it writes the identity of the next process which is permitted

to enter its critical section into the shared memory. The use of co-routines ensures that only one process executes in its critical section at any one time, and deadlock is avoided because there will always be one process which is permitted to execute its critical section. Explicit passing of control is problematic, because processes are tightly coupled. Consider a system where process A executes once every hour and where process B can execute every minute. If process A passes control to process B, then process B will also execute only once every hour. Thus all the processes are constrained to operate at the pace of the slowest process in the system. Another problem results from this close-coupling effect. Consider two processes A and B. If process A passes control to process B and process A crashes, then process B will get stuck waiting to enter its critic[illegible] ese reasons, co-routines are not regarded as an effective solution to [illegible]ng problems [46].

3.5.2 Semaphores

Semaphores were developed by Dijkstra [10,49]. A semaphore is a low level interprocess communication and synchronization primitive which can be used to implement higher-level primitives. A semaphore is essentially an integer variable which can only take on non-negative values. When the semaphore has been assigned an initial value, the primitive operations *wait*() and *signal*() can be performed on it.

Consider a semaphore s:

$P(s) = wait(s)$: If $s > 0$ then $s := s - 1$; else suspend calling process.

$V(s) = signal(s)$: If there is some process that has been suspended by a $wait(s)$ then wake up that process; else $s := s + 1$.

Dijkstra and others have demonstrated how semaphores can be used to solve many problems in concurrent programming, including the producer-consumer problem, the sleeping barber problem, the bounded buffer problem, and the dining philosophers problem [46]. Semaphore "integer values" are stored in regions of shared memory, and for this reason semaphores are only suitable for interprocess communication and synchronization in centralized systems. Much has been written regarding the use of semaphores in interprocess communication. For more background the reader is referred to Ben-Ari [46], Brinch-Hansen [9] and Peterson and Silberschatz [35]. The original proposal is described in [10].

3.5.3 Monitors

A monitor is created by an initialization process which is executed before the concurrent processes begin. The initialization process assigns a value to *monitor variables*. It then terminates, leaving a set of variables and procedures to manipulate these variables. (Monitors are very similar to classes in object-oriented programming. They incorporate data as well as the procedures to manipulate this data.) The only way that the monitor variables can be accessed is by calling the monitor procedures. The monitor procedures can only be executed by one process at a time. This ensures that the process executing the monitor procedure will have exclusive access to the monitor variables, and hence mutual exclusivity is guaranteed. One of the major advantages that monitors have over semaphores is that by allowing the use of data types and procedures, they allow the programmer to use structured programming techniques.

Monitors are high-level interprocess communication primitives. Any required *waiting* and *signalling* on conditions is programmed into the monitor procedures. Once the monitor is correct it will always function correctly. The programmer will not erroneously omit any signal to release mutual exclusion, as he could quite easily do using semaphores. Although monitors are higher-level primitives than semaphores, they can nevertheless be used to implement semaphores. Despite their flexibility, monitors can only be used in centralized systems which support shared memory. Monitors are the standard IPC primitives in Concurrent Pascal (Brinch-Hansen [50]). A basic example illustrating the use of monitors can be found in Ben-Ari [46]. For a more detailed theoretical treatment of monitors, the reader is referred to Brinch-Hansen's and Hoare's original papers [50,12].

3.5.4 Communicating Sequential Processes

In his paper "Communicating Sequential Processes" (CSP), Hoare [51] asserts that input and output are fundamental programming primitives for parallel systems. Using the principle of Dijkstra's *guarded* command, he develops a mechanism which facilitates communication between concurrent processes. In CSP, guards and data typing are used to effect conditional input of messages. Communication only occurs when process A names process B as a destination for output *and* when process B

names process A as a source for information input. The absence of buffering means that the input and output commands are blocking. A process A which desires to output data to some other process will be suspended until the destination process requests to input data from process A. Similarly, any process which desires to input data from a process A will be blocked until process A explicitly outputs the data to the blocked process. This synchronous communication primitive is known as the *rendezvous* (this mechanism was later incorporated into the ADA language). Although the blocking of processes and explicit naming conventions enforced by CSP make the model highly restrictive, support for message-passing makes the protocol suitable for use in distributed systems [51].

3.5.5 The ADA rendezvous

ADA is a language which was designed specifically for programming real-time concurrent systems. ADA's interprocess communication mechanism is based on the rendezvous concept developed by Hoare in CSP. Whichever process "arrives" at the rendezvous first is required to wait for the other process to arrive. A single transaction is completed when the rendezvous is made. Because the rendezvous can only be made by one process, mutual exclusion of the transaction is assured. There is a two-way exchange of information at the rendezvous. The *accept* statement performs the rendezvous, and information is passed from the calling process and back from the called process. The *accept* statement does not allow a process to rendezvous with some specific process; the rendezvous will be made with any process which is entitled to make the rendezvous and tries. ADA implements guards by means of the *select* statement and *when* clause. The *select* statement facilitates a rendezvous with some specific process out of several alternatives. If, in the *select* statement, all the guards (*when* clauses) are open, then a rendezvous is performed with the first process that calls. The ADA rendezvous restricts parallelism, even though it can be used in concurrent systems [46,52,4].

3.5.6 Distributed Processes

Distributed Processes (DP) was a concurrent programming concept developed by Brinch-Hansen (1978) [44]. It was aimed specifically at solving real-time distributed

computing problems. Brinch-Hansen specifies a typical real-time system as one which has a fixed configuration, cyclical processes, and preferably a single task per processing module. Essentially, Distributed Processes extends the monitor concept to the case where processes are distributed. Monitors in centralized systems contain procedures which can be called by processes running in the same processing module. These procedures modify the data contained in the shared memory. Distributed Processes is a system which allows procedures to be defined within processes which can be called by remote processes. Parameters and data are communicated by message-passing. Mutual exclusion is assured during the remote procedure call (RPC) because the calling process is blocked until the RPC is completed. Guarded commands and guarded regions are used to control non-determinism. Guards enable a process to select a statement based on the state of its local variables (conditions). Guarded regions are blocking, whereas guarded commands are skipped or generate exceptions when all the guards are closed. DP does not provide a mechanism which enables the called process to select a requesting process [14].

3.5.7 Liskov's proposal

Liskov proposes a model for reliable, fault-tolerant interprocess communication. The proposal specifies a model consisting of *guardians*. Each guardian consists of one or more processes, and is located entirely at a single processing module in the distributed system. Inter-guardian activity is more expensive than intra-guardian activity because calls are made to remote processing modules. Each guardian has *ports* through which messages are passed. The ports are typed, and so message communication can be regulated by type checking. Liskov's proposal resembles DP in its use of remote procedure call, except the call is non-blocking. The major innovations in Liskov's proposal concern techniques for dealing with the failure of the network, transmission medium, and processing modules. Liskov advocates the use of *atomic objects* which are acted on by *action calls* which can be either *top-level calls* or *nested calls*. The use of action calls and atomic objects means that backup versions of all atomic objects are kept and are only overwritten when calls are committed. If a call is aborted, then the backup version of the object is re-instated. Liskov proposes these primitives as extensions to the CLU language [5].

3.5.8 Accent

In shared memory systems, data can be communicated by reference. Most of the proposals for distributed systems that have been examined so far have not considered the passing of pointers in messages (i.e. communication of data by reference). It is widely held that the support of shared memory over a network is impractical, because of communication overhead, reliability problems and the consequences of failures. Accent provides a novel solution to the problem. The basic Accent IPC primitive is an asynchronous message-passing protocol, where messages contain data which is being communicated by value. These messages may, in addition, contain pointers to pages of virtual memory. The primitive maps the specified pages of virtual memory to the receiver process's physical address space, thus allowing data to be communicated by reference. This greatly reduces the likelihood of failure, and at the same time, the copying of messages to and from buffers is significantly reduced. Accent was developed for applications where the cost of message passing is much less important than the cost of virtual memory applications. Messages in Accent vary in size, but it is quite common for messages to be very large. It is this which makes the option of by-reference communication an attractive one. Accent overcomes these problems by integrating the functions of virtual memory management and interprocess communication. The developers of Accent note that virtual memory management and message-passing are traditionally regarded as distinct problems. This is because message-passing is frequently used in real-time applications such as process control, whereas virtual memory management is hardly applicable [53].

3.5.9 Kramer's proposal

Kramer and co-workers (1984) [2] propose a software architecture and interprocess communication mechanism which caters to the specific requirements of real-time distributed computer control systems. The system is modelled as a number of *modules*, each consisting of several *tasks*. Each processing module runs one or more *modules*, however execution of a single module may not be performed by more than one processing module. Modules communicate with one another by message-passing. Messages enter and leave modules through *ports* which are typed, giving added security without runtime overhead. Exit ports are not required to specify entry ports

(and vice versa) in their definitions. Association of entry and exit ports is performed during system installation (or modification) by *linking* them to one another. The method of linking entry and exit ports allows modules to be reused as and when required. Kramer's proposal features both blocking (synchronous - *send* - *wait* - *receive* - *reply*) and non-blocking (asynchronous - *send* - *receive*) message-passing primitives. Both the blocking and non-blocking primitives have been designed to cope with failures. The *send* - *wait* - *receive* - *reply* (blocking) primitive implements a one-to-one communication pattern, whereas the *send* - *receive* (non-blocking) primitive provides a one-to-N communication pattern. Each *receive entryport*, when used with the *send* - *receive* primitive is required to buffer the incoming messages. The message queue buffer size is statically allocated, and the default size is one message. This means that old messages which are not consumed, are overwritten.

3.5.10 Plits

The Plits project (1979) [54] was aimed at incorporating message-passing primitives for programming distributed systems into a high level language. The Plits model of a distributed system consists of a number of modules which may communicate with one another only by message-passing. Modules have complete control over their own internal states, and it is therefore possible to code modules so that they do not reach an undesirable state. The Plits primitives and high-level language commands were designed based on the assumption that communication links in distributed systems are unreliable. The message-passing protocol is asynchronous, and messages are sent by value. Order of message delivery is preserved, and message queues are bounded only by system resources. Many of the Plits language concepts have matured and found their way into current object-oriented languages. The Plits developers claim that very strong typing does not necessarily promote careful programming. They maintain that the encapsulation of local data and code in modules, together with a well-structured message-passing protocol, is more effective in enforcing careful programming. Plits's *receive* primitive can select specific messages by means of *transaction keys*. It is assumed that each Plits module will always reside at the same processing module.

3.5.11 Mascot

Mascot (Modular approach to software construction, operation and test) is a programming methodology which was developed for centralized systems. It combines the advantages of semaphores and monitors into a powerful IPC mechanism. Both mutual exclusion (semaphores) and cross-stimulation (monitors) are ensured through the use of primitives operating on a single type of control variable called a control queue. A Mascot subsystem consists of subsystems which are connected by "intercommunication data areas". Two basic types of intercommunication data areas are provided - *channels* and *pools*. Channels pass message data from one activity to another, whilst non-transient data are deposited in pools. The *join - leave* primitive pair simulates the behaviour of a binary semaphore, whilst the *wait - stim* primitive pair simulate the wait and signal on a condition variable. When used together (*join - leave - wait - stim*) semaphore and monitor functionality can be obtained by using only half the number of control variables. Mascot IPC primitives, although well suited to message-passing (in centralized systems), still rely on shared memory [55].

3.5.12 Lamport's solution

In 1986 Lamport published a solution to the mutual exclusion problem, which is fundamentally different to other solutions [13]. (Some of the concepts underlying his solution were initially published in an earlier paper [15].) His solution (hereafter referred to as *Lamport's solution*) is equally applicable to both distributed and centralized systems. Lamport begins by making two observations about other solutions to the mutual exclusion problem:

- All solutions are based on indivisible atomic operations.
- Solutions which are based on transient communication techniques where only the reader-process is allowed to set the communication bit, all only solve one specific case of the mutual exclusion problem — the *producer-consumer* problem.

Before Lamport's solution is described we will examine these two problems in more detail.

Atomic operations

An atomic operation is one which is either performed in its entirety or not at all. That is, an atomic operation is an indivisible action. The mutual exclusion problem addresses the issue of two operations affecting one another, such as in a situation where interprocess communication takes place. Two atomic operations performed concurrently will have the same effect as performing the actions in some order. Lamport notes that assuming the availability of atomic operations for the solution of mutual exclusion problems is simply assuming that a low-level solution to the mutual-exclusion problem does exist. Atomicity of operations may even be assured at the hardware level, however Lamport remarks that no solution which is based on atomic actions can be considered a fundamental solution to the mutual exclusion problem.

Transient and non-transient communication

Two processes may communicate a single bit of information either by setting a bit or by sending a message. A message is a transient phenomenon, and the receiver process must be waiting to receive it if communication is to be successful. (The setting of a bit is non-transient, and the reader process is not required to wait for the write operation.) Lamport shows that the mutual exclusion problem cannot be solved using transient communication techniques when the following conditions hold:

- A process need not communicate whilst executing in its critical or non-critical sections. It only needs to communicate when entering or leaving its critical section.
- It is possible that a process may remain in its non-critical sectio [13] indefinitely.

These conditions clearly show that the producer-consumer problem can be solved using transient communication techniques. In [13], Lamport discusses in detail the situation where these conditions exist. He also explains why a non-transient communication mechanism where only the reader process can set the communication bit does not solve the mutual exclusion problem for the general case.

Lamport's mutual exclusion protocol

Lamport's solution does not assume that atomic actions are provided by the underlying subsystem. (For example - that two operations performed on a single memory cell occur in some definite order.) (In [56] Lamport improves his approach to proving the correctness of multiprocess programs developed in [57]. In this paper he dispenses with the need to decompose algorithms into atomic operations.) The proposed protocol uses non-transient communication. A communication bit, which can only be reset by the writer process, is used for synchronization. All writes to the communication variable (bit) must be totally ordered. If V^i denotes the ith *write* then

$$V^1 \longrightarrow V^2 \longrightarrow \cdots$$

Each communication variable is "owned" by a specific process, and so this total ordering of writes is not difficult to ensure. Lamport notes that one of the underlying assumptions in this theory is that each process knows exactly which other processes it will have to communicate with. In [58] and [59], Lamport presents four solutions to the mutual exclusion problem in a distributed system of N processes. Fairness and failure tolerance are considered, and each of the four solutions satisfies stronger conditions than the previous one. (Communication overhead also increases progressively.) For details of these solutions, the reader is referred to [58]. Lamport's solutions are all based on his mutual exclusion protocol. Each process owns a communication variable which can only be written to by the owner process. This communication variable serves as a synchronization flag. A process may enter its critical section by setting its flag to *true*, and by making sure that all other flags (i.e. those owned by other processes) are false. Under these circumstances mutual exclusion is assured. (In [60], Anger determines the extent to which Lamport's solution differs from solutions which assume atomic actions.)

3.6 Interprocess communication in computer control

Mutual exclusion in real-time DCCSs is provided by an appropriate interprocess communication primitive. We highlight IPC requirements for real-time DCCSs by examining the applicability of the models discussed above.

3.6.1 An evaluation of existing proposals

Co-routines, semaphores and monitors all rely on shared memory, and therefore they are not suitable for use in a distributed system. Although the MASCOT IPC mechanism is based on a message-passing protocol, it also relies on shared memory. Both CSP and the ADA rendezvous restrict parallelism because the sending process and receiving process are required to sychronize with one another. Thus the process which arrives at its synchronization statement first is blocked. Brinch-Hansen's "Distributed Processes" also restricts parallelism because the calling process is blocked until the remote procedure call has been completed. Despite this disadvantage, Brinch-Hansen's primitives were designed for real-time applications, and they exhibit many features which are desirable in an IPC for distributed real-time applications. The Accent IPC integrates virtual memory management with interprocess communication. The type and size of messages transmitted in real-time DCCSs make the provision of shared memory over the network unnecessary. Plits, Liskov's proposal and Kramer's proposal, all address the programming of distributed systems from a high-level language perspective. All three models assume that communication links are unreliable, and they all address the problems of synchronization and communication in distributed systems. The Plits project does not address the problems of real-time computing (see chapter 5), and hence it is not regarded as being suitable for control applications. Liskov's and Kramer's proposals are highly relevant to real-time DCCSs, only they deal with high-level language primitives, and assume solutions to the basic mutual exclusion problem. Lamport's mutual exclusion protocols solve the mutual exclusion problem at a theoretical and algorithmic level. They do not address the implementation of high-level language primitives for interprocess communication.

3.6.2 A theory of interprocess communication

In the following chapter we begin the derivation of a model of process interaction in real-time distributed computer control systems. This model can be used to explore and express the requirements of interprocess communication primitives which are to be used in real-time distributed control systems, for example, whether they should be blocking or non-blocking, reliable or unreliable, whether or not they should buffer

messages, etc.

The interprocess communication proposals discussed above can be split into two broad categories:

1. Protocols and algorithms which are aimed at guaranteeing logically correct mutual exclusion.

2. High-level language primitives which assume the availability of one or more underlying mutual exclusion protocols. These primitives implement lower-level mutual exclusion protocols in various guises, thereby allowing the programmer to concentrate on applications programming, and greatly reducing the amount of complex systems programming that he has to perform.

In our research we aim to isolate and examine the issues which impinge on the design of protocols and primitives which fall into categories 1 and 2. This will allow us to develop a general theory of real-time dynamic systems, and the computer control of such systems. A model of process interaction based on this theory can then be used to analyze and design protocols and primitives which fall into both categories 1 and 2. Such a model is necessary because we believe that:

1. An effective high-level primitive will not simply assume that some underlying mutual exclusion protocol exists. The high-level primitive should be "designed around" an effective mutual exclusion protocol if it is to be effective.

2. A careful analysis of the fundamental issues in real-time control systems leads to a clear specification of how the mutual-exclusion protocol should behave.

3.7 Conclusion

The subject of interprocess communication and synchronization was examined in detail. A number of theories of interprocess communication were compared with one another, highlighting the strengths and weaknesses of the various proposals in different applications.

Some recommendations regarding the design of IPC primitives for use in real-time DCCSs were made in the second half of the chapter. These include:

- High level language primitives must be designed to operate "in concert" with low-level mutual exclusion protocols.
- A general theory of real-time dynamic systems and the computer control of such systems is required if the problems of interprocess communication in real-time distributed systems are to be solved effectively.

In the next chapter we develop a new model for data representation and data sampling. We use this model together with the recommendations made in this chapter to develop a more comprehensive model of process interaction in real-time distributed computer control systems.

Chapter 4

Sampling and Data Representation

4.1 Introduction

In process control applications, computers gather data by means of sensors and data acquisition equipment. We use the term *data* to refer to qualitative or quantitative information which is meaningful to the system. In computer controlled systems this information is usually generated at discrete points in time and communicated between the plant and the control system. The data may be logged for future reference, presented to a human operator or to some other subsystem via a suitable interface, or it may be used in the execution of some control strategy. The essential elements underlying all such applications are measurement, sampling and data representation. Measurement is the quantification of certain properties of the physical system. Sampling is the operation whereby measurements are transferred to the computer system, and data representation is the mechanism which causes sampled measurements to be stored and made accessible in the computer. Little exists in the way of theory which describes the synergy of these operations. Practical implementations are based largely on experience and heuristics which cannot necessarily be applied to all systems. With the use of distributed computer systems in control applications increasing, the need for a *theory* describing the synergy of measurement, sampling and data representation is even more apparent. It is the aim of this chapter to lay the foundations for such a theory.

When a computer control system is modelled, the plant and the internal operations in the computer are treated as processes or sets of processes. Measurement, sampling and data representation thus together constitute an interprocess communication mechanism which facilitates communication of data between the physical plant and its control system.

In this chapter we examine measurement, sampling and data representation from first principles. We begin by describing the measurement process and its effect on the plant. A distinction is made between continuous and discrete time measurement systems. Thereafter we examine different sampling methods, and develop data representation techniques from our intuitive model of the plant and its control system. The application of these sampling methods and data representation techniques in computer control applications is then analyzed. The concepts developed in this chapter form the basis of our *p_var* model. Our intuitive model forces us to reconsider our understanding of sampling and plant-initiated communication.

4.2 Representation of plant properties

Consider a physical plant with unknown dynamic characteristics. We say that such a plant has a set of well defined *properties*, which are intrinsic to the physical reality of the plant, and which completely describe the state of the plant (fig. 4.2). These properties are continuously variable. They may change continuously with time or they may change as a result of an *event*. An *event* is defined as an occurrence at a point in time in the plant or its environment which causes a significant change in one or more properties.

Any attempt to measure properties or to quantify them in a discrete manner, results in *measured values*, which represent the properties with finite accuracy after a finite measurement delay. *True values* would be obtained if a perfect measurement system were used to measure the properties (Finkelstein, [29]). Because no perfect measurement system exists, measured values are not capable of exactly representing properties.

Consider a human observing a physical plant. The human perceives changes in the plant's properties as rapidly as his visual system and mental processing capabilities permit. The human is the measurement system. He may not be able to perceive

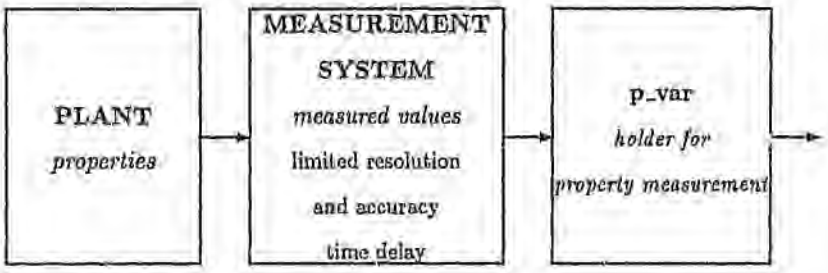


Figure 4.1: Quantification of plant properties

very small changes in the properties, and those that he does perceive, he may notice only some time after they have occurred. These limitations are due to the deficiencies of his own faculties; they do not result from any deficiency in the plant, and are thus independent of the properties. Physical sensors and measurement systems behave in much the same way.

Let us now consider the process of sampling as it is performed in continuous and discrete systems:

4.2.1 Continuous-time measurement

A continuous-time measurement system derives a quantitative numerical representation of a property from a transducer designed to be sensitive to that property. Continuous-time measurement systems have finite resolution due to noise, hysteresis and constructional limitations. For example, a human observer, generally assumed to be a continuous-time measurement system, is also subject to these limitations. His visual system cannot discern changes smaller than some minimum. Accuracy is limited by non-linearities in measurement equipment, and the measurements themselves are corrupted by noise. In continuous-time measurement systems, the time that it takes for a change in the physical property to be reflected in the measurement is sometimes almost negligible, since electrical signals propagate extremely quickly. Often, however, the measurement system consists of electro-mechanical components (such as springs and magnets) which have a limited dynamic response capability. Non-linear components with significant high order dynamics are common, and the settling times of these components can introduce noticeable delays.

4.2.2 Discrete-time measurement

Discrete measurement systems measure the plant properties at discrete points in time. The measurement operation is called sampling. Where measurements are performed at points in time equidistant from one another, sampling is said to be periodic. (Note that repetitive sampling does not have to be periodic. However, processing of data which has been acquired using periodic sampling is simpler and faster [28,61].) Before a discrete measurement system can be designed, the dynamic characteristics of the physical plant must be examined and quantified in order to determine the required sampling frequency. The relationship that exists between the plant dynamics and the required sampling frequency is described in Åström and Wittenmark [28] (see chapter 5). Measurement noise, hysteresis, non-linearities, and constructional limitations of sensors all affect discrete-time measurement systems in much the same way as they affect continuous-time measurement systems. Discrete-time measurement systems lose their ability to represent properties if they do not sample the properties frequently enough.

4.2.3 The effect of discrete-time measurement

Aside from the measurement delay which occurs in both continuous-time and discrete-time measurement systems, discrete-time systems may introduce an additional delay in reflecting a property change in a measured value. If the property change occurs immediately after sampling has been performed, that property change will only be reflected in the measured value which results from a subsequent sampling operation. This delay implies that a change in the property could have occurred some time in the past, where this time is at most equal to the sampling period, T_s. The use of slightly out-of-date measured values is permitted because the choice of sampling period is based on the dynamic characteristics of the property. It is therefore assumed that the magnitude of the property change during one sampling period can be neglected. If sampling is fast relative to the plant dynamics, then continuous-time dynamic system theory can be used. However, if the plant response time and the sampling rate are of a similar order then digital control techniques must be used (Åström and Wittenmark [28]).

4.2.4 Sampling of properties

It is clear that the measurements of properties and the properties themselves are distinct. A physical process consists of several subsystems. The state of a particular subsystem at a point in time is quantified by assigning numbers or symbols to a suitable set of well-defined physical *properties* such as temperature, position or voltage. It is necessary to represent these numerical or symbolic values internally in the distributed computer system in order to provide an internal "image" of the state of the subsystem. This image is then used as the basis for all monitoring and control functions.

We use the general term *property variable* or *p-var* to mean a holder for the value of a property. The concept of a property can be broadened to include entities for which the human operator supplies the value and internal entities where the value is calculated by the computer system.

The operation of sampling is used to quantify a property and place the value in a *p_var*. Different policies can be used to initiate sampling (fig. 4.2):

Event-triggered sampling The sampling operation is triggered as the result of a change in a property.

Periodic sampling Sampling is performed at pre-determined equidistant points in time, regardless of the value of the property.

These two definitions allow us to define two further concepts:

Non-periodic sampling Sampling is performed repetitively at points in time which are *not* equidistant. This concept is illustrated by the following example:
Consider a system where three samples are taken at times t_1, t_2 and t_3. The sampling period between the first two samples is described by $T_{21} = t_2 - t_1$. Similarly the period between the next two samples is given by $T_{32} = t_3 - t_2$. In a non-periodic system $T_{32} \neq T_{21}$.
One important difference between non-periodic sampling and event-triggered sampling is that a non-periodic sampling system can assume that another sample will be taken. The time that may elapse before that sample is taken is uncertain. Event-triggered systems may not assume that another sample will be forthcoming, since sampling depends on the occurrence of *events*.

Continuous sampling We believe that the relationship between discrete and continuous time control theory should be such that a discrete system with the sampling rate tending to infinity should tend to the continuous-time system [62]. For this reason we define continuous sampling as periodic sampling where the sampling period $T_s \rightarrow 0$. This definition can also be stated in terms of non-periodic sampling. Although $T_{32} \neq T_{21}$, $T_{32} \rightarrow 0$ and $T_{21} \rightarrow 0$. In the limit, $T_{32} = T_{21}$.

Plant properties usually change continuously with time. Such changes can only be detected instantaneously if the properties are observed by means of *continuous sampling*. The continuous sampling requirement can be relaxed where two conditions hold:

1. A small change in the value of a property is insignificant and need not be observed by the sampling process. We refer to the smallest change that must be observed as the *value granularity*, γ.

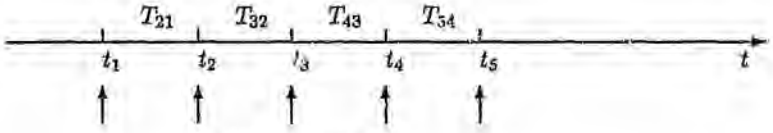
2. The property has dynamic characteristics such that a change of magnitude γ will not occur in less time than δ, where $\delta > 0$. We refer to δ as the *temporal granularity* of the system.

This pair of conditions can be restated as follows:
Continuous sampling (where $T_s \rightarrow 0$) is only required where the rate of change of the property is infinite ($dp/dt \rightarrow \infty$). This is the only condition under which a value can change at a rate γ/δ where $\delta \rightarrow 0$. The impulse function (or Dirac function) and the step function are examples of functions which exhibit this behaviour.

A property must be measured to some finite resolution. This desired resolution affects the value granularity, γ. The rate of change of the property and γ affect the temporal granularity, δ. The sampling period, T_s, is a function of δ. In the case of an ideal impulse, γ is irrelevant because the value changes instantaneously. Hence $\delta \rightarrow 0$ for all γ. Whenever $dp/dt < \infty$ then γ becomes important. If γ is large, then even if dp/dt is quite large, a larger value of δ may suffice.

From this analysis it is clear that continuous sampling is never required in physical systems. (For all physical properties, $dp/dt < \infty$.) What does emerge from this analysis is that plants exist with properties that meet either (or both) of the

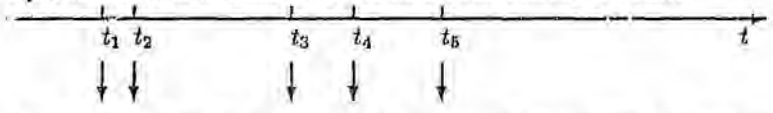
a) PERIODIC SAMPLING



Receiver samples

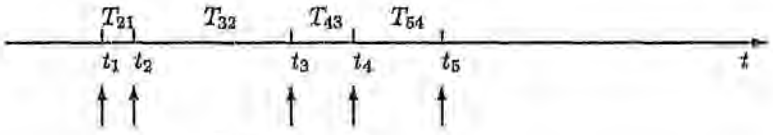
b) EVENT-TRIGGERED SAMPLING

T_{ij} is not defined because other events are not certain to occur.



Events occur at t_1, t_2, t_3 and t_4, and are communicated to the receiver

c) NON-PERIODIC SAMPLING



Receiver samples

d) CONTINUOUS SAMPLING

$T_{43} = T_{32} = T_{21} \to 0$ (therefore also holds when $T_{43} \neq T_{32} \neq T_{21}$)

Figure 4.2: Sampling mechanisms

following conditions:

1. γ is very small. Even if dp/dt is low, δ may still be close to zero. This condition may exist where a positional or pressure sensor may be required to measure very small changes in the property.

2. dp/dt is very large. Even if γ is also large, δ may still have to be close to zero. (Switches and circuit breakers can change from one state to another almost instantaneously.)

4.2.5 Periodic sampling

Periodic sampling (fig. 4.3) can be used to observe properties as long as the sampling period, T_s does not violate the constraints imposed by δ. The relationship between T_s and δ follows from the Shannon sampling theorem, although in real-time systems, which cannot tolerate the delay caused in Shannon reconstruction, a higher sampling frequency is used [28]. Suffice it to say here that for any property, p, where $\delta_p > 0$, there exists a sampling period T_s which can be used to sample p periodically.

$$\forall \quad \mathrm{p} \quad \text{s.t.} \quad \delta_{\mathrm{p}} > 0 \quad \exists \quad T_s > 0 \tag{4.1}$$

Each sampling operation incurs computational overhead. Even the simplest sampling algorithm is required to communicate with the data acquisition hardware using some protocol, and bits must be transferred into local memory. The lower the value of T_s, the more the computational overhead. If it were possible to design and build computers with infinite processing power and communication systems with infinite bandwidth, then it would be possible to design sampling systems which could sample properties at intervals of T_s, where T_s is very small. Modern computers possess tremendous computing power, and communication systems are becoming more efficient, and still it may seem that real applications are limited to sampling at much slower rates. The reason for this is that it is extremely expensive in monetary terms to dedicate a supercomputer to the process of sampling a single property. This is all the more true where the property has both high and low order dynamic components. A typical example of such a system might be a property which is changed by an operator t' owing a switch. The switch may only be operated once or twice a day, but the dynamics of the switching process are extremely fast, and may even

approximate a step (or an impulse). Periodic sampling of such a property requires that T_s be chosen such that the dynamics of the switching process can be detected. (In practice, it is normally acceptable for the property change to be detected some time after it occurs.) This means that the computer will have to perform a great deal of unnecessary computation in order to detect an infrequent event.

4.2.6 Event-triggered sampling

Event-triggered sampling (fig. 4.3) solves the problem of this inefficient use of computing power. It is a mechanism whereby the *change in the property* causes the property to be quantified and the value placed in the *p_var*. We maintain that this mechanism is an artificial way of getting around fast sampling requirements, because it *apparently* requires the plant to initiate communication with the data acquisition system. The plant, however, is unaware of the existence of this data acquisition system, so it cannot initiate communication of its own accord. An analogy to this method of sampling is the case of a photographer who wishes to take a snapshot of some unintelligent animate object. To say that the plant can initiate communication with the data acquisition system is akin to saying that the unintelligent animate object can tell the photographer when to take the snapshot. Clearly, any unintelligent animate object which gains the ability to communicate with a photographer is fundamentally different to the original unintelligent animate object. The same is true for a physical plant which is given the ability to communicate with the control system. In practice, it is obvious that the physical plant cannot initiate communication with the control system. (See section 4.3.4.)

4.2.7 Front-end periodic sampling system

The control system is therefore usually front-ended by some *continuous* or *periodic* sampling system, which may trigger on a threshold or by some other mechanism. (In section 4.3.3 we argue that this front-end data acquisition system is part of the control system and *not* part of the plant—see fig. 4.4) For example, a pressure sensor may monitor pressure constantly and trigger its output when the pressure reaches a preset level. It is this front-end system, not the plant, which initiates communication with the control system. Periodic sampling (or, in the limit, continuous sampling,

as described above) therefore underlies all sampling. It is for this reason that we say that event-triggered sampling is artificial, and that all sampling *is* inherently periodic.

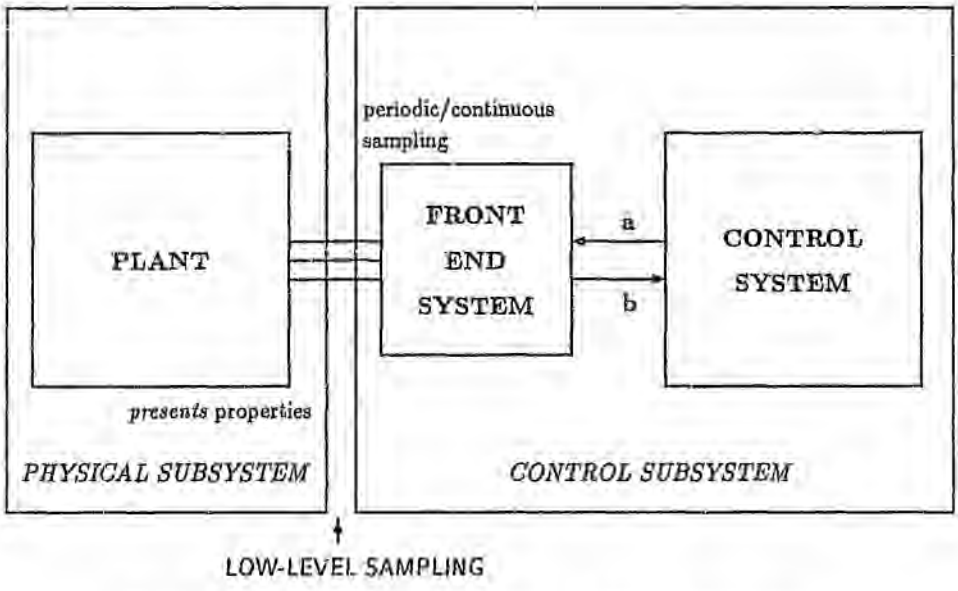


Figure 4.3: Periodic (**a**) and event-triggered (**b**) sampling policies

4.3 Data representation

The value which is stored in the *p_var* may represent either a state value or a state-increment value.

State value If the value stored in the *p_var* represents the value of the property relative to some absolute definitive reference, then we say that the *p_var* contains a state value.

State-increment value If the value stored in the *p_var* represents the net change in the property since the last communication of a state-increment value, then we say that the *p_var* contains a state-increment value.

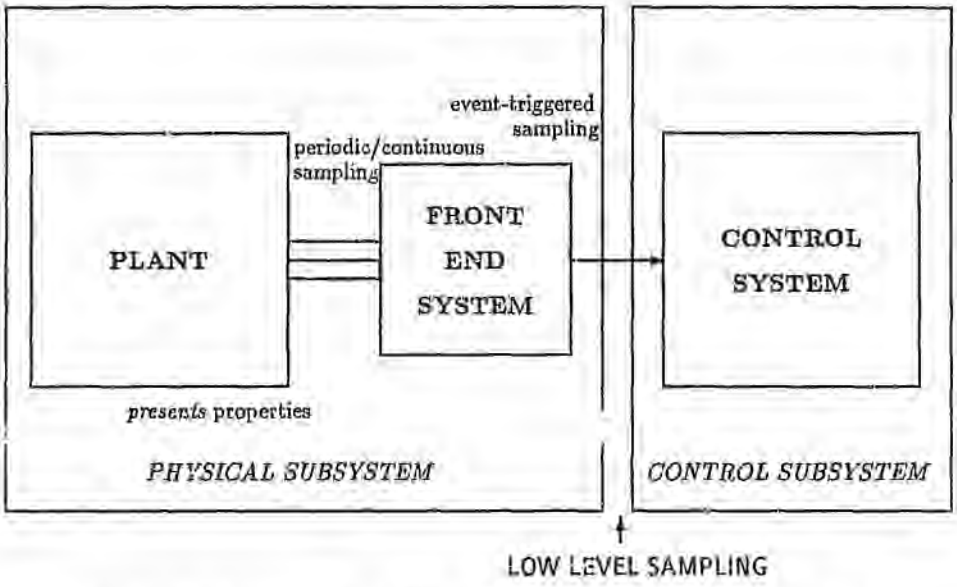


Figure 4.4: An incorrect view of the mechanism of event-triggered sampling

State values and state-increment values may be required to be persistent or they may have to be consumed. We define persistence and consumption of data as follows:

Consumed Data values which are to be used once and once only must be consumed. The read operation which is used to extract the value from the holder (the *p_var*) must empty the holder of its contents.

Persistent Data values which are to be used repeatedly (or more than once) must persist until they are invalidated by some other mechanism (for example, time-stamps). The read operation which is used to extract the value from the holder must ensure that the value in the holder is preserved.

4.3.1 State-increment values

State increment values [1] are a useful way of communicating information where the number of bytes needed to convey the change in the property, Δp, is significantly less

[1] State-increment values should not be confused with events. Events result in *property changes*, and these property changes can be communicated either as state values or state increment values.

than the number of bytes needed to communicate the value of the property. For this reason, state-increment values are often used in transaction processing systems (e.g. banking). When deciding whether state values or state-increment values are better, it is important to ascertain what sort of values the low-level sensors acquire. If the sensor acquires a state-increment value, but would like to communicate a state value, then some local computation must be performed by the sensor. For example, an automatic teller machine (ATM) can be regarded as the bank's "sensor". It acquires state-increment values (withdrawals and deposits) from the plant (customers). For the ATM to transmit a state value, it would need to have the customer's entire itemized transaction record (current state) stored locally. It would then update this transaction record with the new transaction (state-increment) and communicate this updated transaction record (new state) to the computer. The difficulties involved in implementing such a system clearly make the option of communicating state increment values an attractive one.

State increment values have several major drawbacks:

1. Reliable communication between sender and receiver must be ensured. If the state-increment value is lost or corrupted in transmission, then the state at the receiver will always be an incorrect reflection of the property. For this reason a reliable, failure-tolerant network communication protocol is required for state-increment communication. PAR (positive-acknowledgement and retransmission-on-error) protocols are commonly used to ensure that messages are correctly received. Some high-reliability primitives (Liskov [5]) use a *commit-rollback* protocol to ensure failure atomicity. This mechanism allows the sender to ascertain whether the message has been delivered or not. The rollback mechanism allows the sender to retract its transmission in the case of failure. When dealing with state-increment values it makes no sense to re-transmit the message "just to be sure", as receipt of a duplicate state-increment value will again cause the state at the receiver to be an incorrect reflection of the property.

2. Every state-increment value must be consumed. This is consistent with the failure atomicity constraint that is, that each state-increment value must be used once and once only. If the same state increment value is used more than

once, then the state at the receiver will no longer reflect the property.

3. State-increment values must be queued because each one must be used once and once only. Buffering incurs extra overhead, and has ramifications regarding flow control [26]. (See chapter 6.)

4.3.2 State values

State values are naturally derived from the observation of the properties of a physical plant. State values have major advantages over state-increment values:

1. State values overwrite one another. In the event of a transient communication failure which causes the loss of a state-value message during transmission, the receiver will not possess an up-to-date accurate image of the property. An accurate up-to-date image of the property will be restored when a subsequent state-value message is received. Because state-value messages overwrite one another and because they do not depend on any initial value, state value protocols tolerate transient communication faults. There is also no problem with transmitting multiple copies of the same state value message, as the receiver will simply overwrite old state values with newer ones. One factor which complicates state value communication is the loose coupling of the sender and receiver. The sender transmits state values and it does not know whether the receiver has received them. The receiver polls for the state message, and it assumes that whatever value it finds is a true reflection of the property. If the sender fails, however, the receiver will have no way of discovering this. We advocate the association of validity times (*sample lifetimes*) with state values to overcome this problem [63]. When the validity time of the state value message expires, the receiver may assume that a failure has occurred.

2. State values may be polled (periodic sampling), whereas state-increment values can only be transmitted using event-triggered sampling. This means that it is possible to design more deterministic systems using state values.

3. In control systems it is usual to monitor properties using state values rather than state-increment values. This has two advantages. Firstly, the need to compute a new state from the old state and the state-increment at the receiver

is eliminated, and secondly, the states derived from the properties can often be related to the plant states as identified in state-space modelling.

4. There is no need to queue state value messages, as the latest state value includes all the effects of all previous state value messages; hence older state value messages can simply be overwritten or discarded. This means that flow-control problems caused by buffers becoming full will never arise [63]. (See chapter 6.)

Normally, periodic sampling would be used to obtain state values, whereas event-triggered observation could be used to obtain either state values or state-increment values.

Consider a distributed system where the time between events is much greater than the sampling period, and where the number of bits needed to represent the sample value is small. Assume that the system provides perfectly reliable message transmission. In such a case it is possible to transmit the result of sampling in messages using either state values or state-increment values. The two approaches yield identical results. However in practical real-time control systems, where perfect reliability cannot be assumed, and where the number of bits needed to represent the sample value is indeed small, state values have distinct advantages, particularly with respect to the handling of message loss and computer system behaviour under overload conditions. The use of periodic sampling and state values ensures a steady network load and automatic recovery from message loss. Periodic state value sampling and event-triggered state-increment value sampling are compared in table 1.

4.3.3 Plant-control system communication

Consider a computer controlled system which consists of a physical plant, a front-end data acquisition system, and a computer control system. If the front-end data acquisition system is ignored, then event-triggered sampling implies that the plant communicates state or state-increment information to the control system as and when significant events in the plant occur. Periodic sampling implies that the control system captures an image of the plant properties at regular intervals in time (by polling the plant). The difference between these two sampling techniques there-

Performance Criterion	Periodic State Sampling	Event-Triggered State Increment Sampling
Message Size	Large	Small
Communications overhead with light network traffic	High	Low
Communications overhead with heavy network traffic	Low	High
Fault Tolerance	High	Low
Communications overload handling	Good	Poor
Predictability	Good	Poor

Table 1: Comparison of Periodic and Event-Triggered Sampling

fore appears to be in the initiation of communication. Event-triggered sampling apparently requires the plant to initiate communication with the control system. In section 4.2.6, it was established that the plant is not able to initiate communication with the control system, but that a front-end data acquisition subsystem can be incorporated into the system to quantify the plant state and initiate communication with the control system. It is important to realize that different levels of abstraction can be identified in plant-control system communication. The quantification of plant properties by the front-end data acquisition system represents the most fundamental level of plant-control system communication. Communication of information from the front-end data acquisition system to the control system is a more abstract form of plant-control system communication.

4.3.4 Plant-initiated communication

A careful examination of apparent plant-initiated communication leads to an understanding of the different levels of abstraction in the operation of data acquisition.

An event in the plant occurs when a property changes. In an event-triggered system, the plant is required to *notify* the control system of this event. The effect of the property change is communicated as either a state value or a state-increment value. The plant, however, cannot initiate communication with the control system. It merely *presents* its properties to the environment. A front-end data acquisition system is used to quantify the plant properties. This operation is initiated by the front-end data acquisition system, and it is performed periodically or continuously. The image of the plant properties that exists in the front-end data acquisition system can then be communicated to the control system using either event-triggered or periodic sampling. Thus sampling at the most fundamental level is always periodic or continuous.

4.3.5 Decoupling the plant from the control system

We maintain that the front-end system must be treated as part of the control system because it does not (or should not) change the performance of the plant itself. If the front-end system modified the performance of the plant, then the control system would have to be modelled as part of the plant. This would greatly complicate the design of the control system. For this reason, front-end systems are designed so that their effect on plant behaviour and performance is minimized. This is typically achieved through the use of buffer circuits. In theory it is impossible to design a front-end measurement system which has *no* effect on the plant. This problem has received much attention in theoretical physics [64,65]. In practice the effect can reduced to the extent where it is negligible.

4.3.6 Plant-control system boundary

Traditionally however, the data acquisition subsystem (sensors etc.) is viewed as part of the physical plant. In practical situations, the plant-control system boundary is therefore usually defined to include the data acquisition subsystem *in the plant* because the control system accesses data which has already been measured by the sensors. We claim, however, that the boundary is defined in this way because it makes the plant-control system interface clean and convenient. We maintain that theoretically it is more correct to include the data acquisition subsystem in the

control system component, since the plant is designed and implemented in a way which minimizes the effect of attaching external systems. According to this view, any subsystem which is used to quantify plant properties which is an intrinsic component of the physical plant would not form part of any data acquisition subsystem. Such a subsystem would constitute a natural component of the open-loop plant. (Our contention is that the front-end system *must* be viewed as part of the sampling system, otherwise we are, in effect, saying that the controlled plant and uncontrolled plant are fundamentally different physical entities.)

This argument leads us to conclude that fundamentally, all communication is initiated by the control system. Accordingly, all communication which appears to be initiated by the plant is a specific case of control system initiated communication, where the control system has delegated the task of periodic or continuous sampling to its front-end device, and has requested the front-end device to notify it of any significant events.

4.4 Sampling of properties

The analysis of sampling methods and data representation techniques presented above shows that four techniques can be used to sample properties.

1. Periodic sampling with state-increment value message communication
2. Event-triggered sampling with state-increment value message communication
3. Periodic sampling with state value message communication
4. Event-triggered sampling with state value message communication

4.4.1 Periodic sampling with state-increment value message communication

Out of the four possible techniques listed above, this is the only one which cannot be applied in practical applications. State-increments result from events in the plant, and new state-increments do not incorporate old state-increments. For this reason every state-increment value must be reliably communicated to the receiver. If state-increment values are to be polled, then it is highly probable that at some time,

more than one state-increment will occur between polling calls. (Queuing of state-increment values can violate principles of flow control–this is discussed in chapter 6.) Lost state-increment values or the multiple use of state-increment values will cause the state at the receiver to provide an incorrect reflection of the property.

4.4.2 Event-triggered sampling with state-increment value message communication

Event-triggered sampling is generally employed in conjunction with state-increment message communication. The advantages, disadvantages and applications of the sampling mechanism and data representation technique have already been discussed. The problem of state-increment values which are produced but do not get sampled (which may be encountered in periodic state-increment sampling) does not occur with event-triggered sampling. The very occurrence of the event which produces a state change causes the state-increment to be transmitted to the receiver. Although this technique is widely used in transaction processing systems, it is also well suited to communication of infrequent but sudden "events" in the plant. (That is aside from reliability and fault-tolerance problems.) Equation 4.1 shows that it is theoretically possible to use periodic state value sampling to solve any physical problem that can be solved with event-triggered state-increment sampling. State-increment values can only be effective if they are consumed after use. (They must be used once and once only.)

4.4.3 Periodic sampling with state value message communication

The application of periodic sampling techniques and representation of data as state values in computer control systems has already been discussed. State values must be *persistent* so that the receiver can always have some image of the property. The usefulness of this technique becomes clear when one considers a classical control application, such as a PID controller. The set-point may be a ramp. The source of the set-point is considered the sender, and the PID controller is considered the receiver. The absolute value of the set-point is communicated to the receiver as a state value whenever the receiver polls the sender. The polling frequency is determined by the dynamics of the ramp (i.e. it is a function of γ and δ.) — see chapter 5. The

set-point state value is not consumed when it is read by the *PID* algorithm. If communication of a state value is unsuccessful, then the correct value will be established with the transmission of subsequent state value messages. Failure of the sender is established by means of a timeout mechanism. (See chapter 5.)

4.4.4 Event-triggered sampling with state value message communication

This technique represents a mixture of the two previous techniques. One of the major drawbacks with event-triggered state-increment sampling its low fault-tolerance. If one of the state-increments is lost in transmission, or used more than once, then the state at the receiver will cease to be an accurate reflection of the property. For this reason it has been necessary to use PAR and commit-rollback protocols in these applications. Communicating state values by event-triggered sampling improves reliability and fault-tolerance, and to some extent eliminates the need for high-reliability protocols. If the state value which is transmitted as the result of some event, is lost or corrupted, then the correct state value will be re-instated once the next event has occurred. This is not an ideal solution as the correction is only effected with the occurrence of the next event. One of the fundamental differences between non-periodic sampling and event-triggered sampling is that in non-periodic sampling, communication will definitely occur again, only it is not known when it will occur. With event-triggered sampling, communication is only initiated with the occurrence of an event. It cannot be assumed that some event will definitely occur.

4.4.5 Sampling in computer control applications

Persistency of state values

The fourth technique, described above, raises some interesting issues regarding the persistence and consumption of state values. It has already been explained why state-increment values must be consumed, and why periodically sampled state values should be persistent. In systems which employ event-triggered state value sampling, the nature of the receiving process should determine whether the state value should be persistent or consumed. If the value is to be used once only, then it must be consumed, whereas if it is to be used continuously or at random, then it must be

persistent.

Values which must be persistent and consumed

The application of event-triggered state value sampling is best illustrated by means of an example. Consider a keypad which is used by a plant operator to enter a set-point. This set-point is used by two "receivers" in the system; the one is a PID control algorithm, and the other is an operator event-logging process. The PID control algorithm is required to read and use this state value continuously, whereas the logging process should read this state value only once, as the entry of a set-point is a single event. In this example, the property is the set-point p. It seems as if we have contradictory requirements, that is, that p be persistent and that it be consumed. We therefore maintain that it is up to the receiver, which places the sampled value in a *p_var*, to decide whether its *p_var* is persistent or whether it can be consumed. This conclusion also follows from the rather more obvious fact that the sender (plant) cannot cause the property to be consumed. (It may be argued that entering the value of p is an event in the system, and that the consequence of the event is merely translated to a local state value. We maintain that the set-point p is nevertheless a property of the plant environment, and it should be continually represented as a state value in the computer control system.)

As a solution to the "contradictory" requirement stated above, we propose that any receiver wishing to consume the state value (for example, the logging process) create its own instance of a *p_var* (actually an *image p_var* — see chapter 6) which is consumed when read. Any process which requires the state value to be persistent can create a persistent instance of the same *p_var* (*image p_var*).

Analog events

When examined closely, the example given above seems unusual. The effect of an operator entering a set-point via a numeric keypad cannot be viewed as a *continuous* time-varying analog signal. It is clearly not a digital event, which has only two possible states; either on or off. We will describe the effect as an *analog event*. An analog event is multi-valued (not only 0 and 1) and can change state almost instantaneously as is the case with a digital signal. Event-triggered state value

systems are most useful for handling analog events. Digital events can be adequately communicated using event-triggered state-increment values. Since digital signals have only two possible states, receipt of any message which is transmitted as the result of an event, regardless of whether it contains a state or state-increment value, could be used to toggle the state at the receiver. (This is only true if an event *always* denotes a *change* of state.)

Reliability issues

Event-triggered state value systems are at least as reliable as event-triggered state-increment systems. Consider a system which consists of a sender and a receiver which communicate over unreliable communication links using an unreliable protocol. If the link is physically disabled, then the receiver will continue to use the last received value indefinitely. The transmission of state values instead of state-increment values would do nothing to improve the reliability of the system under these conditions. Consider an alternative scenario, where a random transmission failure occurs due to the corruption of a single message. Subsequent message transmissions are successful, and the messages arrive uncorrupted at the receiver. State values are clearly superior to state-increment values under these conditions. With state values, the receiver only reflects the incorrect state until subsequent uncorrupted messages are received. With state-increment values, the receiver will continue to reflect the incorrect value even after the next uncorrupted message has been received, since an increment has been *lost*.

The disadvantages of event-triggered sampling and of state-increment value transmission are mutually exclusive. In an event-triggered system, the receiver will use the last received value if the sender fails. This necessitates the use of reliable network protocols. State-increment data representation can be problematic if the receiver somehow loses its initial value, because state-increments are relative to an initial value.

4.5 Conclusion

In this chapter sampling and data representation in computer controlled systems has been reviewed, and theory which enables plant-control system interaction to be

modelled has been developed. We claim that periodic state value sampling is best suited to computer control, although we realise that in certain applications event-triggered sampling may be more practical. We come to three important conclusions, namely:

1. All sampling is inherently periodic, and that event-triggered sampling is an *artificial* mechanism which is used to meet fast or continuous sampling requirements.

2. Plant-control system communication must be regarded to always be control system (receiver) initiated. This conclusion follows because front-end data acquisition devices are actually components of the control system and not the physical plant.

3. The sender should not make any assumptions as to whether its state value messages will be consumed or persistent. It is the nature of the *receiver* process which determines data persistency.

The theory developed in this chapter forms the basis of our *p_var* model of process interaction in distributed real-time systems. The use of different sampling mechanisms with various data representation techniques was analyzed, and their application in computer control systems was evaluated. The theory also shows that the conventional understanding of the plant-control system boundary is not necessarily correct for the *theoretical* analysis of the interaction between the plant and the control system.

Chapter 5

Real-time control

5.1 Introduction

In the previous chapter we developed a model showing how computer-controlled systems can be split into plant and control system components. The dynamics of the physical plant describe how the state of the plant changes with the progression of time. Plant dynamics are expressed by qualitative or quantitative models, or by experimental data.

For control to be effective, feedback must be applied to the plant by the control system before the plant state deviates significantly from the plant state which prevailed when the measurements were made. Computer control systems are thus required to operate within time constraints which are a function of the dynamics of some physical system.

In this chapter we attempt to define real-time. The differences between high-speed and real-time computing are examined, and current definitions of the term "real-time" are analyzed. We then present a definition of real-time which has no implications regarding speed of operation.

The requirements of present-day real-time systems and of real-time systems of the future are discussed, and the underlying support structure that is needed is identified. A nine-point list of guidelines and techniques which will assist in providing real-time support is proposed. A distinction is made between generalized real-time computer systems and real-time computer control systems.

The use of real-time as a metric for temporal progression is the examined, and the

temporal relationship that exists between the dynamic behaviour of a plant and its real-time operating constraints is explored. Mechanisms which implicitly recognize and exploit this relationship are incorporated into our model. It is shown how these mechanisms can be used to enhance system reliability and improve fault-tolerance. The chapter concludes with a discussion of several implementation issues, such as real-time clocks and clock synchronization algorithms. These issues currently pose technological, philosophical or logical problems, and are discussed extensively in the literature.

5.2 What is real-time?

Real-time is one of the most misunderstood concepts in computing. Many commercial and industrial products claim to provide real-time performance simply because they operate faster than others. Many computer control system designers, who are often quick to criticize commercial data-processing definitions of real-time, have themselves not grasped the essence of real-time.

This misunderstanding is graphically illustrated by Tom DeMarco in his foreword to Hatley and Pirbhai's "Strategies for real-time system specification" [19]. He recalls a discussion he had with some real-time data-acquisition engineers which forced him to get to grips with the nature of real-time systems.

"Most systems people use the term real-time rather loosely," the young manager said. We were seated over dinner with three members of her staff and some other managers who took part in the day's seminar. "They say they've got a real-time constraint when they're worried about impatient insurance brokers and bankers sitting in front of their terminals. A real-time system in their minds is just one that needs to be 'quick as a bunny'. If they fail to meet that constraint, their users might be inconvenienced, or even annoyed. When we use the term, it means something rather different." Her co-workers began to smile, knowing what was coming. "We build systems that reside in a small telemetry computer, equipped with all kinds of sensors to measure electromagnetic fields and changes in temperature, sound and physical disturbance. We analyze these signals and transmit the results back to a remote computer over a wide-band channel. Our computer is at one end of a one-metre long bar, and at the other end is a nuclear device. We drop them together

down a big hole in the ground, and when the device detonates, our computer collects data on the leading edge of the blast. The first two-and-a-quarter milliseconds after the blast are the most interesting. Of course, long before millisecond three, things have gone downhill badly for our little computer. We think of that as a real-time constraint."

5.2.1 Speed of response

This extreme example clearly conveys the idea that a real-time constraint is a constraint which, when not met, means the difference between a functional system and a non-functional system. On-line "demand-time" systems, such as airline-booking, banking, and other transaction processing systems, claim to be real-time because in order to maximize performance and minimize user inconvenience, they are required to operate as fast as possible. (Ramamritham [30] refers to these "demand-time" systems as "real-time systems". Our "real-time systems" he calls "hard real-time systems".) When the delay to process a transaction is reduced to the point where interactive operator-system co-operation becomes viable, designers of commercial transaction-processing systems would say that the system is a real-time system. Thus real-time would appear to be defined in terms of speed (of response). We reject this definition of real-time because we maintain that a real-time constraint should be a quantifiable constraint which explicitly affects the behaviour or performance of the system if it is not met. Therefore we maintain that real-time must be defined in terms of performance deadlines which *must* be met. It is not necessary to introduce any notion of speed or delay into this definition.

5.2.2 Time-critical systems

From this perspective, the system discussed by DeMarco in the above quotation is somewhat misleading. Although the real-time constraint in force is quantifiable and explicitly affects the behaviour of the system if not met, the description of the system leads us to believe that it is unique in that it functions at a higher speed than most other systems. Consider a system such as the one given in the above quotation, where instead of the first two-and-a-quarter milliseconds being critical, the first two-and-a-quarter hours are critical. Designers of commercial transaction-processing systems,

with their definition of real-time, would almost certainly laugh at any suggestion that this system is a real-time system. According to our understanding of real-time however, the time constraint which is in force is still quantifiable, and it still explicitly affects the behaviour or performance of the system if not met. In practice however, with a given hardware device which can only operate at a given rate, it is easier to meet this less demanding time constraint.

5.2.3 Interrupts

Many systems claim to provide real-time performance because they are interrupt-driven. This claim shows that the meaning of the term "real-time" (and of real-time operating constraints) is often misunderstood, even in the context of control applications. Most industrial real-time computer control systems make use of interrupts to switch the processor to a more urgent process [66]. Interrupts however, are not necessarily serviced instantaneously. The processor may be executing in a critical section, during which it cannot be pre-empted, or it may first have to service other more urgent interrupts (which have a higher priority). The maximum time that it takes for the system to service an interrupt is termed the *interrupt latency* of the system. Real-time engineers use interrupt latency as a metric for specifying the real-time constraints that a system is capable of meeting [66]. The use of interrupts does not ensure real-time performance; it merely facilitates the pre-emption of one task by another more urgent one, thereby allowing real-time constraints to be met. Interrupts are often used because "as fast as possible" is usually fast enough for the application. (Interrupt latency is usually negligible compared to the process dynamics, and thus interrupts often appear to provide instantaneous response.)

5.2.4 Understanding time constraints

The traditional understanding of a given real-time constraint of x seconds is that the process must be completed (or respond) within x seconds of some defined event. (Normally the initiation of the process.) There is however, a more rigid way of viewing this constraint. It could be understood to mean that after precisely x seconds the process should initiate some event. The commercial system designer's understanding of real-time and speed of response implies that "the faster, the better"

is a principle which always applies. The use of interrupts, which often provide near instantaneous response is therefore most appealing. There are many applications however, where a "window in time" exists during which some operation must be performed, or during which some value is valid. Thus it is possible that an operation could be performed too soon (not only too late). When an operation must be performed during some "window in time", the speed-dependent definition of real-time loses all meaning, (too fast is just as bad as too slow) and many of the perceived advantages of interrupts are lost.

5.2.5 Defining "real-time"

The term "real-time" has been loosely applied, and it is defined differently whenever it is used. Hinden and Rausch-Hinden [66] define a real-time system as one which has the capability to accomplish its job within the time constraints of the specific application and at the rate needed to move the input/output data. Kopetz *et al* [7] note that a real-time control system must respond to a stimulus from the control object within an interval dictated by the environment called the response time. They maintain that the system must guarantee this response time under conditions of peak load and for all anticipated fault conditions. A definition of the term "real-time" should emphasize the critical nature of real-time constraints without implying that speed of response is critical. Any definition which emphasises speed of response will be inadequate in describing systems governed by "window-in-time" time constraints.

We therefore define a **real-time system** as:
A system which has the inherent ability to ensure that operations are initiated and/or completed within specified, quantifiable time constraints which would explicitly affect the behaviour or performance of the system if not met.

In a computer control system, a **real-time constraint** defines the temporal bounds within which the system must register a change in the plant properties and generate a response based on that change.

This definition in no way alleges that "brute force" techniques are necessary to implement real-time systems. Rather, it states that all systems which exhibit certain architectural characteristics, no matter how little processing power they possess, will be capable of meeting some real-time constraints. Tighter constraints can be met

by using more powerful processors. Our definition states that a real-time system is one which uses certain techniques to ensure that time constraints are met. It says nothing about how to enforce *tighter* time constraints.

5.2.6 "Brute force" techniques

Real-time performance is playing an important role in large scale computer systems. Stankovic [24] notes that there is a popular misconception that advances in supercomputer hardware will take care of real-time requirements. The use of more powerful hardware does nothing to ensure that timing constraints will be met automatically. Stankovic concludes that, on the contrary, the availability of more powerful hardware will simply result in attempts to solve more demanding real-time problems (with tighter time constraints). With systems suffering from the same old *architectural* deficiencies, the "brute force" approach is doomed to failure. With the increasing complexity of real-time systems, which will be distributed, and which will employ artificial intelligence and other flexible computation techniques, it is crucial that real-time system design and implementation techniques be re-examined. We note that Stankovic's understanding of the term "real-time" concurs with our definition.

"...However, the objective of real-time computing is to meet the individual timing requirements of each task. Rather than being fast (which is a relative term anyway), the most important property of a real-time system should be predictability; that is, its functional and timing behaviour should be as predictable as possible..."

5.2.7 Design and implementation of real-time systems

We propose a number of guidelines and techniques which can be used to help ensure correct real-time system behaviour:

1. Specification methods for real-time systems and semantic theories for real-time languages should include a time metric. This area has been the subject of a great deal of research, and it is beyond the scope of this paper. The reader is referred to [60,70,17,19,22,18,67,68,71,66,72,73].

2. Sources of non-determinism in scheduling algorithms and hand-coded interrupt routines should be eliminated from real-time systems [24].

We maintain that polling and non-pre-emptive time-based scheduling provide effective mechanisms for dealing with systems which have well-known real-time constraints. Although execution of processes will never be "as fast as possible", problems which result from an excessive interrupt load will not arise. The non-deterministic nature of interrupts makes it difficult to predict failures and debug the system. A non-pre-emptive time-based scheduler which employs polling, offers deterministic performance, and is ideally suited to the implementation of cyclical systems with "window-in-time" real-time constraints. The only drawback is that a detailed knowledge of all the processes and their dynamics is required.

3. Non-determinism resulting from non-linear performance degradation due to slight deviations from uniform traffic in network communication systems must be minimized [24]. Periodic sampling helps to maintain network traffic at a constant level [63].

4. System resources should be employed in such a way that real-time constraints can be met. The designer should not increase the probability of hardware or software failure by overextending the system resources [24,63].

The response time of the controlling computer must be much faster than that of the environment being controlled. If the computer used in an interrupt-driven system is too slow, the system may function correctly for some time, simply because the interrupt load and frequency is low enough. When it fails, however, it will fail unpredictably and suddenly. This is because such a system has a maximum throughput which cannot be exceeded. This maximum throughput is difficult to ascertain as it is almost impossible to model the load on the system under all possible conditions. On the other hand, a system which uses a deterministic scheduler and periodic state sampling can use knowledge of the dynamics of the system to ensure that processes are allocated to processing modules in a way which enables the real-time constraints to be met (as long as systems are designed to operate under conditions of maximum load—see point 5.) Such a system will never "coincidently" function correctly, and then fail. Because the system exhibits deterministic behaviour, it will either function correctly or not all. Thus debugging of the system is greatly simpli-

fied. Processing modules which are unable to cope with their process load are easily identified because failures occur in a deterministic and predictable fashion. The graceful degradation of such systems also makes it easier to isolate processes which are trying to enforce unrealistic time constraints.

5. Real-time systems should be designed so that they comfortably meet the timing constraints of the system when it operates under conditions of maximum load [63,7].

6. Facilities for the management of time must form an integral part of any real-time system. Our model incorporates such facilities. This is discussed in section 5.6. (See Cohen and MacLeod, [63] and Stankovic, [24].)

7. Real-time languages which make time management facilities available to the applications programmer must be developed [24,63,74]. (See appendix C.)

8. Techniques must be developed to check whether a system composed of a number of real-time processes executing on specific processing modules will be capable of meeting its real-time constraints. (Such system verification techniques are likely to be highly dependent on the determinism of scheduling algorithms and interprocess communication mechanisms [75].)

9. Techniques which make it practical to implement re-usable real-time software modules must be developed. Methods for structuring software for non-real-time applications have been developed [76,20], and similar techniques are required for use in real-time control applications, where software is inherently modular.

5.3 Real-Time Systems and Real-Time Control Systems

There is a subtle difference between a real-time computer system and a real-time control system. A real-time *computer system* is a combination of various functional software units which must be configured in such a way that they can achieve the desired goal within given time constraints. A real-time *control system* consists of a

physical plant which has a predictable dynamic response (usually), controlled by a system which generates control actions based on the plant output and characteristics. These control actions are aimed at driving the plant to some desired set-point. The control system must be able to generate control actions within the time constraints dictated by the dynamic behaviour of the plant. Real-time computer systems exhibit flexibility which enables the software processes to be juggled around so that the time constraints can be met. In a real-time control system, the physical plant constitutes a process or set of processes which does not exhibit this flexibility. The *control* processes must thus be juggled around in order to meet the time constraints imposed by the physical plant. The structure and overall configuration of real-time computer control systems is therefore subject to the limitations of the physical plant. (See chapter 6.)

Real-time constraints in more general (non-control) distributed real-time computer systems are often met by dynamically relocating processes to processing modules which have lighter processing loads. In real-time control systems, the plant constitutes a process or number of processes, and plant-interfacing equipment limits flexibility [2,63,44]. A processing module which runs a sensor task is actually connected to a sensor. It can therefore be assumed that the processing modules in a real-time *control* system will perform the functions designated to them at design time.

5.4 Real-time as a metric of temporal progression

In the physical world which we relate to, events and durations are temporally ordered and related to one another by means of an objective clock which measures physical time. In chapter 3 we examined Lamport's theory of time and the ordering of events. He notes that events can be ordered by using either a logical clock or a physical clock.

Logical clocks turn the irreflexive partial ordering implied by Lamport's precedence operator (defined in chapter 3) into a total ordering, by means of the *clock condition*. (For any events a and b, if $a \rightarrow b$ then $C\langle a\rangle < C\langle b\rangle$ — a and b are events internal to the system and C denotes a clock.) Lamport shows that anomalous behaviour can result in a system where events external to the system affect the system. He shows that this anomalous behaviour can be overcome in two ways [41]:

Either

1. the information relating events external to the system to events internal to the system can be explicitly introduced into the system (generally this information is not known [4]), or

2. a system of clocks which satisfies the *strong clock condition* can be used. (For any events a and b in $\mathcal{L}$: if $a \rightarrow b$ then $C\langle a\rangle < C\langle b\rangle$ — a and b are events which may be internal *or* external to the system.) Lamport goes on to show that it is possible to construct a system of physical clocks which run independently of one another and satisfy the *strong clock condition*.

Because the information required in method 1 is generally not known, method 2 is used to order events in a distributed system.

In practical terms, this means that when all the events affecting a system are internal to the system, then a logical clock is sufficient to establish the order of events. The logical clock is internal to the system, and so events external to the system cannot relate to it in any way. External events can only be ordered by a clock external to the system which can be referenced by processes both internal and external to the system.

There is another problem with the use of a logical clock as a metric of temporal progression in real-time control systems. A logical clock does not provide a metric for determining the duration of a time interval [77,41,4,78]. This is because a logical clock system is only incremented as a result of the occurrence of events in the system (logical ticks). Kopetz [27] notes that any abstraction from the concept of real-time neglects an essential property of the problem of coordination and synchronization in real-time distributed control systems. He concludes that a metric of real-physical time is essential in real-time distributed control systems, because real-time is a critical resource. To understand this concept, we will employ the model defined by Kopetz and Ochsenreiter [79,80].

5.4.1 The time line

The progression of time from the past to the future is called the *time-line*. An *event* is an occurrence at a point in time, i.e. a happening at a cut of the time-line, which

itself does not take any time. An *interval* (or *duration*) is the time-line between two events; that is, a section of the time-line. In reality, events cannot take no time to happen, and so every event has a duration. In many instances though, the duration of the event is very short and it can be ignored. Both the event and interval views of time are widely used. Neither view is ideal for solving all temporal problems. The two views are complementary, and each is useful in solving a specific class of temporal problem. An interval can be viewed as the time during which an infinite number of events, each of infinitesimal duration, can occur. An event can be viewed as an occurrence which has an infinitesimal duration.

5.4.2 Measuring time intervals

In a computer control system, events which occur in the physical plant are external to the computer system, and therefore a clock system which only satisfies the *clock condition* cannot be used to establish the order of events. In real-time computer control, it is also not sufficient to simply ensure correct logical ordering of events in the system. Often real-time constraints demand that the time *interval* between two events must not be longer (and/or shorter) than some specified duration, or that the system must respond to stimuli within specified time limits. Such constraints can only be met if the clock system which is used provides a metric notion of time, thus allowing it to be used to measure the duration of time between two events.

A clock which counts periodic physical ticks, which by convention are taken to mark out constant time intervals, provides a reference for *physical time* [4]. Such a clock is incremented periodically, regardless of whether there are events occurring in the system or not. (It therefore satisfies the *strong clock condition*.) We assume that some international radiated time standard, such as UTC [4,79] provides a reference for physical time. We therefore use physical real-time as an objective clock in computer control systems.

5.4.3 Meeting the strong clock condition

The use of a single global physical clock which can be read by any processing module introduces an unwanted element of centralization into an otherwise distributed system [63]. Even if a single, perfectly reliable physical clock were used in a dis-

tributed system, the non-deterministic time taken for the synchronization message to reach the processing module means that each processing module may be synchronized with the global physical clock at a slightly different time. Thus perfect clock synchronization would still not be achieved. Methods for dealing with this difficulty are discussed in section 5.7.

5.5 Plant dynamics and real-time computer control

A sampled data signal is obtained by periodically sampling a continuous-time data signal. The discrete (sampled) signal will only approximate the original signal if the original signal is sampled regularly enough. The sampling frequency must be high enough to ensure that sufficient samples are available to enable the fastest dynamic components in the original signal to be accurately represented. Åström and Wittenmark [28] give a heuristic technique for determining an optimal sampling frequency for use in computer controlled systems. They note that if the sampling period is too long then it will not be possible to reconstruct the continuous time signal, whereas if the sampling period is too short, then the load on the computer is increased. They define N_r to be the number of samples per rise time, and they advise that

$$N_r = \frac{T_r}{h} \approx 2 \text{ to } 4 \tag{5.1}$$

where T_r is the rise time (of the closed-loop system) and h is the sampling period. In a first-order system, the rise time is equal to the time constant. In a second order system

$$T_r = \omega_0^{-1} e^{\phi/\tan\phi} \tag{5.2}$$

where the damping ratio $\zeta = \cos\phi$. (ω_0 is the natural frequency of the system, and ϕ is the pole angle.)

5.5.1 Shannon sampling

The criterion which must be met by the sampling frequency, if the discrete signal is to approximate the original signal effectively, is accurately described by Shannon's sampling theorem.

Shannon's sampling theorem states that the sampling frequency should be at least twice as great as the highest frequency in the continuous time signal. (Shannon states his theorem in terms of the Fourier transform of the continuous time signal. He states that a continuous-time signal whose Fourier transform is zero outside the interval $(-\omega_0, \omega_0)$ can be uniquely described by its values at equidistant points in time if it is sampled at a frequency higher than $2\omega_0$.)

$$f_s \geq 2f_{max} \tag{5.3}$$

where f_s is the sampling frequency, and f_{max} is the highest frequency in the continuous time signal. The Nyquist frequency, f_N is equal to half the sampling frequency, and it defines the maximum bandwidth of a system which can be sampled at a given sampling frequency.

5.5.2 Time delay in reconstruction

A sampling frequency which satisfies Shannon's theorem is not normally adequate in computer control systems, because Shannon's reconstruction employs the inverse sampling operator, which is a non-causal operator, and therefore results in a delay in signal reconstruction. For this reason, other sampling and reconstruction techniques (such as zero, first and higher order holds) are used. In view of the fact that for most signals, the reconstruction of the signal using these techniques is not accurate, higher sampling rates are used to limit the error. This is especially true in digital signal processing appications, where error minimization in signal reconstruction is a primary aim. In control applications this is usually the case [28], and so lower sampling frequencies can be used, although they are still considerably higher than the Shannon frequency. We take Åström's heuristic to define an optimal sampling frequency.

In our treatment of real-time computer control systems, we find it more natural to deal with sampling periods (also referred to as sampling intervals), rather than with sampling frequencies. The notion of time that is used in real-time computer control systems is the time-line which describes durations or *intervals*, which are delimited by events.

5.6 Real-time and the p_var model

5.6.1 Messages have an associated time stamp

Time affects the correctness of a real-time system [24,45,63,7], and the use of out-of-date values is as incorrect and potentially dangerous as is the use of incorrect data values [79]. In many industrial systems, the "last good value" is used. It is assumed that this value is "approximately correct", and that it is better than no value at all. Although this assumption may be valid in many applications, there are other applications where it could prove disastrous to use a slightly out-of-date value. We maintain that the system should be made aware of data which becomes invalid. A receiving process can then make use of a time-out mechanism to decide what action to take. The programmer may decide that it would be best to use the last correct value, or he may decide to initiate some fault recovery or notification action. Provision of an effective timeout mechanism requires that time management facilities be incorporated into the system at the lowest possible level [24,63].

It is for this reason that expiry or validity times form an integral part of every *p-var*. Data value-expiry time pairs are atomic units which cannot be split up. In a physical implementation of a DCCS, the very operation of reading the value would cause the expiry time to be inspected, thus ensuring that invalid data values are not erroneously used.

Sampling implies that the present value of a property can only be inferred from a sample that was obtained in the past. The result of each sampling operation is represented by a set of the form

$$\langle \textit{p-var} \text{ name, sample value, } t_0,\ t_d \rangle$$

where t_0 is the point in time at which sampling was performed and t_d is the point in time when the sample value loses its power to represent the *p-var* (fig. 5.1). We say that the sample is *current* at its time of use t_{use} if

$$t_0 < t_{use} < t_d \tag{5.4}$$

The lifetime, d, of a sample, can be specified instead of t_d according to the following equation

$$t_0 + d = t_d \tag{5.5}$$

The handling of the sample lifetime, d is discussed in section 5.6.4.

The sample is said to be *current* $\forall t: \; t_0 < t < t_d$, where t_0 is the point in time at which sampling was performed, and t_d is the expiry time of the message.

Figure 5.1: Sample lifetime and message validity.

5.6.2 The sample lifetime-sampling period relationship

Sample values in physical systems only remain current for a limited time. Real-time systems should use these time limitations to enforce correctness and to improve reliability and fault tolerance. One way of doing this is to associate a "sample lifetime" with each *p-var*. The sample lifetime, d, is an attribute of the *p-var*, and is a function of the physical dynamics which govern the behaviour of the corresponding property. Each sample lifetime should satisfy the following conditions:

1. If the *p-var* holds a value obtained by periodically sampling a continuous-time measurement of some property, then

$$d \geq T_s + \Delta t_{\max} \tag{5.6}$$

where T_s is the sampling period, and where $\Delta t_{\max}$ is the upper bound on the (usually non-deterministic) time delay which results from the finite network transmission delay and computation overhead. (Although Δt is often ignored in practical implementations, it becomes important when trying to establish a consistent state in a distributed system—a difference of Δt can quite easily make a consistent state appear inconsistent.)

If d meets this requirement, then two problems which arise as a result of the finite, non-deterministic network transmission delay, Δt, are eliminated. The problems are:

P1. Ensuring that there is always a valid message at the receiver, and

P2. Ensuring that the receiver does not use a message after d seconds have elapsed since the sender transmitted it. (That is, even if the *receiver* deems the message to still be valid according to its own local clock.)

These problems are discussed below.

2. If the *p-var* holds a value obtained by event-triggered sampling, then d should be less than the shortest time between two events. This enables the task reading the *p-var* to resolve individual events.

   Although event-triggered sampling can be used to obtain either *state values* or *state increment values*, state increment values must be "consumed" (i.e., used once and once only). State values, however, may or may not be consumed. Since the process which reads the *p-var* decides how the sample value is to be treated (i.e. if it is to be consumed or not), it should be free to "consume" the sample value if necessary. Therefore tasks which observe events must have the inherent ability to distinguish one event from another. Hence the requirement that d be less than the shortest time between two events.

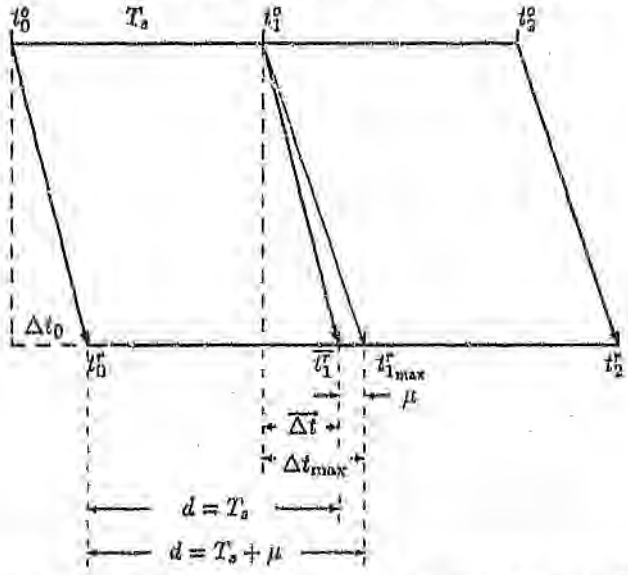
In an implementation, the sample lifetime can be used to invalidate the value contained in a *p-var* after d seconds. The signal that is being sampled can only be effectively represented if d satisfies one of the above two conditions.

From point 1 above it is clear that the sample lifetime, d, and the update or sampling period, T_s, are two separate quantities. A physical network has a transport delay, Δt, associated with the transmission of every message. Δt is usually non-deterministic. The type of protocol, physical transmission medium, and transmission rate all affect Δt. Usually it is possible to specify a probable upper bound on Δt under given load conditions. The characteristics of Δt are the cause of problems P1 and P2 defined above.

Problem 1: Ensuring that the receiver possesses a valid message

Since *every* message will be delayed by Δt during transmission, it is sufficient to require that T_s be equal to d less the maximum *deviation* in network transmission delay, μ, from the mean, $\overline{\Delta t}$ (and not less the maximum *total* transmission delay as implied by equation 5.6). (See fig. 5.2.) Problem 1 can be solved by choosing d and

The receiver will always possess a valid message if $d > T_s + \mu$.



μ is the maximum deviation from the average network transmission delay, $\overline{\Delta t}$.

Figure 5.2: Ensuring that there is always a valid message at the receiver.

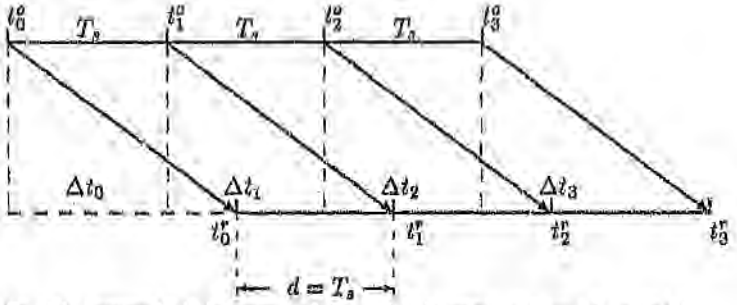
T_s such that

$$d > T_s + \mu \tag{5.7}$$

Thus it is quite possible to ensure that the receiver always possesses a valid message, even if $\Delta t \gg d$ (see fig. 5.3). (This mechanism can be problematic when used in periodic sampling of Boolean properties—see section 5.6.3.)

If this condition is met, then under conditions of normal operation the *p_var* will always contain a valid data value; that is, there will be no time that the *p_var* contains invalid data unless some fault or error condition arises. If a fault condition does arise, and no new data value is received, then the *p_var* will contain obsolete information. Clearly this problem could be overcome by extending the sample lifetime, d, or transmitting new values at more regular intervals, possibly at every $d/2$ seconds. Problem 1 is easier to solve than problem 2, since the validity of the message is established with reference to the receiver's clock only. (Note that in section 5.6.3 we show that there are times when we do require the property value message at the receiver to be invalidated.)

It is normal to sample data at a rate more conservative than Åström's recom-



In this example it is assumed that Δt_i is constant — i.e. $\mu = 0$.

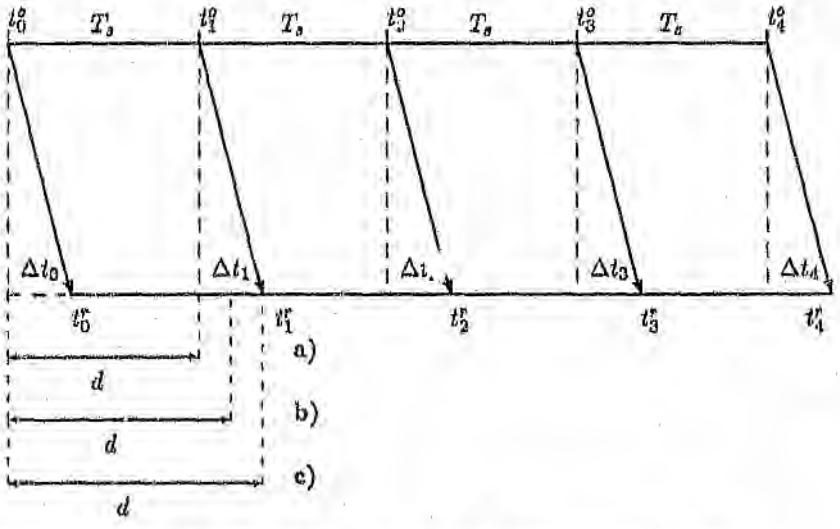
Figure 5.3: A system where $\overline{\Delta t} \gg d$.

needed sampling rate, because the network transfer delay is usually non-deterministic. It is possible to design a system where $T_s = d$. The current *p-var* value would expire and be regarded as invalid data just as a new, current *p-var* became available. Such a system would not tolerate any variation in network transmission delay.

Problem 2: Messages which the sender regards as invalid

Another more subtle assumption is implicit in a system where $T_s = d$. It is assumed that a time-lag of Δt resulting from the network transmission delay is unimportant (fig 5.4). Consider a sample with lifetime d which is transmitted over a network which has a maximum transmission delay of Δt_{max}. If the receiver uses the *p-var* value during the last Δt seconds of its lifetime, then it is effectively using a data value which the sender had intended to be invalid by this time. (This problem can also be overcome in a system where the clocks of the processing modules are closely synchronized, and where absolute expiry times are transmitted instead of sample lifetimes. Clock synchronization becomes much more critical under these conditions [4].) If the sampling rate of the system is high enough relative to the dynamics of the system, then this may well be justified. The assumption cannot be made in the general case however, and so we require that the *sample lifetime* (not the sampling period) be calculated from Åström's criterion, and that the sampling period, T_s be calculated by subtracting Δt_{max} from d.

$$T_s = d - \Delta t_{max} \tag{5.8}$$



a) $d = T_s$: $\forall t : t_i^s + T_s < t < t_{i+1}^r$ Sender regards message as invalid

b) $T_s < d < T_s + \Delta t_{max}$: $\forall t : t_i^s + d < t < t_{i+1}^r$ Sender regards message as invalid

c) $d \geq T_s + \Delta t_{max}$: $\forall t$: Sender regards messages as valid

It is assumed that $\forall i$: $\Delta t_i = \overline{\Delta t}$ (and therefore $\overline{\Delta t} = \Delta t_{max}$.)

Note: t_i^s refer to the times of observation at the sender, and
t_i^r denote the times of message arrival at the receiver.

Ensuring that the receiver does not use a message which the receiver regards as invalid requires that d and T_s be chosen such that $d > T_s + \Delta t_{max}$.

Figure 5.4: Avoiding messages which the sender regards as invalid.

Since this calculation takes into account the maximum *total* network transmission delay, non-determinism in the network transmission delay (P1) is automatically taken care of.

Error detection latency

Because the processing module which reads the *p-var* assumes that the sample value is valid for d seconds, d also defines the error detection latency. If d is very large ($d \gg T_s$), then it is only possible for the reader task to detect a failure well after it has occurred. Thus the choice of the value of d is also limited by the desired error detection latency.

Computational requirements

All computations and manipulation of data must be completed between updates, and so the sampling period gives some indication as to how powerful the computing hardware must be. The required values of d and T_s can thus be used to ensure that hardware and systems software are not used in applications which are too demanding.

5.6.3 Boolean-valued properties and state-conformity

Until only the sampling of analog signals has been considered. We have used our understanding of temporal and value granularities (chapter 4) to derive methods for selecting sampling frequencies which allow state-conformity constraints in analog systems to be relaxed. In the case of properties which reflect Boolean values (either 1 or 0), the temporal granularity of a property change is infinitesimal. If two processing modules in a distributed system have different versions of the property value, they will be operating with data which is *contradictory* (and not just a little different as in the case of an analog signal). It is thus imperative that all processing modules in a distributed system possess the same image of the property at all times. The physical limitations of the sampling and message transmission operations make it inevitable that the image of the property that is maintained in the various processing modules will lag behind the property itself. This is acceptable, provided that all processing modules have a consistent image of the property value. In [4], MacLeod addresses

this problem and presents a detailed solution for the case of event-triggered sampling.

An algorithm for ensuring state-conformity

There are two possible results of sampling operations; either a property change has occurred, or a property change has not occurred. When no property change has occurred, then state-conformity is not an issue. It is quite in order for one processing module to operate with the nth image of a property value, and for another to operate with the $(n+1)$th image of the property value, since the property value did not change between sample n and sample $n+1$. Hence, samples may overlap. Consistency becomes an issue when a property change is detected. When a property change is detected, the sender must wait until all images of the property have been invalidated by the passage of time before it asserts the new property value. We refer to the time that the sender must wait as the "hold-over" time, t_{ho}. If the sender waits t_{ho} seconds before asserting the new property value, it is certain that no valid image of the old property value will exist in the system (figs. 5.5 and 5.6).

MacLeod's solution

MacLeod [4] discusses an algorithm like the one proposed above for use in periodically-sampled systems, although he does not provide details. His solution requires that the local clocks of any two processing modules be set to within ϵ seconds of one another because absolute expiry times are transmitted. His relaxation of the perfect clock synchronization requirement—i.e. that a synchronization error of ϵ is allowed—forces the receiver to view incoming message expiry time pessimistically. He assumes that messages become invalid ϵ seconds earlier than their validity times reflect.

5.6.4 Handling of sample lifetimes

In our algorithm, d is transmitted as an absolute validity time (not an expiry time or observation-expiry time pair). It is measured from the time that the sender transmits the message. When the receiver receives the state value message, it extracts d from the *p_var* and subtracts Δt_{max} from it. It then uses this result and the current reading of its local clock to calculate the expiry time of the property value (relative

to the local clock). The receiver is thus pessimistic because it invalidates the message at a time which is calculated based on the assumption that the message took Δt_{max} to travel from the sender to the receiver. As a result of this pessimism, the receiver could invalidate the message almost Δt_{max} seconds earlier than necessary. (This corresponds to a case where message transmission was almost instantaneous.) Therefore if $d = T_s + \Delta t_{max}$, the receiver will never be without a valid message. (Except where a Boolean-valued property change is detected and the sender deliberately waits for all images of a property value to become invalid.) Making $d > T_s + \Delta t_{max}$ therefore ensures that even in the worst case, there will always be an overlap of samples.

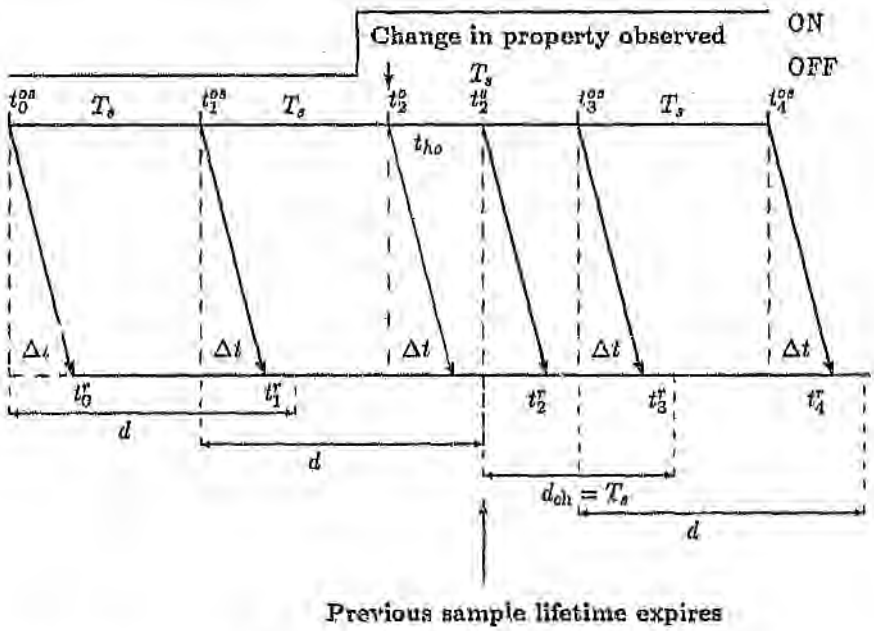
Application in the case of Boolean -valued properties

In the case of periodically-sampled Boolean-valued properties, making $d > T_s + \Delta t_{max}$ is counter productive, because when a property change is detected, the sender has to withhold the assertion of the new property value for a time called the "hold-over" time, t_{ho}.

$$t_{ho} = d - T_s \tag{5.9}$$

Bearing in mind that d cannot be less than $T_s + \Delta t_{max}$, it is easy to see that t_{ho} cannot be less than Δt_{max} (fig. 5.6). (If $d < T_s + \Delta t_{max}$ then images of property values could be invalidated even when no property change occurs.) When $d > T_s + \Delta t_{max}$ then t_{ho} becomes greater (fig. 5.5). In the case of periodically-sampled Boolean-valued properties, it is preferable to minimize the delay t_{ho} in asserting the new property value. (This analysis assumes that the "hold-over" time will always be less than T_s. When $t_{ho} = T_s$ then one sample has effectively been missed out.)

It is easy to see that the receiver may possess an invalid image of a property value for no longer than $2\Delta t_{max}$ after the occurrence of a property change. This will occur if the message containing the property value before the property change (message n) was delivered instantaneously, and the first message reflecting the state change (message $n + 1$) took Δt_{max} to deliver. This result concurs with MacLeod [4], who calculates the maximum time that an invalid message could persist for is 2ϵ, where ϵ is the maximum error in clock synchronization between any two distributed clocks.



d_{ch} is the shortened sample lifetime which is used when the sender asserts a state change. $d_{ch} = T_s$.

The new state is only asserted when enough time has passed to cause all sample values at recievers to become invalid.

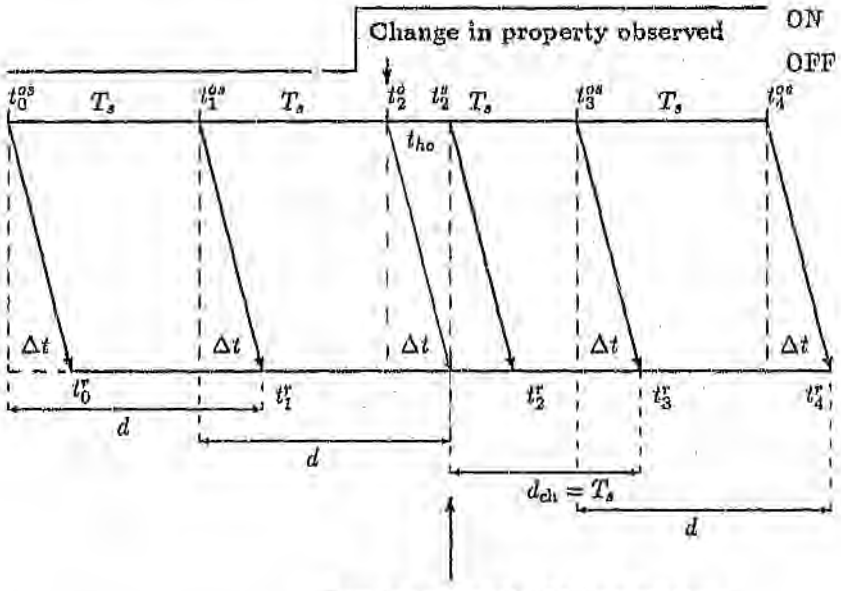
It is assumed that $\forall i: \quad \Delta t_i = \overline{\Delta t}$ (and therefore $\overline{\Delta t} = \Delta t_{max}$.)

Note: t_i^o refer to the times of observation at the sender, and
t_i^r denote the times of message arrival at the receiver.
t_i^s refer to the times of message transmission (and state assertion) at the sender.
t^{os} is used to denote times where t^o and t^s coincide.

t_{ho} is the "hold-off" time which the sender waits before asserting the new state.
$t_{ho} = d - T_s$

Case A: $d > T_s + \Delta t_{max}$

Figure 5.5: State consistency of boolean-valued properties: case A.



d_{ch} is the shortened sample lifetime which is used when the sender asserts a state change. $d_{ch} = T_s$.

The new state is only asserted when enough time has passed to cause all sample values at recievers to become invalid.

It is assumed that $\forall i$: $\Delta t_i = \overline{\Delta t}$ (and therefore $\overline{\Delta t} = \Delta t_{max}$.)

Note: t_i^o refer to the times of observation at the sender, and
t_i^r denote the times of message arrival at the receiver.
t_i^s refer to the times of message transmission (and state assertion) at the sender.
t^{os} is used to denote times where t^o and t^s coincide.

t_{ho} is the "hold-off" time which the sender waits before asserting the new state.
$t_{ho} = d - T_s = \Delta t_{max}$

Case B: $d = T_s + \Delta t_{max}$

Figure 5.6: State consistency of boolean-valued properties: case B.

Note that our algorithm, which requires that d be transmitted as an absolute validity time (not an expiry time), does not require that the clocks of the processing modules be synchronized (in contrast to MacLeod's solution [4]). Each processing module must however know the value of Δt_{max} so that it can view incoming messages with due pessimism. We believe that this requirement is more basic than any clock synchronization requirement.

5.6.5 Boolean properties and non-determinate states

In the event of a Boolean property change, the algorithm described above requires all corresponding property images to be invalidated by the passage of time before the new property value is asserted. The invalidation of property images is thus a regular occurrence in a correctly functioning system, and does not represent an error condition. In the analog systems described previously however, the invalidation of a property image (*image p_var*) indicates that an error has occurred (for example, a lost message).

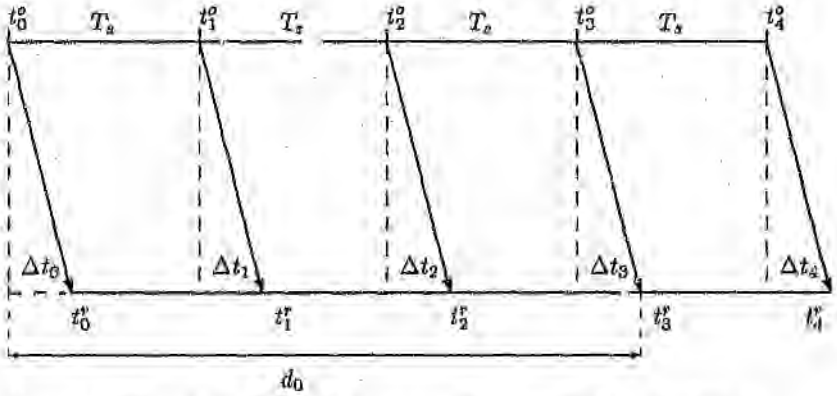
Another difference between analog and discrete systems is that in discrete systems only a finite set of stable states are acceptable *p_var* values. The transition time for a property to change from one stable state to another may be non-zero. Any sampling operation performed during this transition time should indicate that the property is in a non-determinate state.

Consider a discrete system where a transition occurs from a stable state $S1$, which persists during the interval (t^b_{S1}, t^e_{S1}), to a stable state $S2$, which persists during the interval (t^b_{S2}, t^e_{S2}). If $t^e_{S1} \neq t^b_{S2}$, there must exist a non-determinate state ND which persists during the interval (t^e_{S1}, t^b_{S2}).

In discrete systems it may thus be desirable to distinguish between a legitimate non-determinate state and an invalid state which represents an error condition. When a property change occurs and the sender delays the assertion of the new property for $t_{\Delta o}$, the property image can be allowed to assume a non-determinate state for $t_{\Delta o}$ seconds, before it becomes invalid and returns a *message expired* condition. Where the direct result of a sampling operation indicates that the property is in a non-determinate state, the property images should be allowed to assume a non-determinate state for a time $t_{ND} = \max(t^b_{i+1} - t^e_i)$. (The property images can

then be *invalidated* after the maximum *ND*-state persistence time has transpired.) If real-time fault-tolerance and reliability is desired, then discrete systems must incorporate mechanisms which distinguish between non-determinate states which arise in the normal functioning of the system, and invalid states, which arise due to error conditions.

5.6.6 Fault-tolerance and reliability enhancement



During the sample lifetime, d_0, three messages arrive at the receiver. The loss of any 2 of these 3 messages can be tolerated.

Figure 5.7: Fault-tolerant communication using sample overlapping ($n = 3$)

Consider a system which assigns a lifetime d to its samples, and which samples an *analog* signal with a sampling interval $d - \Delta t$. The sample lifetime has been calculated using Åström's criterion as given in equation 5.1. The system is required to exhibit a certain degree of tolerance to transient communication failures, which result in lost or corrupted samples. The system is time critical, and an error detection latency of even one sample lifetime is not acceptable. Therefore d cannot be longer than is dictated by Åström's criterion, yet error detection and recovery must be performed in this time.

System reliability and fault-tolerance can be improved by sampling faster than the rate which follows from Åström's criterion. Values obtained by sampling at a higher frequency would still be valid for the duration which follows from Åström. For example, if sampling were performed at twice the Åström frequency then a

second sample would be obtained before the first became invalid. Thus the value contained in the *p-var* will not expire and become invalid if one sample is lost or corrupted (fig 5.7). Increasing the sampling frequency is achieved by including an "overlap" in the sampling period. (Note that this technique cannot be used in the periodic sampling of Boolean-valued properties, where state-conformity must be assured, because there we are forced to allow images of property values to become invalid—see section 5.6.3.) An "overlap" of n can be introduced by subtracting the term $[d(n-1) + \Delta t_{max}]/n$ from T_s (as long as $\Delta t << d/n$). Consider an "overlap" of $n = 2$:

$$T_{ovr} = T_s - \frac{d}{2} \tag{5.10}$$

$$T_{ovr} = d - \Delta t - \frac{d}{2} - \frac{\Delta t}{2}$$

$$T_{ovr} = \frac{d}{2} - \frac{\Delta t}{2}$$

interval, sampling will be performed almost twice as regularly

sampling period is now such that two observations can be

perform

value contained in the *p-var* becomes invalid. A sampling period which provides an overlap of n is given by the equation:

$$T_{ovr} = d - \frac{d(n-1)}{n} - \frac{\Delta t}{n} \tag{5.11}$$

5.7 Implementation of a distributed real-time clock

Our algorithm for ensuring state-conformity in the reflection of Boolean-valued properties does not require the synchronization of distributed clocks. Nevertheless, clocks are still required to be synchronized for reasons explained earlier in this chapter. The implementation of a distributed time base which satisfies the strong clock condition is a non-trivial problem. A physical clock counts periodic physical ticks which may be produced by the oscillation of a pendulum, quartz crystal, or atom [79]. The periodicity of the ticks produced by such physical systems is not absolutely constant, and so two local clocks which have been synchronized will inevitably drift and reflect different times sometime after synchronization. (The rate of drift is dependent on the characteristics of the tick source.) Therefore, if the source of physical time for an ensemble of local clocks, then it is necessary to re-synchronize the clocks periodically

In order to correct the drift. The process of re-synchronizing the clocks introduces a further element of uncertainty into the physical time base, since the transmission delay which occurs in sending a signal from one processing module to another is not deterministic. Kopetz and Ochsenreiter [79,80] note that in systems where the variability of the network delay is considerably larger than the intrinsic drift rate of the quartz crystals (tick source), the divergence of the clocks in the ensemble will occur at a rate much greater than would be due to the effect of the intrinsic drift rate of the quartz crystals alone. They propose an algorithm for reducing the effect of the variability of the network delay to the point where the intrinsic drift rate of the quartz crystals becomes dominant once again. We do not attempt to resolve the difficulties involved in providing a fault-tolerant distributed physical time base. We merely state the problems that arise due to clocks which are not synchronized, and we list some of the solutions which have been proposed. The issues are discussed in depth by Lamport and Mellar-Smith [81], Kopetz [79] and MacLeod [4].

5.7.1 Independent real-time clocks

In loosely coupled distributed computer control systems, processing modules are autonomous, and therefore each processing module must have its own real-time clock. All the clocks run independently. Any centralized time-keeping system, such as a system-wide time interrupt from a master, would introduce elements of centralized control and reduce the reliability of the system. Even if a centralized physical clock is used, the variability of the network delay in making references to the clock means that it will always be possible to find time intervals where some clocks in the system have just ticked and others have not ticked yet. MacLeod notes that in this sense "time is infinitely divisible" [4]. This statement defines one of the most difficult problems in the theory of distributed real-time systems — how can state-conformity be achieved if it is not possible to synchronize clocks precisely?

MacLeod notes that two approaches are used to deal with the problem.

1. The system can avoid referencing the clock during the interval of uncertainty. This approach is best suited to tightly coupled systems, where the interval of uncertainty is small. In loosely coupled systems however, it would prove to be wasteful.

2. All algorithms can be designed to make use of the physical time reference in a way which makes them tolerant to clock deviations of one tick. Kopetz [27] has shown that it is not necessary to allow for a clock discrepancy of more than one tick. Algorithms have been developed which allow clocks to be synchronized to within ϵ seconds, where ϵ depends on the delay characteristics of the physical network and the clock synchronization algorithm employed [4].

5.7.2 Synchronization techniques

MacLeod lists three categories of clock synchronization techniques:

Independent accurate clocks: The clocks function independently, but do not drift excessively during an acceptable time interval. This method is not commonly used because of the high cost of accurate clocks. Standard quartz clocks drift too rapidly for most applications [4].

Common external reference: Processing modules depend on an external time source for synchronization. For example a dedicated synchronization line could be used to pulse all processing modules periodically.

Mutual feedback: Lamport and Melliar-Smith have proposed several algorithms which do not rely on a common time reference [81]. The processing modules monitor each other's clocks and agree on a common time. (For example, a processing module averages the times reflected by all clocks which reflect times not too different from its own clock.)

5.8 Conclusion

The meaning of real-time in the context of computer control systems has been examined. Popular misconceptions regarding terminology have been explained, and a new definition of the term "real-time" is given. The state of real-time system design is reviewed, and one of the most important proposals of this chapter is the nine point list of guidelines and techniques which can be used to assist in ensuring correct real-time behaviour.

The differences between real-time computer systems and real-time control systems are noted, and the use of real-time as a metric for temporal progression is

discussed. It is shown how the time constraints of a system follow from the dynamic characteristics of the plant. The model developed in the previous chapter was extended to provide implicit support for the management of time.

One important observation which is made in this chapter is that the required sampling period does not follow directly from the plant dynamics. The required sample lifetime follows from the plant dynamics, and the required sampling period can then be calculated by taking the network delay characteristics into account.

A major contribution of this chapter is the development of method for ensuring state-conformity in the periodic-sampling of Boolean properties, which does not require any synchronization of distributed clocks. A novel reliability and fault-tolerance enhancement technique based on oversampling is also presented. It is also shown how this technique fits in with our model, and how it is used in conjunction with the control engineer's analysis of the plant dynamics. A review of current techniques which are used to overcome problems involved with the implementation of a fault-tolerant physical time-base is given.

The results and findings stated in this chapter form an integral part of our model of process interaction in real-time distributed computer control systems. Some of the theory is also used and expanded upon in chapter 8, where we examine the state-conformity problem and its formulation.

Chapter 6

Information ownership and flow-control

6.1 Introduction

In this chapter the flow of information in real-time distributed computer control systems is analyzed. Processes typically interact with one another by exchanging data. It is thus crucial to have an understanding of the general principles and specific peculiarities of data flow in computer control systems if process interaction is to be modelled. The subject is examined by exploring two principles which clearly show where computer control systems bear similarity to other systems, and where they are unique.

The first part of the chapter examines the viability of performing flow-control in real-time computer control systems. The use of buffers is discussed, and the nature of the processes which constitute a distributed computer control system is examined.

In the second part of the chapter we show that in real-time distributed computer control systems, data items (or *p_vars*) are effectively owned by a single process. Understanding of this ownership principle allows us to make assumptions about data-flow in control systems.

The results which follow from our examination of the principles of flow-control and information ownership are used to extend our *p_var* model. The resulting model effectively describes the interaction of processes in real-time distributed computer control systems. The *p_var* model can be used to specify interprocess communication

primitives for use in real-time distributed computer control systems, and these can form the basis of a generalized real-time control language.

6.2 Flow control

The central component of a computer system is its central processing unit (CPU) which processes data at a fixed rate. Data is moved from some source, called the input, processed by the CPU, and moved to some destination, referred to as the output. The input source and output destination also have finite processing rates (or limited throughput capabilities). If the speed of the input source is lower than the speed of the CPU, and the throughput capability of the CPU is lower than that of the output destination, the system will function without suffering from any bottleneck problems.

6.2.1 The bottleneck problem

In general, the "bottleneck problem" can be stated in terms of a producer process, P, and a consumer process, C. Whenever the rate of (data production carried out by) the producer is greater than the rate of the consumer, a bottleneck occurs ($dP/dt > dC/dt$).

In this problem the producer and consumer processes are not necessarily intelligent processes. In a stand-alone computer workstation, the producer may be the user entering information by means of the keyboard and the consumer may be the process executing on the CPU. At the same time the video display system may consume data which is produced by the CPU. The throughput of a system is said to be "bounded" by the slowest operation. For example, a computationally-intensive process which directs its output to RAM storage is usually compute-bound. That is, the system is quite capable of storing data at a rate which is greater than the CPU's data production rate. The speed is limited by the computational intensity of the algorithm, and not by the system's data transfer and storage capabilities. If however, the same process is directing its output to disk, then the system is likely to be disk-bound. Since the random seek and write time of the disk is greater than the RAM storage time by several orders of magnitude, the disk may not be able to cope with the producer process's output rate. Similar situations arise in computer

controlled systems. Two common instances are:

1. Where the computational intensity of the control algorithm means that it is unable to consume information at the rate at which the data acquisition process produces it.

2. Where the limited dynamic response capabilities of an actuator prevents it from operating at a rate which is sufficient to consume data being produced by the control algorithm process.

Two main approaches are used in dealing with the bottleneck problem:

6.2.2 Buffering

If the rate of data consumption at the receiver is lower than the sender's rate of data production, then the receiver can queue the incoming data in buffers. The receiver can then remove the data from the buffer when it is ready to process it. This approach can be used when the following conditions apply, either:

- The rate at which the sender produces values varies. On average the receiver is capable of consuming the produced data at the required rate. It is only when the sender produces data at an above-average rate that the receiver is required to buffer the incoming data; or

- The sender produces data at a constant rate, but transmission of the data is intermittent; that is , it operates for fixed, finite (usually short) intervals in time. Although the constant rate is greater than the potential operating rate of the receiver, the receiver can buffer the data produced in the short interval during which transmission occurs so that it can be processed in the interval during which no transmission occurs.

Clearly buffering will not suffice in an application where the receiver perpetually produces data at a rate which is greater than the maximum rate of consumption at the receiver. Under these conditions a finite length buffer is bound to become full. (A buffer of infinite length would be required.) The full buffer condition can be dealt with in two ways:

1. Discard the data — data arriving at a full buffer is ignored. This is an unpredictable condition, and is usually undesirable.

2. Overwrite data — data arriving at a full buffer simply overwrites and replaces the old, unused data. This approach is commonly used with circular buffers. Overwriting of data can still yield correct results, especially where each data value includes the cumulative effects of all previous data values. (For example, where the data values are state values.) If overwriting of data yields correct results, then it seems unnecessary to buffer incoming messages.

Buffer length is usually a function of the severity of the bottleneck, the variation in the rate of data production and the data representation technique. In applications where state value messages are communicated, the latest message incorporates the effects of all previous messages (sec. 4.3.2). A buffer which holds only one message is therefore sufficient in many applications where messages are overwritten.

6.2.3 Flow-control

If the receiver cannot consume data at the rate at which the sender produces it, then some type of flow-control must be exercised. Flow control requires that the producer and consumer processes make an effort to regulate the flow between them so as to minimize the bottleneck effect and the loss of information. Flow-control can be exercised by either the sender or the receiver.

1 Sender — The sender may employ some type of positive acknowledgement and retransmission-on-error protocol to effect communication. With such a protocol, the receiver must acknowledge receipt of each message. If the sender does not receive this acknowledgement, then it re-transmits the message. When the buffer at the receiver is full, messages are discarded, and so no acknowledgement is issued. The sender thus has a way of detecting that the receiver's buffer is full (fig. 6.1).

2 Receiver — A protocol may be used whereby the receiver can explicitly inform the sender that the receive buffer is full. This may be done by means of a special message which is communicated from the receiver to the sender (fig. 6.2).

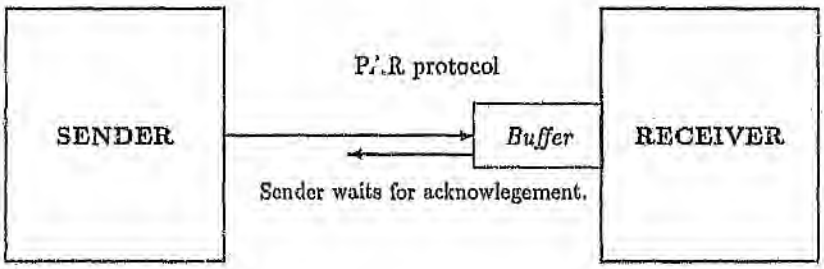


Figure 6.1: Flow-control enforced by the sender.

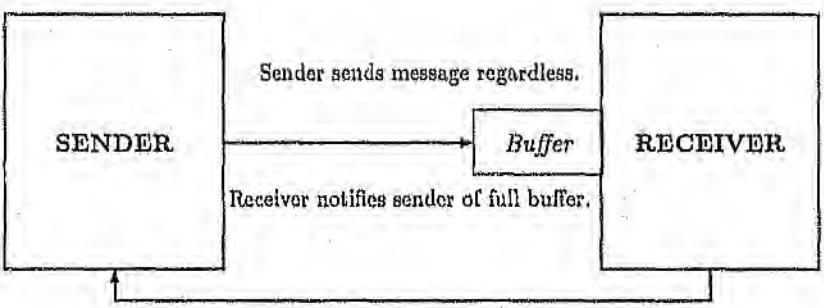


Figure 6.2: Flow-control enforced by the receiver.

Flow control can be employed in conjunction with a system of buffers. Wherever flow control is to be employed, both the sender and the receiver must be capable of intelligent communication, that is, they must be able to interpret and understand messages of instruction from one another. Of cou ; the sender must also possess the inherent capability of regulating its output.

6.2.4 The nature of processes in control systems

In section 5.3 we noted that real-time computer control systems differ from real-time computer systems because they do not offer the same flexibility in task allocation. Another major difference is that in a real-time computer system the processes are usually software tasks which are inherently capable of regulating their output. In real-time computer control systems, some processes are software tasks, whilst others are subsystems which constitute part of the physical plant. Subsystems which are part of the physical plant are not inherently capable of regulating their output of "data". The "data" produced is manifest in the plant's properties. The plant's properties constitute a reflection of the physical state of the plant which is presented to the front-end data acquisition system. The front-end data acquisition system quantifies these properties and communicates the values (either state or state-increment) to the receiver (control system) using either periodic or event-triggered sampling. Any attempt by the receiver to flow-control the front end system by requesting that no more data be sent until the receive buffer has been emptied of some of its contents would be tantamount to the control system requesting that the plant "freeze" its state whilst it is functioning. (This is apparent flow-control—see 6.2.5.) Similarly, an attempt by the front-end system to delay transmission of data to the control system (because of a full receive buffer) will be understood by the control system to mean that no change had occurred in the physical plant.

6.2.5 Apparent flow-control in real-time control systems

Since it does not make any physical or logical sense for a plant to delay the reflection of its current physical state, it follows that the receiver (control system) is not capable of flow-controlling the sender (physical plant). In practice, a front-end data acquisition system is considered the sender. This system is capable of interpreting

and understanding control messages sent by the receiver, and it can regulate its data output. It was explained in chapter four why theoretically, this system should be regarded as part of the control system. Nevertheless, in practical applications, the front-end data acquisition device quantifies the plant properties and communicates the resulting data to the control system. Sometimes the communications protocol tries to exercise some form of flow control. Although the front-end data acquisition device has an up-to-date image of the plant properties, the control system will not be aware of this, and so it will be operating with an out-of-date image of the plant. (Validity times can be used to invalidate the out-of-date values.) Flow-control in this situation only *appears* to be permitted in this situation by virtue of the front-end data acquisition system, which is considered the sender. If however, the *plant* is considered the sender (and the data acquisition system part of the receiver), then it is clear that flow-control is not feasible.

6.2.6 The use of buffers

Buffering is therefore the only mechanism which can be used to deal with the bottleneck problem in control systems. In any system where $dP/dt < dC/dt$, the bottleneck problem will not arise. If the system meets the conditions described at the beginning of section 6.2.2 then buffering will be effective. In systems which transmit state-increment values using event-triggered sampling, it is assumed that the system will always satisfy these conditions if ever $dP/dt > dC/dt$. It must be noted that even if these conditions are guaranteed, state values still provide greater fault tolerance, and so they are preferable in control systems, where communication is not always reliable. In many applications the conditions given in section 6.2.2 cannot be ensured, and hence a finite length buffer will not be sufficient. The only alternative is to allow new messages to overwrite old messages (new data to replace old data). This forces us to resort to the use of state value messages rather than state-increment value messages. When a new state-value message overwrites an old one, no information is lost, since the new state-value represents the absolute value of the property.

6.2.7 Dynamic limitations of actuator subsystems

It should be noted that problem two described in section 6.2 is subtle, because there the *process dynamics* cause the bottleneck. For example, consider a state value which represents an actuator set-point. If the state value is overwritten within the time that the actuator takes to respond fully, the actuator may never reach its set-point. In such a situation either a faster actuator must be used, or the control design must be examined. If however, the state value is only overwritten within such a short space of time under occasional conditions (where it is difficult to determine an optimal buffer size or too costly in terms of overhead to implement buffering) then state-values offer a workable solution.

6.2.8 Implicit flow-control

Flow control is implicitly provided by a system which uses state value messages, if a sufficiently short periodic sampling interval is chosen during system design and the receiver is designed to handle the corresponding steady message traffic [7]. (See fig. 5.1). Where the communication system is not reliable, state value messages offer several other advantages (see chapter 4).

6.3 Ownership of information

We use the general term *real-time property variable* or *p-var* to mean a holder for the value of a property. Each property which the control system wishes to measure or monitor is mapped to a *p_var*. It follows that each *p_var* represents the value of a unique property of the system. The concept of a property can be broadened to include entities for which the human operator supplies a value or quality which is based on his own subjective judgement, as well as internal entities, where the value is calculated by the computer system.

6.3.1 Assigning ownership of p_vars to processes

It is not logical to have a state-value representing a temperature (for example) which is derived from a temperature sensor in the plant, and to then allow an operator (for example) to overwrite that value. Integrity of the data values can only be assured

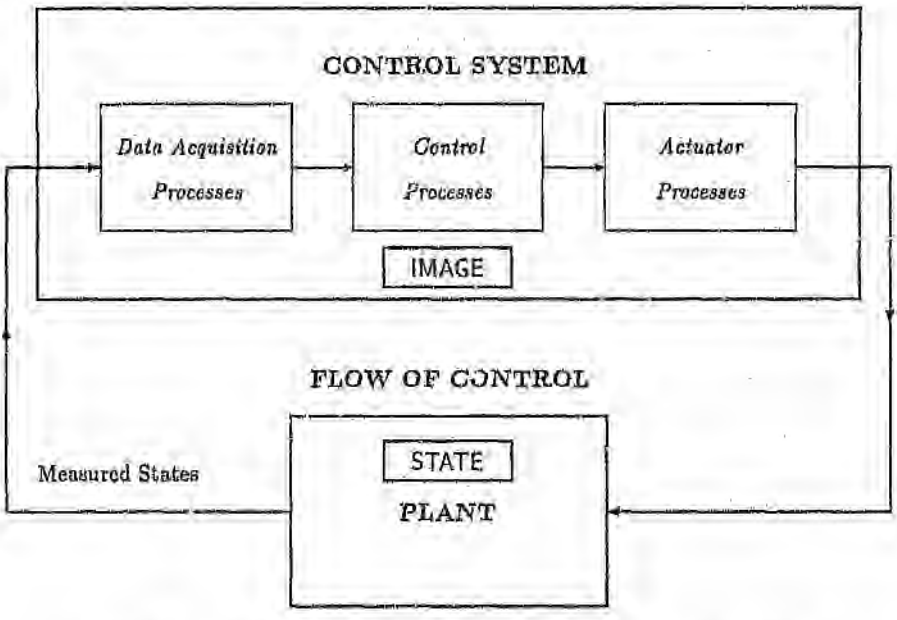


Figure 6.3: Diagram showing the system synergy and flow of control

by granting ownership of the *p_var* to the process which quantifies the property and writes the result into the *p_var*, and by refusing to allow any process other than the owner process to write to (or update) the *p_var*. We therefore say that a *p-var* is "owned" by only one subsystem; that is, that subsystem is able to set the value of the *p-var*.

Subsystems other than the owner subsystem can observe a *p-var* but they cannot modify its value. *P-vars* are also used to hold the values of properties of subsystems which are not generally considered part of the physical plant. In the case of a physical property the value is physically encoded. For operator-supplied entities the value or quality is encoded in the operator's consciousness, and for entities generated by the computer system the values are stored in a memory location.

6.3.2 The single-writer concept

Every plant subsystem has unique properties. Each property is quantified by a process which writes the resulting value into a *p_var* which it owns. Any other process or subsystem which needs to use the *p-var* may do so by instantiating an

image p_var.

Image p_vars

Image p_vars are instantiated by remote processing modules. The *p_var* owner process broadcasts *p_var* maintenance messages which result in the *image p_vars* being updated. A *p_var* can thus be viewed as a global variable, which can only be updated by a single owner process. *Image p-vars* can be maintained by processes running on remote processing modules which enable these global variables to be used throughout a distributed system.

Redundant sensors

According to the single-writer concept, there is no physical system where a single parameter or variable can be set by more than one subsystem. This is clear in the case of a control system, where a sensor is used to measure a single physical quantity [82,7]. Where a second sensor is used to measure the same quantity, the two resulting measurements are distinct and different. The two measurements will inevitably differ because no two sensors will have identical characteristics. The difference is usually small and is disregarded in practice. Nevertheless, one sensor could fail and generate meaningless output, whilst the other could still be functioning correctly. This clearly shows that multiple measurements of the same physical quantity have to be treated as distinct and separate quantities. It is the process of measurement (sampling) which separates multiple measurements of the same quantity into several separate and distinct quantities.

This can be stated formally as follows:
If x is a physical property, and $\mathcal{F}$ and $\mathcal{G}$ are two independent measurement processes, then

$$\mathcal{F}[x] \neq \mathcal{G}[x] \tag{6.1}$$

But $\mathcal{F}[x] \rightsquigarrow \bar{x}_1$ and $\mathcal{G}[x] \rightsquigarrow \bar{x}_2$ where $\bar{x}_1$ is the *p-var* containing the value of property x as measured by measurement process $\mathcal{F}$, and $\bar{x}_2$ is the *p-var* containing the value of property x as measured by measurement process $\mathcal{G}$. (An overscore is used to denote *p-vars* . Processes are said to "lead to" *p-vars*. The "leads to" operator is denoted by the "$\rightsquigarrow$" symbol). Therefore $\bar{x}_1$ and $\bar{x}_2$ are independent entities.

Properties of the plant environment

The single-writer concept also extends to the plant environment. Consider a control system which uses an operator-supplied set-point. The set-point is a property which resides in the operator's mind. The value of this property is held in a *p-var* in the system. Although in practice no two operators should ever be allowed to vary a set-point simultaneously, distributed systems may allow a set-point to be varied from more than one operator console. Thus it is theoretically possible for two subsystems (operators) to change a single *p-var* (the set-point) simultaneously. Such a system appears to violate the single-writer concept.

This apparent violation is easily explained by considering what happens if the set-point is modified from two operator consoles simultaneously. The system must provide an arbitration mechanism for dealing with clashes, contention and contradictory instructions which may occur when the human discipline is violated. This mechanism is actually the subsystem which decides what the value of the set-point must be when different set-point changes are received simultaneously. The decision is made by an algorithm which uses user-defined selection methods and priorities. The selection subsystem is required to observe both operator console set-points in order to produce an actual set-point. Hence each operator console set-point must be regarded as a unique property and should be represented as a separate *p-var*.

The incorrect view of the system (fig. 6.4) can be explained simply by imagining a set-point *p-var*, $\bar{r}$. The set-point can be modified from two operator consoles, A and B. It would seem that $A \rightsquigarrow \bar{r}$ and that $B \rightsquigarrow \bar{r}$, thus violating the single-writer condition. The correct view of the system (fig. 6.5) states that $A \rightsquigarrow \bar{r}_A$ and $B \rightsquigarrow \bar{r}_B$. The actual set-point is some function of $\bar{r}_A$ and $\bar{r}_B$.

$$f(\bar{r}_A, \bar{r}_B) \rightsquigarrow \bar{r} \tag{6.2}$$

6.3.3 Local writers

The single-writer concept does not in itself guarantee that *p-vars* will not have to be updated over the network. Perhaps the owner process will reside on a remote processing module, and updating the *p-var* will still involve granting exclusive access to the remote writer. This is not trivial because the process which disseminates

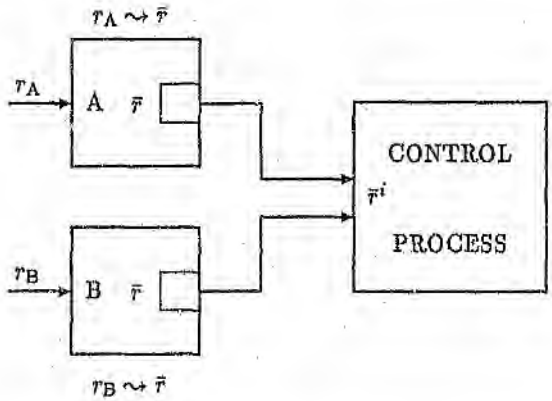


Figure 6.4: Incorrect model of dual set-point entry.

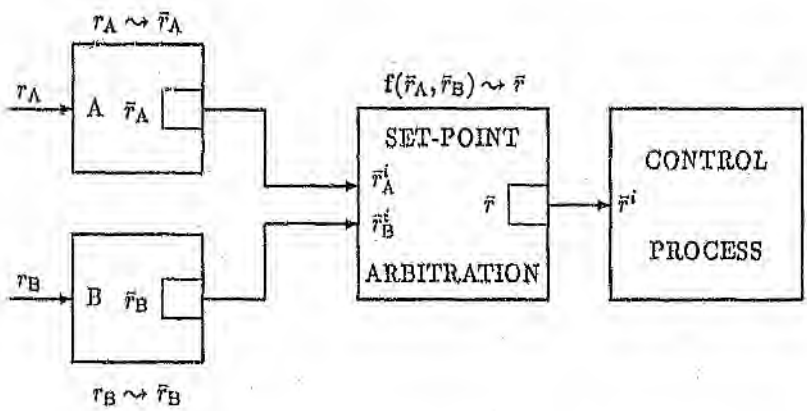


Figure 6.5: Correct model of dual set-point entry.

image p_var maintenance messages could be in the midst of reading the current *p_var* value when the remote writer begins to modify the value.

In conventional distributed computer systems, reliability and throughput are improved by allowing tasks to be dynamically relocated to other processing modules in the system. Conventional distributed operating systems employ load-balancing algorithms to maximize use of the available processing hardware. In distributed computer control systems, plant-interfacing equipment limits flexibility [2,44,63], and there is no sense, for example, in having a system which allows a processing module which is connected to a sensor, run an actuator task. It is therefore reasonable to assume that the processing modules in a distributed computer control system will perform the functions designated to them at design time.

This assumption simplifies the distribution of real-time information and helps improve the performance of real-time DCCSs for two reasons:

1. It can be assumed that each *p-var* will always be modified by a local subsystem. Consider a system where each *p-var* is owned by a subsystem, and where tasks cannot be dynamically relocated. If a system is designed so that each *p-var* resides in the memory address space of that processing module which is permitted to modify the *p-var* value, then updating the *p-var* only involves writing to the local memory address space. If the system is designed so that all *p-vars* are only updated by local tasks, then the maintenance of *image p-vars* is greatly simplified. Because *image p-vars* are only used to receive data values, it follows that there is no need to provide mechanisms to control the *updating* of distributed data. If the updating of the *p_var* value and the maintenance of *image p_vars* corresponding to that *p_var* are performed by the same process (or even within the same processing module), then it is not difficult to ensure that the *p_var* update is completed either before or after reads have been completed (that is, that reads and writes are atomic operations).

2. The overhead incurred by systems which perform dynamic process assignment or process attraction is large. In such systems, a process administration system is required in order to ensure that all processing modules are being used effectively, and that all processes which should be executing have been assigned to a processing module. The assumption that the process-processing module

configuration is fixed results in lower process administration overhead.

6.3.4 Lamport's theory and p_var ownership

Lamport's theory of interprocess communication was described in chapter 3. In this theory, synchronization is achieved through the use of *communication variables* [13]. Lamport's communication variables "belong" to a single process, and only the "owner" process is able to write to the variable. This mechanism allows the mutual exclusion problem to be solved without assuming the availability of facilities which ensure atomic operations (reads and writes). Although Lamport is attempting to solve a different problem to ours, we note his terminology and the similarity of his solution to our own. He requires that all writes to a single communication variable be totally ordered. The fact that the owner process is the only process which is permitted to write to the communication variable guarantees that this condition is always met. Our solution, which is based on the single writer concept outlined above, also assumes that the owner process is the only process which is permitted to update *r_p_var*. Because we make such an assumption, the need to support distributed writers to a *p_var* is eliminated.

Note that because our *image p_vars* are effectively messages on the network, "concurrent" reading and writing of a *p_var* is not a problem, since it is the actual *p_var* which is updated, but processes which read the *p_var* actually read the *image p_var*. The *image p_var* is created as a result of a message which is received from the owner process. Since it is up to the owner process to issue messages which cause *image p_vars* to be maintained, ensuring that the owner first completes the update of the actual *p_var* itself is not difficult. It is therefore quite reasonable to assume that the operations performed by a single process are serialized (and totally ordered). This corresponds to Lamport's observation that it is not difficult to ensure that all writes to a communication variable are totally ordered if all writes are performed by the same process.

Converse of Lamport's result

Our theory which states that each *p_var* is written to by a single, local owner process can be viewed as the converse of the assumption underlying Lamport's protocol.

Because each *p_var* is only written to by a single, local process, we claim that it is unnecessary to provide for the mutual exclusion of concurrent distributed writers. Lamport shows that mutual exclusion of concurrent writers can be assured through the use of communication variables which are owned and updated by a single, local process.

We do not compare our theory to Lamport's protocol because we are trying to develop a theory which describes the interaction of processes in computer control systems. Lamport's protocol is a solution to the mutual exclusion problem. In [13], he provides rigorous mathematical proofs for his protocol. We quote details of his solution only to show that the single-writer concept has been used in the field of computer science to solve problems in distributed computing. This discussion is meant to highlight the similar assumptions which were made in both theories as a result of the single-writer concept.

6.4 Conclusion

Two classic problems associated with data-flow in control systems have been considered in this chapter. The first is the bottleneck problem, where the sender produces data more rapidly than the receiver is capable of consuming it. The second concerns the requirement to support the updating of *p_vars* by remote processes. A notation for expressing process-*p_var* relationships was developed in order solve this second problem. We conclude that distributed computer control systems possess two unique data-flow characteristics. These characteristics enable us to make assumptions which are used to extend our model.

The conclusions we arrive at are:

1. The control system is not capable of exercising flow-control over the physical plant. This fact is a further reason for using state-value messages which overwrite one another. Apparent flow-control, where the control system exercises flow-control over the front-end data acquisition system, although feasible, should be avoided.

2. Each *p_var* is owned by a single process. The owner process is the only process which is allowed to write to the *p_var*.

3. The process-processing module configuration in a real-time distributed computer control system is fixed due to hardware and connectivity limitations. These limitations make it reasonable to assume that it will always be possible to design systems so that the *p_vars* are written to by local writers only.

The assumption of a single, local writer eliminates the need to support remote writers. It is shown how this result concurs with Lamport's theory of interprocess communication in distributed systems. We claim that enforcing the single-writer concept is an effective way of improving integrity of *p_var* values.

The incorporation of the single-writer concept into our model, enables *p_vars* to be used to model the process interaction in real-time distributed computer control systems. In the following chapters we will show how interprocess communication primitives which make use of these special data-flow characteristics can be designed on the basis of the *p_var* model.

Chapter 7

The p_var model

We now outline the *p_var* model of process interaction in real-time distributed computer control systems. The model is based on the theory of real-time dynamic systems and their computer control, which results from the unification of the concepts discussed in the preceding chapters.

7.1 Computer controlled systems

- A computer controlled system has a *plant component* and a *control system component*.
- The control system component consists of a computer control system, which is typically distributed, and a number of front-end data acquisition subsystems.
- Where the front-end data acquisition subsystem is sampling the plant, the control system is deemed the *receiver* and the plant the *sender*. The receiver cannot flow-control the sender and the sender cannot flow-control the receiver in this case.

7.2 Distributed computer control systems

- A distributed computer control system consists of a number of loosely-coupled *processing modules*. Each processing module is a functional computer with its own local memory.

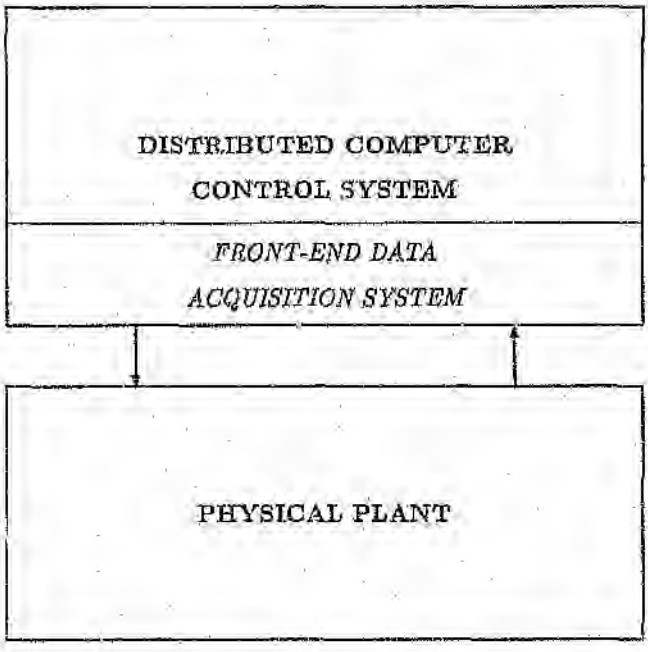


Figure 7.1: Computer controlled system

- The processing modules in a distributed computer control system may communicate with one another only by message-passing over a communications subsystem such as a local area network.

7.3 Processes

- Each processing module is capable of executing one or more software processes at any one time (either because the processing module is a multiprocessor computer or because it is executing some multitasking operating system).
- The process-processing module configuration is fixed during system execution.

7.4 The physical plant

- A physical plant possesses a number of *properties* which define the state of the plant.
- The properties are quantified (or qualified) by the *sampling* operation.

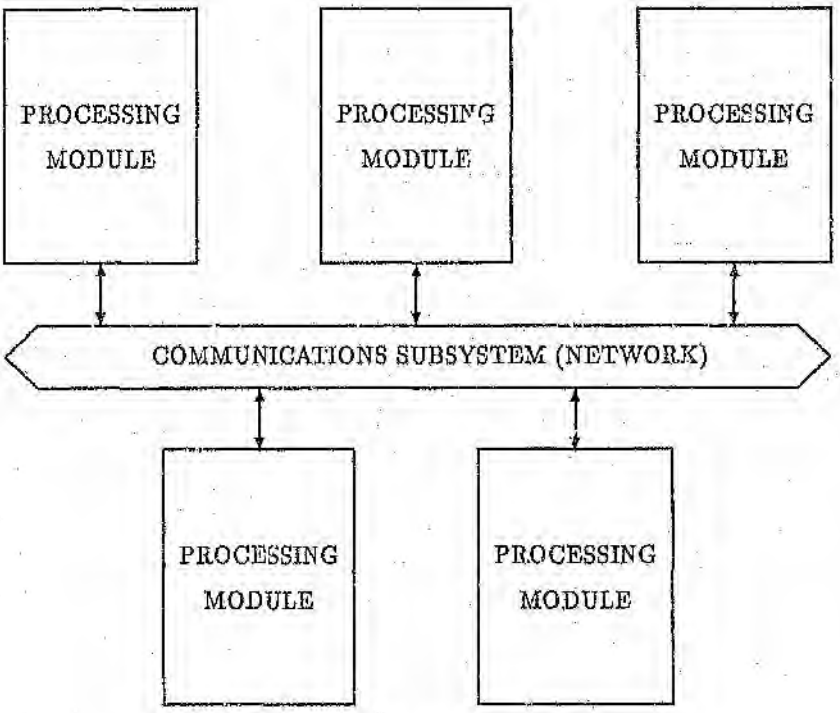


Figure 7.2: Distributed computer control system

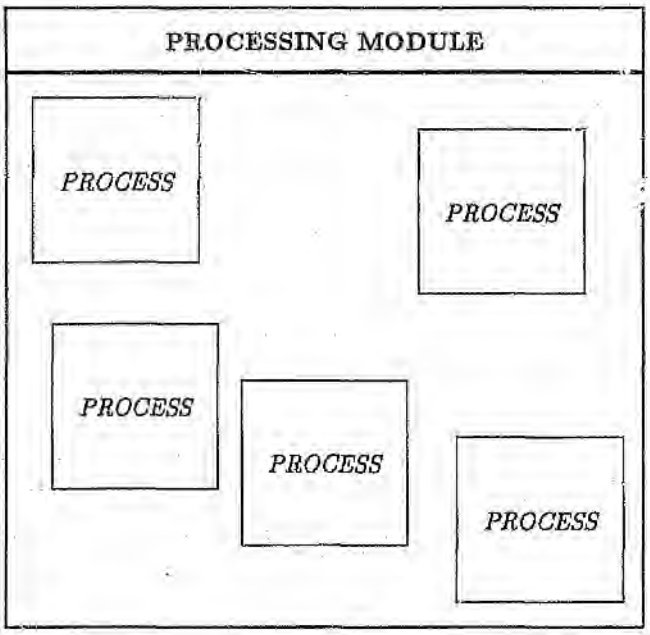


Figure 7.3: Processing module

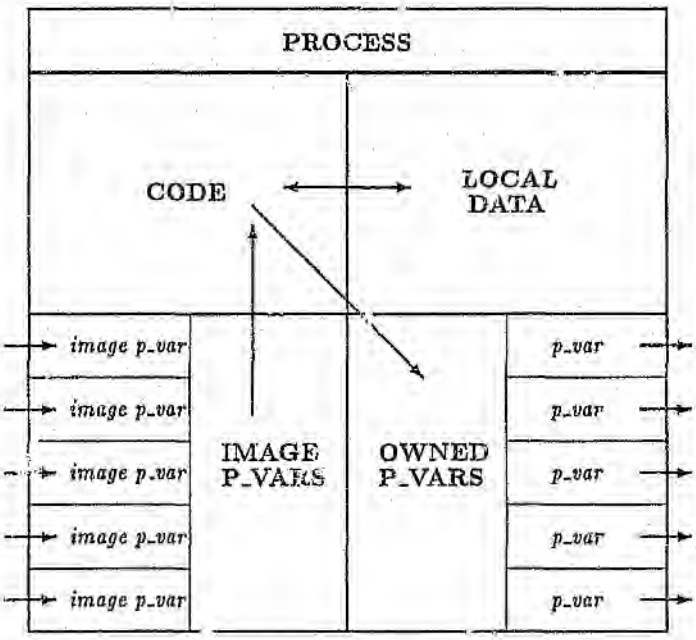


Figure 7.4: Process

P_VAR		
NAME	*SAMPLE VALUE*	*SAMPLE LIFETIME*
Name assigned to the *p_var* which corresponds to a single property in the physical plant.	State value which contains the result of either periodic or event-triggered sampling.	*d*, the time after which the sample value can no longer be assumed to represent the property. The sample lifetime field may be composed of several sub-fields which can be used to specify "window-in-time" time constraints.

Figure 7.5: P_var

- Periodic sampling is usually used to update the state values contained in *p_var* sample value fields. The model does however allow for the use of event-triggered sampling.

7.5 P_vars

- Sample values are expressed as state values, and are inserted into *p_vars*.
- The state values contained in *p_vars* are persistent. They are overwritten by new state-value messages which are transmitted by the sender. (However *image p_vars*, when instantiated, can be designated as either "persistent" or "consumed".)
- Each *p_var* has a unique name and contains the *sample value* as well as a *sample lifetime*.
- The meaning of the sample lifetime depends on whether periodic or event-triggered sampling is used.
- Sample lifetimes are a function of the dynamic behaviour of the plant.

- The *p_var's sample lifetime* field may be broken down into several sub-fields which enable "window-in-time" time constraints to be expressed.
- Each *p_var* is owned by a single process which is called the *owner process*.
- The owner process and the *p_var* reside in the same processing module, that is, the *p_var* is only written to by a local writer.
- A remote process which requires access to a property represented by a particular *p_var* may instantiate a corresponding *image p_var*.
- *Image p_vars* are updated by *p_var maintenance messages* which are sent by the *p_var* owner process.
- Ownership of each *p_var* by a single, local process enforces a unidirectional flow of information from the *p_var* to its *image p_vars*, and consequently the need to support the distributed updating of *p_vars* is eliminated.

Chapter 8

Applying the p_var model

8.1 Introduction

In this chapter it is shown how the *p_var* model can be used to design practical real-time distributed computer control systems, and how it can be used to approach theoretical problems.

Two case studies are presented. The first describes the design and implementation of a set of interprocess communication primitives for use in real-time distributed computer control systems. The systems described have been developed and used over the past three years.

In the second case study we state the problem of ensuring state-conformity in real-time distributed computer control systems. The state-conformity problem is one of the most important problems in the area of distributed control. We show that our model eliminates the state-conformity problem in a system where a single property is sampled by multiple sensors. The problem of ensuring global state-conformity in a system where multiple properties are being sampled is then stated. No attempt is made to solve it; we merely show how our model allows the clear, unambiguous definition of a set of conditions which must be met if global state-conformity is to be ensured.

8.2 An implementation of IPC primitives

A set of interprocess communication primitives—based on Distributed State Variables (DSVs)—was developed from the *p_var* model. The primitives use periodic state value sampling. No provision is made for event-triggered sampling or for state-increment values. Our implementation is suitable for use in systems which sample analog signals. *No effort is made to ensure state-conformity in the reflection* of Boolean-valued properties. It can be seen from chapter 5 that it is not difficult to modify these primitives so that they ensure consistent representation of Boolean valued DSVs. The programmer's reference guide to *DSVLIB*, the DSV implementation, is given in appendix A. Source code of the main header files is brief, and so it has been included in appendix B.

8.2.1 Distributed State Variables and the p_var model

In our implementation, each Distributed State Variable can be regarded as a *p_var*.

- A DSV contains the value of a plant property which has been quantified by the measurement system.
- Each DSV contains a number of fields:
  1. A *DSV name* field
  2. A *DSV value* field
  3. A *DSV lifetime* field
  4. A *status byte* which is used to communicate system information—the existence and meaning of this byte is not known to the applications programmer.
- All DSV values are state values.
- All DSVs are updated using periodic sampling.
- Each DSV is owned by a single process. The owner process and the DSV reside in the same processing module.
- Any process which wishes to access a value in a DSV which is owned by a remote process may instantiate a corresponding *image DSV*.

- Each processing module must periodically broadcast all DSVs which are owned by processes executing in the the processing module, so that any processes which have instantiated corresponding image DSVs can read *image DSV update messages* off the network.
- Each processing module must periodically read *image DSV update messages* off the network which correspond to the image DSVs which have been instantiated by the local processes.
- A primitive instruction is provided which allows a value to be placed in a DSV. The primitive causes the old value-lifetime pair in the DSV to be overwritten and it assigns a *lifetime* to the DSV.
- Another primitive instruction is provided which allows a value to be read from a DSV. Reading the value of the DSV does not cause the DSV value to be consumed. When the value is read, the DSV lifetime is automatically inspected. The expiry time is calculated from the message validity time and the current time as reflected by the processing module's local clock. The value is returned even if it is no longer valid, although invalid values are flagged as invalid. This enables the applications programmer to take whatever action he wishes when an invalid DSV value is encountered.
- Primitives which instantiate DSVs and image DSVs are provided.
- A primitive which causes DSVs and image DSVs to be deleted is provided.

8.2.2 Distributing the DSVs

DSVs can be distributed over the network if each processing module maintains two tables; one containing details of all DSVs which are owned by processes executing in the local processing module (the local DSV table), the other containing details of all image DSVs which are required by processes executing in the processing module (the remote DSV table). Each processing module must periodically broadcast its local-owned DSVs (those listed in the local DSV table) onto the network. Similarly, each processing module must read off the network all *image DSV update messages* which are required to maintain the DSVs listed in the processing module's remote DSV

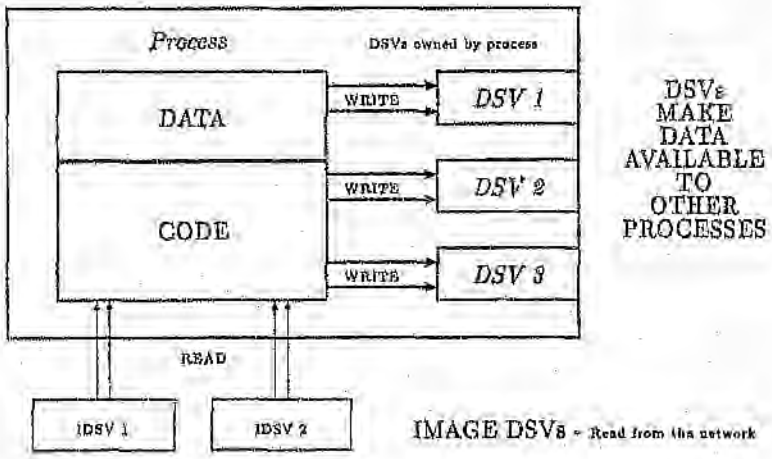


Figure 8.1: The Process - DSV relationship

table. According to this algorithm, a processing module never broadcasts any DSV which does not belong to it, and it only reads those image DSV update messages which are required for the maintenance of the DSVs listed in the processing module's remote DSV table. One advantage of using this algorithm to implement a real-time distributed database is the that it leads to a constant network load. The number of DSVs in the system will not vary greatly, and the transmission of messages is periodic.

8.2.3 Description of systems software

The primitives are implemented as a number of layered C header files which provide primitive calls to the applications programmer. The upper layer, termed the *DSV implementation layer*, defines interprocess communication primitives which are written in terms of generic functions. These generic functions are defined in a lower layer, the *system interface layer*. The system interface layer is operating system specific, and implements the generic system interface calls in terms of local operating system kernel calls and IPC facilities.

Application code can only make calls to the *DSV implementation layer*. Because the *DSV implementation layer* is implemented in terms of standard ANSI C func-

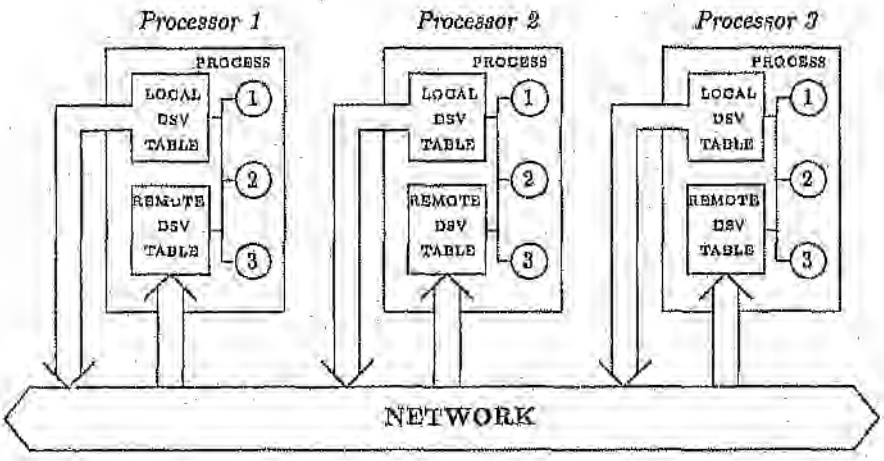


Figure 8.2: Distribution of DSVs

tions and system interface calls, which are provided by the *system interface layer*, the *DSV implementation layer* and all applications code is completely portable. Only the *system interface layer* must be re-written when the primitives are ported to a new operating system. The complexity of the *system interface layer* for a given operating system depends on the suitability of the interprocess communication facilities offered by that operating system to DSV implementation.

8.2.4 The C Language

It was decided to implement the primitives in the C language for a number of reasons. C is widely used by technical applications programmers and systems programmers, and C compilers are available for most operating systems. C is usually one of the first languages available on any new system, because a C compiler can be ported in several hundred man hours. It is a small, flexible language, which gives the programmer access to both high and low level instructions. This flexibility is extremely useful because the programming of control applications requires a mix of low-level hardware interface programming, complex mathematical and algorithmic programming, high-level user-interface and graphics programming, and low-level systems programming [83,84,85,86].

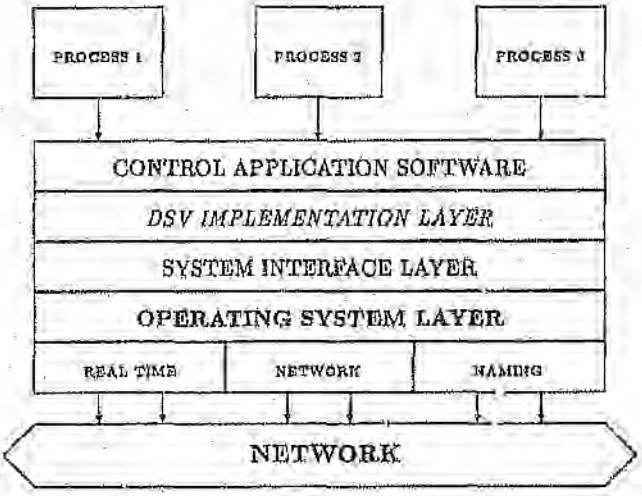


Figure 8.3: Layered structure of DSV software implementation

8.2.5 Separation of Code and Implementation Modules

The C pre-processor has been used to provide a mechanism for configuring inter-process communication without necessitating modification of the code of individual processes.

Each process, when opening a DSV or instantiating an image DSV, is required to specify the name of the DSV. When process modules are designed for re-use, the name of the DSV that will be used in the specific implementation is not known. The pre-processor can be used to define names for all DSVs, and the pre-processor definition can be included as a separate include file in the process module source file. This include file is referred to as the *configuration file*. The relationship between DSVs can be defined at a later stage by editing only the configuration (*.cfg*) file.

For example, a generic process which scales a variable, scalevar, may declare the variable as VAR. Another generic process, scantemp, may be developed to sample the temperature of some plant from a specific type of sensor. This process may refer to the measured value as TEMP. In some application we may wish to scale the temperature of a tank temperature after it has been measured. We could use a DSV called "TANKTEMP" to represent the temperature of the tank. The file scalevar.cfg

would then contain the statement
#define VAR "TANKTEMP"
and the file scantemp.cfg would include the statement
#define TEMP "TANKTEMP".

8.2.8 Operating systems

DSYs have been ported to three different platforms. The first implementation was developed to run under the FlexOs 286 and 386 operating systems. The code was then ported to run in a simple non-premptive multi-tasking scheduler which we developed under MS-DOS. The latest implementation of the code runs under the OS/2 operating system, and makes use of the OS/2 LAN Manager.

FlexOs

Operating system - FlexOS is a Digital Research product aimed at the "flexible automation" market. Although FlexOs claims to be a "real-time" operating system, we were not attracted to it because of its rapid task switching capabilities or low interrupt latency. Our approach in dealing with real-time constraints is aimed at achieving deterministic process behaviour. For this reason, most commercial "real-time" operating systems, with their non-deterministic interrupt-driven schedulers had little to offer us [87,88,89].

What attracted us to FlexOs was its flexibility. The idea behind FlexOS is to allow the systems engineer to configure the operating system kernel according to his own specific requirements. The scheduler and message-passing conventions can be tailored to specific applications. FlexOS has a well-defined pipe system which it uses for interprocess communication. Kernel calls are also provided which facilitate the use of shared memory. The scheduler has an interrupt latency of about 10 microseconds. A networking system called FlexNet has also been advertised, but our department has not been successful in obtaining the software.

FlexOS runs on the Intel 80286/80386, and has been ported to Motorola's 680n0 series of processors. It also emulates MS-DOS, thus preventing the user's PC applications and data from becoming inaccessible. FlexOS is well-designed, and well-suited to control applications, but we stopped using FlexOS because we found software

development difficult because of poor development tools, poor third party support, and the lack of networking software.

DSV implementation - Our implementation of the DSV interprocess communication primitives makes use of the FlexOs named pipe primitive. Each DSV is a named pipe which can hold one message. The pipe is read using a non-destructive read. When the owner process writes to the DSV, it causes the old value-sample lifetime pair to be overwritten. Despite the fact that we could not obtain the FlexNet networking software, we managed to implement a distributed system which consisted of two processing modules which communicated over a full-duplex serial link. The algorithm for distribution of DSVs which is described in this chapter was implemented under FlexOS by Markus [90]. This experimental system was used by Sousa [91] who was investigating the use of re-usable object-oriented software in real-time distributed computer control systems.

Non-preemptive DOS-based scheduler

Operating system - A non-pre-emptive, round-robin scheduler based on the MS-DOS PC operating system was developed in conjunction with Lun [92,93]. This simple scheduler was written in C. It uses a circular linked list to schedule processes. Each process must be cyclical. Each task must explicitly *yield* control to the next process in the linked list [94,95]. (A process usually yields control to the next task each time a single execution cycle of the process has been completed.) One advantage of this non-premptive scheduling mechanism is that the processes behave deterministically. The processor is switched from the control block of one process to the control block of the next process using the C *set_jmp()* and *long_jmp()* instructions. Each process in the system must be written as a C function and compiled into the kernel.

DSV implementation - This scheduler was not developed to provide an alternative to our FlexOs or OS/2 testbeds. A less-sophisticated, more accessible system was needed which would enable development of applications software which made use of DSVs. The ease with which DSVs can be ported led to the development of this DOS-based testbed. Two versions of this scheduler are currently in use in our department at present. One is a version which has been tailored to provide a test bed

for the investigation of real-time distributed artificial intelligence [93]. The other is a version which features DSVs as a standard IPC primitive. This implementation of DSVs is completely centralized. Shared memory is used to implement DSVs. Each DSV is a block of shared memory which can hold a single message plus its validity time. A non-destructive read mechanism is used to read the DSV. When the owner process writes to the DSV, the old value-sample lifetime pair is overwritten.

Zagneov [74] made use of this testbed in an MSc project which was supervised by the author. He designed and began the implementation of a language for programming and configuring real-time distributed control systems. The language consists of standard "building-blocks" which perform standard control functions. They employ DSVs to communicate with one another. Larger, more complex "building-blocks" can be programmed by making use of lower-level and other high-level building blocks. This language eliminates the need for the control engineer to have any knowledge of systems programming. We have also investigated control system design techniques which relate the detail of design to the level of the available "building-blocks". A summary of this research is presented in appendix C.

Zagneov, for the purposes of his work, extended the kernel by incorporating into it a driver for an analog-to-digital / digital-to-analog board. The kernel and primitives were then used to control a physical plant. Zagneov also added a command-line interface for initiating and terminating tasks, as well as simple graphical display primitives to the kernel. At present a MODBUS driver for the kernel is being developed. This will enable the kernel to use a MODBUS-compatible PLC as a plant interface. The possibility of allowing this kernel to be used in a distributed environment with OS/2 LAN Manager (which allows DOS workstations to be connected) is currently under investigation.

OS/2

Operating system - OS/2 is a multitasking operating system which was developed jointly by IBM and Microsoft. It was designed to run on IBM-compatible 80286 or 80386 based computers. The operating system was designed for use in office automation environments. We decided to use OS/2 as a development platform because of the programming support tools that were available, and because of the

flexibility of the OS/2 kernel and the range of features offered by the operating system.

OS/2 provides a multitude of multitasking, scheduling, and interprocess communication features. A number of *sessions* can execute concurrently under OS/2. Each session consists of a number of *processes* which share a common set of I/O facilities. Each process consists of one or more *threads*. Threads are basic units of concurrent execution in OS/2. Each thread is assigned a scheduling priority. The scheduler schedules threads based on these priorities. OS/2 offers three different priority classes; *time-critical*, *regular* and *idle*. Each priority class is divided into 32 priority levels. Several other parameters can be used to "tweak" the scheduler. The system configuration file can specify a maximum number of threads that may execute simultaneously. The *timeslice* parameter, which specifies the CPU time which is to be allocated to each thread is set in the configuration file. The *maxwait* parameter, which specifies the maximum time that any thread will wait before receiving a slice of CPU time, can also be set in the configuration file [96,97].

OS/2 provides a number of interprocess communication mechanisms including shared memory, named and unnamed pipes, semaphores, signals, dynamic data exchange (DDE) and queues [98]. The OS/2 LAN Manager enables files to be transferred and accessed over the network and it allows data to be communicated between processes executing on different processing modules. LAN Manager allows named pipes and another primitive, *mailslots*, to be used to communicate between processing modules.

Mailslots facilitate one-to-many communications on a node basis. This means that one node on the network can write to corresponding mailslots on any node which has an instance of the same mailslot. Within each node, only the process which created the mailslot has access to it. Therefore mailslots do not facilitate one-to-many communication on a *process* basis. Two mailslot services are offered; first class, which ensures message delivery; and second class, which does not ensure delivery but incurs less overhead and is faster. Named pipes provide a one-to-one communication service on a *process* basis. The service is network-transparent. A process may create a pipe, but a connection cannot be established until the second process (which may be executing in the same processing module or in another processing module) has

opened its end of the pipe. This explicit process-to-process connection must be established before any communication can occur.

DSV implementation - Our OS/2 implementation of the DSV primitives constructs a table of locally-owned DSVs and a table of DSVs which are required from remote processing modules in shared memory. Access to this shared memory is controlled by semaphores. The option of using the named pipe IPC primitive, which provides network transparency, was examined, but because of the explicit connection requirements and one-to-one communication limitations, it was deemed to be unsuitable. Mailslots inherently support multicast communication, and hence they are more suitable for implementation of DSVs. Although the process which instantiates a mailslot is the only process in the processing module which has access to it, mailslots can be used to distribute DSVs to other processing modules. Each processing module can execute a specific process which instantiates all mailslots required by the processes executing on that module. That process then unpacks incoming DSVs from the mailslots into the shared-memory tables, and packs outgoing DSVs from the shared-memory tables into the mailslots. This system can be used to implement the DSV distribution algorithm described above. As with our other implementations, each shared memory block holds one DSV value-sample lifetime pair. DSVs are read with a non-destructive read operation. When the writer process writes to the DSV, the old contents are overwritten. At present our OS/2 DSV implementation has no way of obtaining data from a physical plant. We are in the process of writing a MODBUS driver which will enable a PLC to be used for data acquisition and plant interfacing.

8.2.7 Description of hardware configuration

The primitives were implemented on three different platforms:

1. IBM-compatible personal computer running the FlexOs 386 protected-mode multitasking system operating system. The PC was equipped with:
   - Intel 80386 CPU
   - Intel 80387 maths co-processor
   - 2.5 Mb of RAM

- 40 Mb hard disk with 28 ms average access time
- EGA display adaptor and monitor
- 101-key keyboard
- 1.2 Mb $5\frac{1}{4}$ inch floppy disk drive
- Serial port
- Parallel port

2. IBM-compatible personal computer running our own non-pre-emptive DOS-based scheduler. The PC was equipped with:
   - Intel 80286 CPU
   - Intel 80287 maths co-processor
   - 1 Mb of RAM
   - 40 Mb hard disk with 28 ms average access time
   - VGA display adaptor and monitor
   - 101-key keyboard
   - 1.2 Mb $5\frac{1}{4}$ inch floppy disk drive
   - Serial port
   - Parallel port

3. A network consisting of two IBM-compatible personal computers running the OS/2 v1.21 operating system and the OS/2 LAN Manager. Each PC was equipped with:
   - Intel 80286 CPU
   - Intel 80287 maths co-processor
   - 8 Mb of RAM
   - 2 × 40 Mb hard disk with 28 ms average access time
   - EGA display adaptor and monitor
   - 101-key keyboard
   - Mouse

- 1.2 Mb $5\frac{1}{4}$ inch floppy disk drive
- 1.44 Mb $3\frac{1}{2}$ inch floppy disk drive
- LAN adaptor card
- Serial port
- Parallel port

8.2.8 DSV Primitives

Several primitives are provided for creating and communicating using DSVs.

dsv_wopen()

Function prototype: DSV dsv_wopen (char *name)
This function is called by a process wishing to create a DSV. The calling process is given ownership of the DSV and rights to write to the DSV.

dsv_ropen()

Function prototype: DSV dsv_ropen (char *name)
This function is called by a process wishing to instantiate an image DSV. The calling process is given read-only access to the DSV. If the DSV does not exist in the system, an error message is returned and the process will continue trying to instantiate the image DSV.

put_state()

Function prototype: void put_state (DSV dsv_var, void *msg, LONG valid)
This function is used by the owner process to write data to the DSV. The data is automatically time-stamped by the function.

get_state()

Function prototype: MSGVALID get_state (DSV dsv_var, void *msg)
This function is used by subscriber processes to read a value from a DSV. The function automatically inspects the lifetime associated with the DSV value, and returns an error if the DSV value is invalid. Action taken when an invalid DSV is encountered is application specific, and is therefore left to the programmer to define.

dsv_close()

Function prototype: void dsv_close (DSV dsv_var)

This function is called by a process wishing to close a DSV or image DSV.

8.3 State-conformity in sensor-based DCCSs

The problem of ensuring state-conformity in distributed real time systems remains unsolved. There are three state-conformity problems which arise in distributed real-time systems:

1. It is difficult to prevent different processing modules in a distributed system from having conflicting images of a Boolean-valued property.

2. It is difficult to ensure that all processes have the same image of a Boolean-valued property when different processes are sampling the property asynchronously. MacLeod examines this problem in detail in [4] and [8].

3. It is difficult to ensure that the global property representation in the computer system (i.e. the set of all plant properties) conforms to a possible plant state. That is, the computer system should never reflect a plant state which does not occur physically.

8.3.1 State-conformity in the distributed database

The difficulty of ensuring that no two processing modules have conflicting views of some plant property was discussed at length in chapter 5. It was shown that if, when a state change is detected, the sender process delays transmission of the message for a time t_{hat} which must be sufficient to allow all image *p_vars* to become invalid, then no two processing modules will have conflicting images of a *p_var*. Although the possibility of solving the problem in this way has been mentioned by MacLeod [4], the solution proposed in this thesis is unique because the clocks of the processing modules are not required to be synchronized.

8.3.2 Single property state-conformity

Consider two processes, P_1 and P_2, which sample a Boolean valued property, B (fig. 8.4). P_1 samples at time t_2, which is just before the property change occurs,

and again at t_4. P_2 samples at time t_3, which is just after the property change occurs, and again at t_5. During the interval (t_3, t_4) the two processes will have a conflicting image of the property B.

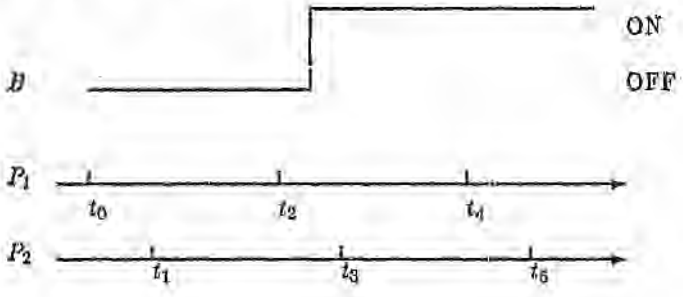


Figure 8.4: Two processes sampling a single property.

The model of information ownership that was developed in chapter 6 states that each *p_var* is owned and written to by a single, local process. According to our model, there should never be more than one source for the value of a single property. *Image p_vars* are updated by *p_var* mainenance messages. The *p_var* is the single source of information for all *image p_vars* . This problem then reduces to the problem of ensuring that all processing modules have the same image of the *p_var*, which is described above and in chapter 5. Where there is a need for redundant sensors, the measurements of the property are distinct and are quantified in distinct *p_vars*. An "effective *p_var*", which represents the plant property, is defined by a arbitration process which reads the *image p_vars* of the two redundant *p_vars*. Again the problem reduces to the problem of ensuring that all processing modules have the same image of a single *p_var* (fig 8.5).

8.3.3 Global state-conformity

The problem of ensuring that the computer system's image of the plant state conforms to the plant's physical state is non-trivial. Consider a process, P_1, which samples a Boolean-valued property, B_1. Process P_2, samples a Boolean-valued property, B_2. The two sampling processes operate asynchronously. P_1 samples B_1 at times t_1, t_3 and t_5, and P_2 samples B_2 at times t_2, t_4 and t_6. B_1 and B_2 both change from "OFF" to "ON" sometime between t_3 and t_4. B_1 however, changes before B_2. In

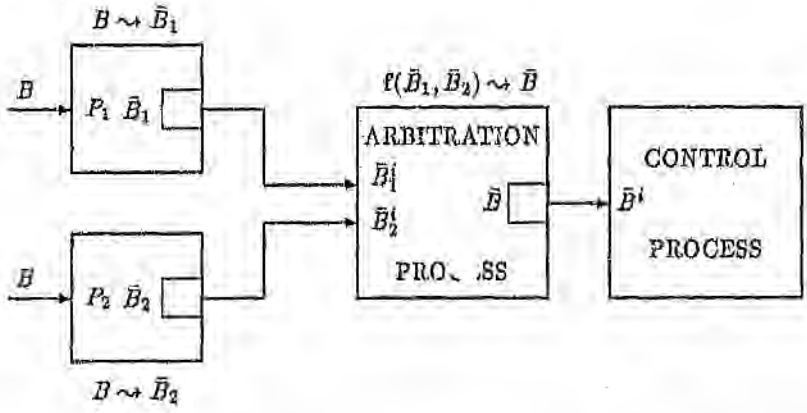


Figure 8.5: State-conformity with two processes sampling two different properties.

the interval (t_4, t_5), the image in the computer system will reflect that B_1 is "OFF" and B_2 is "ON", when in fact B_2 is "OFF" and B_1 is "ON". It is possible that the plant state which is reflected in the computer system may never arise in the plant itself (fig. 8.6). The problem of global state-conformity has not been solved for the case of asynchronous sampling of multiple properties with different sampling rates. The conditions which must be met if this problem is to be solved have also not been clearly and concisely described.

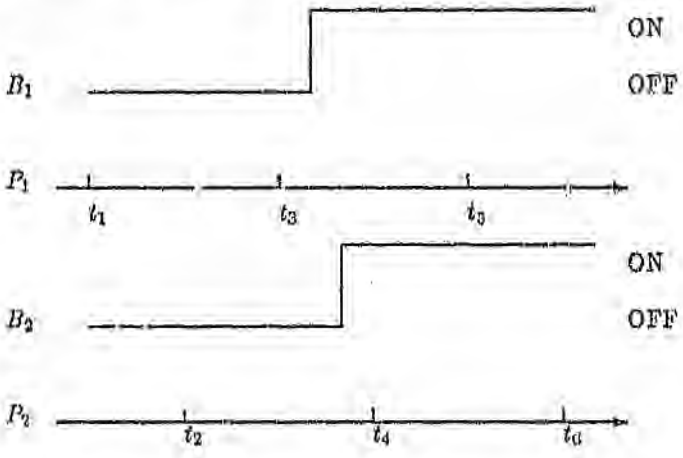


Figure 8.6: Two processes sampling two different properties.

8.4 Conditions for ensuring state-conformity

The *p_var* model enables us to specify a set of conditions which, when met, ensure (i) single property state-conformity, and (ii) global state conformity in sensor-based real-time DCCSs. (No attempt is made to suggest how condition 4, which describes global state-conformity, can be met.)

1. Each *p_var* must be owned by a single, local process. This enforces a unidirectional flow of information from each *p_var* to its *image p_vars*, because there is a single source for each property value.

2. The operation of reading an *image p_var* which has been allowed to become invalid because of the imminent assertion of a property change should return a non-determinate (*ND* state). (Real-time fault-tolerance and reliability constraints which limit the time for which an *ND* state may persist can also be specified—see chapter 5.)

3. *Single property state-conformity* - Whenever a valid *p_var* value is accessed, the system must ensure that *all* corresponding *image p_vars* contain the same valid sample value.

4. *Global state-conformity* - When a valid *p_var* value, reflecting the state of a property at time t', is used at a time t, a complete set of valid *p_vars* which correctly reflects the entire plant state at time t' must be present. (t' precedes t.)

   (This is only a state-conformity constraint. A further real-time constraint could specify the relationship between t and t'. In practice, this constraint means that it must be *possible* to sample all properties and represent them in the control system before any one of them changes state. The constraint is inherently met by an asynchronous system where all properties are being sampled with the same sampling period—as long as the sampling period satisfies the conditions stipulated in chapter 5. It is also inherent in a synchronous system, even if properties are sampled at different rates, as long as the sampling periods are multiples of one another and synchronized with one another.)

8.5 Conclusion

In this chapter, it has been shown how the *p_var* model provides a framework for solving the theoretical problems of real-time distributed computing. It was also shown how the model can be used in the implementation of systems and software.

The first part of the chapter described a set of interprocess communication primitives which have been implemented on three different platforms. Because these primitives were developed using our theory of process interaction, they inherently provide real-time functionality.

In the second part of the chapter our theory of interprocess communication was applied to the problem of state-conformity in distributed systems. This study shows that without an adequate model of process interaction in real-time distributed systems, these theoretical problems cannot even be correctly identified. The application of our model and theory resulted in the elimination of one of the issues in the state-conformity problem. It also facilitated the specification of a set of conditions which, when met, ensure single property and global state conformity in real-time DCCSs.

Chapter 9

Conclusion

9.1 General

Recently there has been a trend towards the use of distributed real-time computer systems which potentially offer greater functional flexibility, better maintainability and better reliability than centralized systems. Little exists in the way of theory which models the behaviour of these systems. Current design considerations are based largely on practical experience and heuristics which cannot be freely applied to all systems. With increasing demands being placed on real-time control systems, the need for a theory for the distributed control of real-time dynamic systems is becoming more apparent.

In this thesis we have tried to develop such a theory by analyzing process interaction in real-time distributed computer control systems. The subject was approached by examining interprocess communication, since process interaction in distributed systems is characterized by interprocess communication. The central issues affecting process interaction were then analyzed at a fundamental level, and the *p_var* model, which unifies the concepts that emerge from this analysis, was proposed.

9.2 Research contributions

The *p_var* model can be used to model process interaction in real-time distributed computer control systems. It thus forms the basis of our theory of real-time dynamic systems and their control.

This research contributes to an enhanced understanding of process interaction in computer controlled systems. Some of these contributions have been incorporated into the model, whilst others constitute more general, conceptual studies. Research contributions include:

9.2.1 IPC primitives and control

We have examined a number of interprocess communication proposals which have been put forward in the literature. They have been analyzed in the context of computer control systems and compared with one another. This research updates earlier work performed by MacLeod [4], and helps to clarify the interprocess communication requirements of distributed control systems.

9.2.2 Sampling and data representation

Sampling methods and data representation techniques used in computer control systems are rigorously examined. The sampling methods and data representation techniques are defined and compared to one another. The use of various sampling techniques with different data representation techniques is examined and the advantages and disadvantages of each combination is discussed. This study has resulted in the precise definition and documentation of a body of essential theory which until now was not defined, although it was applied on an ad-hoc basis.

We also conclude that all sampling is inherently periodic when viewed at the most fundamental level of plant-control system interaction, and that event-triggered sampling is an artificial mechanism which is used to meet fast sampling requirements.

9.2.3 System boundaries

We maintain that plant-control system communication must be regarded to always be control-system (receiver) initiated. This conclusion follows because front-end data acquisition devices are actually components of the control system and not of the physical plant. Although in practical implementations the plant-control system boundary is often drawn so as to include the front-end data acquisition system in the plant component, we claim that this is only for convenience of implementation,

9.2.4 Real-time systems

Our study of real-time systems has enabled us to develop a definition of the term real-time which does not rely on any notion of speed. Our definition defines real-time performance to be dependent on temporal determinism.

A nine-point list of guidelines and techniques which can be used to enforce correct real-time behaviour in real-time distributed computer control systems is proposed.

A method for selecting sample lifetimes based on the dynamic characteristics of the plant and the transmission delay characteristics of the network is presented. Although the method is based on heuristics, it allows a quantitative link to be established between the digital control domain, and the domain of distributed real-time computing; two areas which are seldom treated together.

A reliability and fault-tolerance enhancement technique is proposed which makes the system resistant to transient communication faults which result in lost messages. The technique is based on "sample overlapping". An equation is derived which yields the required sampling interval, given the desired sample lifetime, sample overlap factor, and network transmission characteristics.

A technique for ensuring state-conformity in the representation of Boolean-valued properties within a distributed computer system is proposed. An important feature of this technique is that the distributed clocks in the system do not require synchronization.

9.2.5 Flow-control

Mechanisms for buffering incoming messages and for controlling the flow of information between a sender and a receiver process have been widely applied in communication systems and generalized computer systems. For the first time, the effect of applying these mechanisms in computer control systems is investigated. This study again constitutes a body of essential theory which has not previously been accessible.

The findings of this study show clearly that it is not possible for the control system to apply flow-control to the plant, and that apparent flow-control occurs when the control system applies flow-control to the front-end data acquisition system.

9.2.6 Ownership of information

We conclude that each property variable (*p_var*) is owned by a single process in the distributed system. This process is called the owner process.

We maintain that in a distributed computer control system, it is alwa[illegible]ble to ensure that the *p_var* and its owner process reside in the same process[illegible].

It is shown that the existence of a single local writer for each *p_var* eliminates the need to support the mutual exclusion of multiple writers over the network. It is also shown that this result can be viewed as the converse of Lamport's solution to the mutual exclusion problem in interprocess communication in distributed systems.

9.3 Significance of the work

The *p_var* model is significant because it allows problems in distributed control to be viewed from a purely theoretical perspective. Many of the techniques we describe have been applied, but less formally and on an ad-hoc basis. Although our model is theoretical, it is directly applicable to real-world distributed control problems.

Our model describing process interaction can be used to develop interprocess communication primitives, distributed operating systems, programming languages, system specification tools and design methods, all specifically optimized for use in computer control applications. The model also provides a framework in which some of the theoretical problems in distributed real-time computer control may be solved.

The *p_var* model ties together a number of independent bodies of theory and examines their relevance and correctness in [illegible]ontext of real-time distributed computer control. It was found that the application of much of this general theory to computer control had not been thoroughly investigated and documented. Consequently, we have developed theories of sampling, data representation and flow-control in computer control applications.

9.4 Limitations and caveats

The unique characteristics of discrete properties were highlighted in chapter 5. Although the *p var model* is capable of representing discrete properties, it is limited

in that it does not explicitly recognize the fact that a discrete property may only take on a finite number of state values.

The model has also not been extended to cater for redundancy management. Although we have explicitly managed redundant subsystems through careful naming of processes and *p_vars*, it is felt that a model of process interaction in distributed computer control systems should take into account the special relationship of redundant subsystems to one another.

We have not attempted to solve any of the theoretical problems of distributed control, some of which have been begging solution for many years. An example of such a problem is the global state-conformity problem which was described in the case study. (The single property state-conformity problem is [illegible]d automatically by our treatment of Boolean-valued properties.)

9.5 Recommendations for future work

The most natural extension of this work would be to provide a full treatment of discrete variables. Another useful extension of this work would be to extend the model to describe the management of redundancy. In section 9.4 it was mentioned that redundant subsystems possess a special relationship to one another. We anticipate that it will be possible to express this relationship by the way of process and *p_var* naming conventions.

This work draws on many concepts from the domains of computer science and communication engineering. A great deal is still to be done to incorporate digital control concepts into the theory.

The most challenging problem which has not yet been solved is that of ensuring state-conformity in sensor-based DCCSs. The set of conditions given in section 8.4 captures the essence of the problem and should assist with its solution.

9.6 Summary

The *p_var* model is proposed as a basis for a theory of real-time dynamic systems and their control. The model describes the interaction of multiple real-time processes which typically reside on distinct processing modules (although they do not have

to). These processes interact only by exchanging messages. The model unifies the principles underlying real-time distributed computer control. This enables systems to be examined from a perspective which has a solid theoretical foundation.

Bibliography

[1] Farber, G. (1978), Principles and applications of decentralised process control computer systems, *IFAC World Congress, Helsinki*, Vol. 1, pp 385-292

[2] Kramer, J., Magee, J. and Sloman, M. (1984), A Software Architecture for Distributed Computer Control Systems, *Automatica*, Vol. 20 No. 1, pp 93-102

[3] MacLeod, I. M. and Rodd, M. G. (1983), Interprocess Communication Primitives for Distributed Process Control, *IFAC Symposium on DCCS, Sabi-Sabi, South Africa*

[4] MacLeod, I. M. (1983), A Study of Issues Relating to Real-Time in Distributed Computer Control Systems, *PhD. Thesis, University of the Witwatersrand*

[5] Liskov, B. (1982), On Linguistic Support for Distributed Programs, *IEEE Transactions on Software Engineering*, Vol. SE-8 No. 3, pp 203

[6] Kopetz, H. et al (1983), A Message-Based DCCS, *IFAC Conference on DCCS, Sabi-Sabi, South Africa*

[7] Kopetz, H. et al (1989), MARS: A Maintainable Real-Time Distributed Computer Control System, *IEEE Micro*, , pp 25-40

[8] MacLeod, I. M. (1983), Data Consistency in Sensor-Based Distributed Computer Control Systems, *Proceedings, IFAC Distributed Computer Control Systems*, , pp 87-95

[9] Brinch-Hansen, P. (1973), Concurrent Programming Concepts, *Computing Surveys*, Vol. 5 No. 4, pp 223-245

[10] Dijkstra, E. W. (1965), Solution of a Problem in Concurrent Programming Control, *Communications of the ACM*, Vol. 8 No. 9, pp 569

[11] Dijkstra, E. W. (1968), *Co-operating Sequential Processes*, Academic Press, pp 43-112

[12] Hoare, C. A. R. (1974), Monitors: An Operating System Structuring Concept, *Communications of the ACM*, Vol. 17 No. 10, pp 549-557

[13] Lamport, L. (1986), The Mutual Exclusion Problem: Part 1 - A Theory of Interprocess Communication, *Journal of the ACM*, Vol. 33 No. 2, pp 313-326

[14] Lamport, L. (1986), On interprocess communication—Part I: Basic formalism, *Distributed Computing*, Vol. 1 No. 2, pp 77-85

[15] Lamport, L. (1974), A New Solution of Dijkstra's Concurrent Programming Problem, *Communications of the ACM*, Vol. 17 No. 8, pp 453-455

[16] Warwick, K. (1990), Adaptive Control: an insight, *IEE Computing and Control Engineering Journal*, Vol. 1 No. 1, pp 35-39

[17] DeMarco, T (1978), *Structured Analysis and System Specification*, Yourdon Inc., pp 3-15

[18] Bulman, D.M. (1989), An Object-Based Development Model, *Computer Language*, Vol. 6 No. 8, pp 49-59

[19] Hatley, D.J. and Pirbhai, I.A. (1987), *Strategies for Real-Time System Specification*, Dorset House Publishing Co.

[20] Cox, B. J. (1984), Message/Object Programming: An Evolutionary Change in Programming Technology, *IEEE Software*, pp 50-61

[21] Goldberg, A. (1981), Introducing the Smalltalk-80 System, *Byte*, Vol. 6 No. 8, pp 14

[22] Gomaa, H (1984), A Software Design Method for Real-Time Systems, *Communications of the ACM*, Vol. 27 No. 9, pp 938-949

[23] Yonezawa, A. and Tokoro, M. (1989), *Object-Oriented Concurrent Programming*, The MIT Press

[24] Stankovic, J. A. (1988), Misconceptions about real-time computing: a serious problem for next generation systems, *IEEE Computer*, pp 10-19

[25] Wellings, A. (1991), Real-time software, *IEE Software Engineering Journal*, Vol. 6 No. 3, pp 66-67

[26] Le Lann, G. (1979), An Analysis of Different Approaches to Distributed Computing, *Proc. 1st International Conference on Distributed Computing Systems*, pp 222-232

[27] Kopetz, H. (1983), Real-Time in Distributed Real-Time Systems, *Proceedings IFAC Distributed Computer Control Systems, Sabi-Sabi, South Africa*, pp 11-20

[28] Åström, K. J. and Wittenmark, B. (1984), *Computer Controlled Systems, First Edition* Prentice-Hall

[29] Finkelstein, L. (1987), Errors and Uncertainty in Measurements and Instruments, In *Systems and Control Encyclopaedia, Ed. M. G. Singh*, Vol. 2, pp 1553-1559

[30] Ramamritham, K. and Stankovic, J. A. (1984), Dynamic Task Scheduling in Hard Real-Time Distributed Systems, *IEEE Software*, pp 65-75

[31] Tannenbaum, A. S. and Van Renesse, R. (1985), Distributed Operating Systems, *Computing Surveys*, Vol. 17 No. 4, pp 419-466

[32] Cheriton, D. R. (1988), The V Distributed System, *Communications of the ACM*, Vol. 31 No. 3, pp 314-333

[33] Stankovic, J. A. (1984), A Perspective on Distributed Computer Systems, *IEEE Transactions on Computers*, Vol. C-33 No. 12, pp 1102-1115

[34] Brownbridge, D. R., Marshall, L. F. and Randell, B. (1982), The Newcastle Connection, *Software Practice and Experience*, Vol. 12, pp 1147-1162

[35] Peterson, B. and Silberschatz, A. (1985), *Operating System Concepts*, Addison-Wesley

[36] Gee, K. C. E. (1983), *Introduction to Local Area Computer Networks*, Macmillan Education Ltd

[37] MacLeod, I. M. (1989), Reliability and availability enhancement through the use of $1/n$ active redundancy, *Internal report, University of the Witwatersrand*

[38] Day, J. and Zimmermann, H. (1983), The OSI Reference Model, *Proceedings of the IEEE*, Vol. 71 No. 12, pp 1334-1340

[39] Kirk, M. (1991), Time-critical communication systems, *Computing and Control Engineering Journal*, Vol. 2 No. 1, pp 35-42

[40] Le Lann, G. (1977), Distributed Systems - Towards a Formal Approach, *1977 IFIP Congress Proceedings*, pp 155-160

[41] Lamport, L. (1978), Time, Clocks, and the Ordering of Events in Distributed Systems, *Communications of the ACM*, Vol. 21 No. 7, pp 558-565

[42] Dehning, R. W. and Heher, A. D. (1989), CYGNUS - Evolution of a Supervisory Software Package for Process Control, *RUGSA/SAIMC/SAIEE Symposium on Real-Time Networking, Computing and Control*

[43] Rodd, M. and Deravi, F. (1987), *Communication Systems for Factory Automation*, University College of Swansea

[44] Brinch-Hansen, P. (1978), Distributed Processes: A Concurrent Programming Concept, *Communications of the ACM*, Vol. 21 No. 11, pp 934-941

[45] MacLeod, I. M. (1984), Using Real-Time to Achieve Coordination in Distributed Computer Control Systems, *Proceedings of the IFAC World Congress, Budapest*

[46] Ben-Ari, M. (1982), *Principles of Concurrent Programming*, Prentice-Hall

[47] Burnes, J. G. P. (1980), An Introduction to Real-Time Programming, *Report: SPL International*

[48] Natarajan, N. (1985), Communication and Synchronization Primitives for Distributed Programs, *IEEE Transactions on Software Engineering*, Vol. SE-11 No. 4, pp 396-416

[49] Dijkstra, E. W. (1968), Co-operating Sequential Processes, *Programming Laguages, Ed. F. Genuys*, , pp 43-112

[50] Brinch-Hansen, P. (1977), *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, N.J.

[51] Hoare, C. A. R. (1978), Communicating Sequential Processes, *Communications of the ACM*, Vol. 21 No. 8, pp 666-677

[52] Pyle, I. C. (1981), *The ADA Programming Language*, Prentice-Hall International, London

[53] Fitzgerald, R. and Rashid, F. R. (1986), The Intergration of Virtual Memory Management and Interprocess Communication in Accent, *ACM Transactions on Computer Systems*, Vol. 4 No. 2, pp 147-177

[54] Feldman, J. A. (1979), High Level Programming for Distributed Computing, *Communications of t... ACM*, Vol. 22 No. 6, pp 353-367

[55] Simpson, H. R. and Jackson, K. (1979), Process Synchronization in MASCOT, *The Computer Journal*, Vol. 22 No. 4, pp 332-345

[56] Lamport, L. (1979), A New Approach to Proving the Correctness of Multiprocess Programs, *ACM Transactions on Programming Languages and Systems*, Vol. 1 No. 1, pp 84-97

[57] Lamport, L. (1977), Proving the Correctness of Multiprocess Programs, *IEEE Transactions on Software Engineering*, Vol. SE-3 No. 2, pp 125-143

[58] Lamport, L. (1986), The Mutual Exclusion Problem: Part 2 - Statement and Solutions, *Journal of the ACM*, Vol. 33 No. 2, pp 327-348

[59] Lamport, L. (1986), On interprocess communication—Part II: Algorithms, *Distributed Computing*, Vol. 1 No. 2, pp 86-101

[60] Anger, F. D. (1989), On Lamport's Interprocessor Communication Model, *ACM Transactions on Programming Languages and Systems*, Vol. 11 No. 3, pp 405-419

[61] Slepian, D. (1976), On Bandwidth, *Proceedings of the IEEE*, Vol. 64, No. 3, pp 292-300

[62] Middleton, R. H. and Goodwin, G. C. (1986), Improved Finite Word Length Characteristics in Digital Control Using Delta Operators, *IEEE Transactions on Automatic Control*, Vol. AC-31, No. 11, pp 1015-1021

[63] Cohen, J. E. and MacLeod, I. M. (1991), Distribution of Real-Time Information in Computer Control Systems Systems, *Submitted to the Trnsactions of the SAIE*

[64] Gamnow, G. (1958), The Principle of Uncertainty, *Scientific American*, Vol. 231 No. 1, pp 51-57

[65] Beiser, A. (1983), *Concepts of Modern Physics, Third Edition*, McGraw-Hill

[66] Hinden, H. J. and Rausch-Hinden, W. B. (1983), Special Series on Real-Time Integration, *Electronics Design*

[67] Spivey, J. M. (1990), Specifying a Real-time Kernel, *IEEE Software*, pp 21-28

[68] Diaz, M. and Vissers, C. (1989), SEDOS: Designing Open Distributed Systems, *IEEE Software*, pp 24-33

[69] Kopetz, H. *et al.* (1991) The design of real-time systems: from specification to implementation and verification, *IEE Software Engineering Journal*, Vol. 6 No. 3, pp 72-82

[70] Ostroff, J. S. (1989) Temporal Logic for Real-time Systems, Research Studies Press, John Wiley and Sons inc.

[71] Faustini, A. A. and Lewis, E. B. (1986), Towards a Real-Time Data Flow Language, *IEEE Software*, pp 29-35

[72] Luqi, Berzins, V. and Yeh, R. T. (1988), A Prototyping Language for Real-Time Software, *IEEE Transactions on Software Engineering*, Vol. 14 No. 10, pp 1409-1423

[73] Berzins, V. and Luqi (1990), An Introduction to the Specification Language Spec, *IEEE Software*, pp 74-84

[74] Zagneov, S. (1990), The development of a modular real-time process control language, MSc thesis, *University of the Witwatersrand*

[75] Burns, A. (1991) Scheduling hard real-time systems: a review, *IEE / ,oware Engineering Journal*, Vol. 6 No. 3, pp 116-128

[76] Parnas, D. L. (1972), On the Criteria To Be Used in Decomposing Systems into Modules, *Communications of the ACM*, Vol. 15 No. 12, pp 1053-1058

[77] Jefferson, D. R. (1985), Virtual Time, *ACM Transactions on Programming Languages and Systems*, Vol. 7, N. 3, pp 404-425

[78] Murzalo, K. and Owicki, S. (1983), Maintaining the Time in a Distributed System, *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*

[79] Kopetz, H. and Ochsenreiter, W. (1987), Clock Synchronization in Distributed Real-Time Systems, *IEEE Transactions on Computers*, Vol. C-36 No. 8, pp 933-940

[80] Kopetz, H. and Ochsenreiter, W. (1987), Interval Measurements in Distributed Real-Time Systems, *Proceedings IEEE Workshop on Real-Time Systems*, pp 292-298

[81] Lamport, L. and Melliar-Smith, P. M. (1985), Synchronizing Clocks in the Presence of Faults, *Journal of the ACM*, Vol. 32 No. 1, pp 52-78

[82] Kopetz, H. (1986), Time Rigid Scheduling in Distributed Real-Time Systems, *Proceedings, IFAC Distributed Computer Control Systems*, pp 173-179

[83] Kernighan, B.W. and Ritchie, D.M. (1988), The State of C, *Byte*, Vol. 13 No. 8, pp 205-210

[84] Kernighan, B. W. and Ritchie, D. M. (1978), *The C Programming Language*, Prentice-Hall

[85] Pohl, I. and Edelson, D. (1988), A to Z: C Language Shortcomings, *Computer Language*, Vol. 13 No. 2, pp 51-64

[86] Harbison, S. P. and Steele, G. L. (1984), *A C Reference Manual*, Prentice-Hall, Englewood Cliffs, N.J.

[87] Adams, M. (1988), FlexOS - A DOS Compatible Real-Time OS, *.EXE Magazine*, pp 65-69

[88] FlexOS Programmer's Guide (1986), *FlexOS 386 Operating System Documentation*, Digital Research Corp.

[89] Lampman, R. and Franklin, J. (1989), New operating systems, standards ease control system design and use, *Chilton's I&CS*, pp 41-47

[90] Markus, J. S. (1990), MSc thesis under examination, *University of the Witwatersrand*

[91] Sousa, C. S. M. (1990), MSc thesis under examination, *University of the Witwatersrand*

[92] Lun, V. and MacLeod, I.M. (1990), A simple system executive for distributed real-time computing research, *Transactions of the SAIEE*, Vol. 82 No. 1, pp 52-68

[93] Lux, V. (1990), A framework for distributed real-time intelligence, *PhD Thesis, University of the Witwatersrand*

[94] Heath, W. S. (1984), A System Executive for Real-Time Microcomputer Programs, *IEEE Micro*, pp 20-32

[95] Heath, W. S. (1987), Software Design for Real-Time Multiprocessor Systems, *IEEE Micro*, pp 71-80

[96] Letwin, G. (1988), *Inside OS/2*, Microsoft Press

[97] Iacobucci, E. (1988), *OS/2 Programmer's Guide*, Osborne McGraw-Hill

[98] Duncan, R. (1989), Interprocess Communication in OS/2, *Dr Dobb's Journal*, Vol. 14 Issue 6, pp 14

Appendix A

DSVLIB: Reference Manual

DSVLIB

Distributed State Variables C Programming Library

Version 1.5
Jack E. Cohen
Department of Electrical Engineering
University of the Witwatersrand
Johannesb rg, South Africa

September 1, 1991

1 Introduction

This document explains the structure of the libraries used to implement the Distributed State Variable (DSV) primitives. It defines the inter-relationship between the various libraries, and explains how the software can be ported to other operating systems and hardware platforms. The terminology used is explained, and the various software conventions are documented. It is hoped that the software contained in these libraries will provide an effective applications program interface (API) for real-time distributed systems development. This document is also meant to serve as a guide to programmers wishing to extend the functionality of the library, or who would like to use the existing libraries as the underlying structure for their own libraries.

2 Terminology

- DSV - A Distributed State Variable is the basic interprocess communication mechanism which supports real-time communication between distributed processes.
- API - The Applications Program Interface is the set of software utilities which make DSVLIB features accessible to the applications programmer as function calls.
- PCAO - Process Control Application Objects are functional units of code which perform process control and monitoring functions. They use DSVs for reliable real-time interprocess communication.
- EVENT - An event is an significant occurence at a point in time. An event message is the communication of this occurence.
- STATE - A qualitative or quantitative description of one aspect of a system's status at some instant in time. A state message is the communication of this quantity.

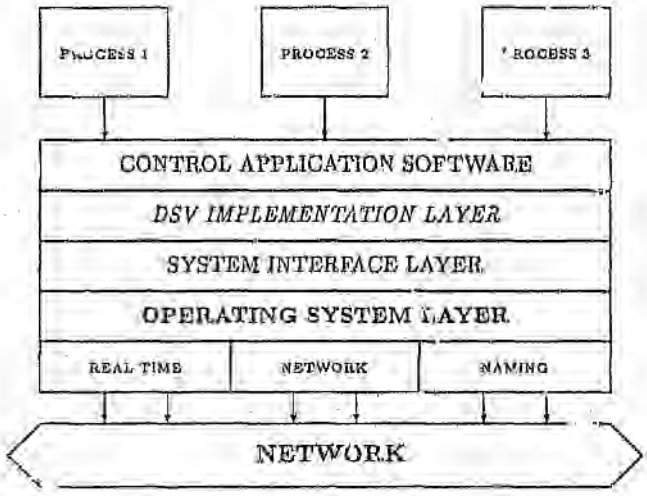


Figure 1: Layered structure of DSV software implementation

3 A ceptual Model

A Distributed State V able is a state-based interprocess communication mechanism for use in distributed systems. Each DSV has an associated expiry time and status byte, and real-time is thus an integral feature of the system. A DSV can only be written to by one process, the owner process, although it can be read by many processes. The only operating system feature that is required to support DSVs is a low-level primitive which guarantees read/write atomicity. DSVs can be polled with full knowledge that the required message will be found, thus doing away with the need for interrupts. The flow-of-control that is inherent in a physical plant makes the DSV mechanism extremely suitable for interprocess communication in control systems.

4 Layered Architecture

DSVLIB has been designed using a layered architecture. It must be emphasized that these layers are implicit layers, and no extra overhead is incurred due to interlayer communication. (Many layered architectures incur extra overhead as a result of interlayer protocols.) DSVLIB uses nested include files to implement layering. The C preprocessor is also used extensively. When the application code (which *includes* the correct files) is compiled, one executable file is obtained. This means that the run-time system is not layered, and the system only appears layered from the programmer's perspective. The *include* files can be viewed as pseudo-layers. This architecture is refered to as implicit layering.

The basic DSVLIB system comprises two layers, with "hooks" into two oth layers. The highest-level layer that has been implemented is the DSV primiti library, which is a collection of high level primitives for interprocess communication

contained in the file dsvfns.c. These primitives are written in terms of low-level operating system dependent functions. Any section of code which is independent of the operating system has been included in this file. All operating-system-dependent code is included in the lower layer *.dsv* file. This architecture permits the whole system to be ported, simply by re-writing the operating system dependent *.dsv* file in terms of the new operating system API. This means that the operating system dependent .dsv file implements the functions required by the file dsvfns.c in terms of the available operating system primitives. Obviously it is convenient to minimize the amount of operating system dependent code for easy of portability.

The file dsvfns.c provides the basic functionality for Process Control Application Objects (PCAO), which are presently being implemented as a distinct, higher layer. Thus the two layers which have been provided form the core of the DSVLIB system. An additional network interface layer will be required for DSVs to be supported in a distributed system.

5 Portability Issues

DSVLIB has been designed for easy portability to a variety of hardware and software platforms. This is one of the reasons that the C language was used to implement the system. The file dsvfns.c has been written using standard ANSI C, which is now supported by a large number of compilers. The *.dsv* file is the only file which is operating system dependent, and it must be re-written in terms of the new API. The pre-processor has been used extensively to mask hardware- and compiler-specific details. The preprocessor has also been used to interface the file dsvfns.c to the relevent *.dsv* file. Macros have been written to simplify the use of generalized functions. Standard preprocessor data type definitions, such as BYTE, WORD, LONG, BOOLEAN etc. have been used to promote portability.

6 Language Issues

As was mentioned above, standard ANSI C has been used in all code which is designed for portability (that is, excluding the operating system dependant *.dsv* layer). Function prototyping has been used even though older Unix systems do not support it. The trend however is to use prototyping, and the newer versions of Unix do support it.

7 Software Conventions

The C programming conventions employed by DSVLIB are mostly standard the major exception being the definition of macro functions. Macro functions are usually defined in upper case, and the functions they reference are written in lower case. It has been decided to write the macros in DSVLIB in lower case as well, so as to provide a consistent API. It is also desirable that the applications programmer view the macro functions as standard C library function calls. The functions referenced by the macros should never be called by the application program, because parameters could be omitted or incorrectly set, leading to incorrect and potentially dangerous programs. Additional types have been defined in terms of conventional types, to improve readability of code.

DSV is a type which specifies a DSV handle. It is defined as being of type LONG.
MSGVALID is a boolean type used to specify the validity of DSV contents. It can be either VALID (defined as 1) or INVALID (defined as 0). It is defined as an unsigned short integer.
EXPTIME is a type used to declare and manage the validity times of DSV contents. It is defined as a long integer.

It was also decided to use the preprocessor to specify errors and flags. Often the operating system provides programmers with header files which define meaningless error numbers and flags as more meaningful textual messages. DSVLIB error messages are specified in DSVERR.H.

8 DSVFNS.C

This file contains functions which provides basic interprocess communication primitives, and implements them in a manner that is completely independent of the operating system.

8.1 Preprocessor Definitions

The #define definitions are listed and explained below:

- MSGVALID - A data type used to declare boolean functions and variables which can return (or assume) the values VALID or INVALID.
- VALID - Defined to be 1. This is one of the two possible values of MSGVALID.
- INVALID - Defined to be 0. This is the other possible value of MSGVALID.
- EXPTIME - This is a long integer type that is used to declare functions or variables returning the expiry time of a DSV.
- NOT_YET_USED - This is a possible value for the status byte of the DSV. The status byte is automatically set to this value when the DSV is created. NOT_YET_USED is defined as 0x01 (1 hex).
- USED - This is defined as 0x02. The status byte of the DSV is automatically set to this value when the DSV is written to.
- ALARM - This is defined as 0x03. The status byte can be set to this value or another defined value to flag certain conditions. These conditions are treated as transient, and unless they persist, the status byte will be set to USED at the next write. Version 2 of DSVLIB will contain a more extensive list of defined DSV status possibilities, and the list will form a seperate header file.
- DELAY - This is a type used to declare a variable representing a real-time delay in hundredths of a second.
- TENDAYS - This defines the number 86400000 as a constant. The constant represents the number of hundredths of a second in ten days. The need for such a constant arises out of the necessity for a maximum validity time. See "check_time()" below.

get_time()

Function Prototype: EXPTIME get_time()
Parameters: - None.

Return Value: The current clock time is returned.

Description:
This function is called by check_time() to determine whether the DSV is still valid. It returns the current clock time when called.

check_time()

Function Prototype: MSGVALID check_time (EXPTIME exptime)
Parameters: **exptime** - Expiry time in hundredths of a second. (Type EXPTIME)

Return Value: VALID or INVALID of type MSGVALID.

Description:
This function is called by a process wishing check if the contents of a DSV are still valid. It returns either VALID or INVALID. The expiry time should not be a multiple field structure, as comparing times then becomes extremely complicated. The problem actually arises when one field in the structure "clocks over". The expiry time would then be "less" than the current time (looking at certain fields only), and it would appear as if the data in the DSV had expired. Consider for example the expiry time structure consisting of two fields, the first containing the number of hundredths of seconds since midnight, the second containing the number of days in the month. At midnight, the first field would reset to zero, and it is actually "greater" than the actual time first field, even though numerically the actual time first field is greater. To establish this as an absolute comparison, the second (day-of-the-month) fields would have to be compared. When the month changes, then months would have to be compared, and when years change, they would also have to be compared. The simplest way to solve this problem is to realize that when the expiry time clocks over, there will be an extremely large time discrepancy, and the failure of a process will only lead to a small time discrepancy. A single number is thus used to represent the expiry time (in hundredths of a second). For this reason the function check_time() discards extremely large timeout errors and attributes them to clocking over. This means that the maximum validity time that could be used if the system is to distinguish between clocking over and an expired DSV is about 14 days. It was decided that ten days would be more than sufficient. Note that the subtraction in the "if" statement will only be performed in the case of clocking over, because the logical "or" is terminated as soon as it is true.

dsv_wopen()

Function Prototype: DSV dsv_wopen (char *name)

Parameters: name - This is the name of the DSV instance as referenced in the configuration file.

Return Value: The handle (of type DSV) which is used to address the DSV.

Description:
This function is called by a process wishing to create a DSV. The calling process is given ownership of the DSV and rights to write to the DSV.

dsv_ropen()

Function Prototype: DSV dsv_ropen (char *name)

Parameters: name - This is the name of the DSV instance as referenced in the configuration file.

Return Value: The handle (of type DSV) which is used to address the DSV.

Description:
This function is called by a process wishing to "subscribe" to a DSV. The calling process is given read-only access to the DSV. If the DSV has not been created yet, an error message is returned, and the process will keep trying to subscribe until the DSV is created.

put_state()

Function Prototype: void put_state (DSV dsv_var, void *msg, LONG valid)

Parameters: dsv_var - This is the handle of the DSV that is to be written to.
msg - This is a void pointer specifying the address of the desired DSV contents.
valid - This is an integer type denoting the validity time in hundredths of a second.

Return Value: None.

Description:
This function is used by the owner process to write data to the DSV. The data is automatically time-stamped by the function. The expiry time is computed by adding the validity time (hundredths of a second) plus 3 (hundredths of a second) to the current time (also in hundredths of a second). The 3 hundredths of a second is used to round the result off to the nearest "clock-tick". The IBM PC/AT has a clock-tick of 32 ms.

get_state()

Function Prototype: MSGVALID get_state (DSV dsv_var, void *msg)

Parameters: dsv_var - This is the handle of the DSV that is to be read.
msg - This is a void pointer specifying an address to where the DSV contents must be copied.

Return Value: VALID or INVALID .

Description:
This function is used by subscriber processes to read data from a DSV. The function automatically inspects the validity time associated with the data, and returns ... of type MSGVALID . MSGVALID is a boolean type, VALID or INVALID. If the ... byte of the DSV is NOT_YET_USED, then get_state() returns an error, sayi... the writer process has not started yet, and INVALID is returned. This situation would arise where the writer process has created the DSV, but not started writing to it yet. If the writer has started (status byte not set to NOT_YET_USED), then get_state() calls check_time(), and VALID or INVALID is returned based on whether the DSV has expired or not.

flag_status()

Function Prototype: void flag_status (DSV dsv_var, void *msg, LONG valid, BYTE newstatus)

Parameters: dsv_var - This is the handle of the DSV that is to be written to.
msg - This is a void pointer specifying the address of the desired DSV contents.
valid - This is an integer type denoting the validity time in hundredths of a second.
newstatus - This is the value to which the status byte must be set.

Return Value: None.

Description:
This function is the same as put_state(), except that the value of the status byte can be changed. (i.e. to flag alarm conditions etc.) It must be noted that the new status byte value is not persistent, and as soon as put_state() is called again, the status byte will be set to USED. This makes intuitive sense, because a routine would use put_state() continuously, and flag_status() would only be used in the event of some abnormal condition. As long as the abnormal condition persists, flag_status() will be used, but when the situation reverts to normality, put_state() will once again be used, thus resetting the status byte to USED.

wait()

Function Prototype: void wait (DELAY delay)

Parameters: delay - This is the time to wait in hundredths of a second.

Return Value: None.

Description:
The function wait() starts a synchronous timer for delay hundredths of a second when called.

get_status()

Function Prototype: BYTE get_status (DSV dsv)

Parameters: dsv - This is the handle of the DSV that is to be queried.

Return Value: A BYTE specifying the status of the DSV is returned.

Description:
get_state() can only be used to obtain the contents of a DSV, but the need may arise for an applications program to check the status byte of a DSV. get_status() allows the application to "p ek" at the status byte of the DSV. This function is useful when debugging code.

get_expiry()

Function Prototype: EXPTIME get_expiry (DSV dsv)

Parameters: dsv - This is the handle of the DSV that is to be queried.

Return Value: A value of type EXPTIME is returned, reflecting the expiry time of the DSV that is being scrutinized

Description:
get_expiry() returns the expiry time of the DSV specified by the handle dsv. This function is also useful in debugging application modules, but would not normally be used in applications code. Applications should not normally deal with expiry times, because messages should simply be validated or invalidated by the underlying system.

get_conts()

Function Prototype: void get_conts (DSV dsv, BYTE* contents)

Parameters: dsv - This is the handle of the DSV that is to be queried.
contents - This is a pointer to BYTE array, in which the contents of the dsv are to be placed.

Return Value: None.

Description:
get_conts() extracts the contents of the contents of the DSV without paying any attention to the status byte or expiry time. It is also useful for debugging applications modules.

Together, the functions get_status(), get_expiry() and get_conts() allow applications to peek at the various DSV attributes. It will be noted that no corresponding functions are provided for setting the DSV attributes. The application should not be able to 'fiddle' the expiry time of a DSV, or modify its contents.

display_dsv()

Function Prototype: void display_dsv (DSV dsv_var)

Parameters: dsv_var - This is the handle of the DSV that is to be displayed.

Return Value: None.

Description:
This function is used for debugging applications modules that make use of DSVs. The values of the DSV status byte and expiry time are displayed when display_dsv() is called.

9 FlexOS.dsv

This file implements all operating-system dependant functions required by the file dsvfns.c in terms of standard FlexOS supervisor calls. (See [3]) The file includes several standard C library files, and then uses the preprocessor to simplify and reduce parameter specification in function calls. The file also contains a preprocessor interface of the file dsvfns.c to the file flexos.dsv.

9.1 Include Files

- stdio.h - This is a Kernighan and Ritchie (K&R) standard C file, which contains functions which route data to and from the standard input and output devices.
- string.h - This file is another K&R standard header file, which includes all the string processing functions.
- stdlib.h - This file includes I/O functions defined by ANSI and it includes some commonly used library routines.
- flags.h - This is a FlexOS header file which defines the various SVC parameters and flags as more readable abreviations.
- error.h - This is a FlexOS header file which defines the various error codes as more readable abreviations.

9.2 Preprocessor Definitions

The dsvfns.c to FlexOS.dsv interface is implemented entirely by preprocessor definitions. This is described after the function specifications. In addition to this interface, the following preprocessor definitions are used:

- READ - Defined to be 0 (zero). This is a boolean parameter passed to the function pipe_open().
- WRITE - Defined to be 1 (one). This is a boolean parameter passed to the function pipe_open().
- MSGLEN - This is defined to be the sum of the message contents length in bytes (CONTSLEN - defined in source code file.), the size of a LONG integer (for the expiry time), and one byte, used for representing the status.
- PIPE - This is a type for representing pipe handles. It is simply a LONG type.

9.3 Type Definitions

The structure dattime, consisting of 4 bytes, has been declared so that the results of FlexOS supervisor calls to the clock can be effectively accessed.

9.4 External Function Calls

Several FlexOS Supervisor Calls (SVCs), which are external to the file FlexOS.dsv are used. Documentation of these functions can be found in [3]. The functions used are the following:

- **s_create** - Creates a file, directory or pipe.
- **s_open** - Opens an existing file, pipe or device file.
- **s_read** - Extracts data from the specified file, pipe or device file.
- **s_write** - Places data into the specified file, pipe or device file.
- **s_close** - Disassociates an I/O stream from a file number.
- **s_get** - Extracts data from a table maintained by the operating system.
- **s_timer** - Delays the calling process until the specified time, or until the specified time has elapsed.

9.5 Functions

---

pipe_read()

Function Prototype: void pipe_read (LONG fnum, BYTE *msg)

Parameters: **fnum** - This is the file handle of the named pipe that is to be read from.
msg - This is a pointer to the byte array which the read contents of the pipe will be placed in.

Return Value: None.

Description:
pipe_read() is a preprocessor macro which calls the function **pipe_rd(fnum, msg, MSGLEN)** , where MSGLEN is the defined length (in bytes) of the message (and pipe). The read operation consumes the pipe contents.

---

pipe_nread()

Function Prototype: void pipe_nread (LONG fnum, BYTE *msg)

Parameters: **fnum** - This is the file handle of the named pipe that is to be read from.
msg - This is a pointer to the byte array which the read contents of the pipe will be placed in.

Return Value: None.

Description:
pipe_nread() is a preprocessor macro which calls the function pipe_nonrd(fnum msg, MSGLEN) , where MSGLEN is the defined length (in bytes) of the message (and pipe). This read operation is non-consuming.

pipe_write()

Function Prototype: void pipe_write (LONG fnum, BYTE *msg)

Parameters: **fnum** - This is the file handle of the named pipe that is to be written to.
msg - This is a pointer to the byte array which the contents of the pipe are contained in.

Return Value: None.

Description:
pipe_write() is a preprocessor macro which calls the function **pipe_wrt(fnum, msg, MSGLEN)** , where MSGLEN is the defined length (in bytes) of the message (and pipe). This function will only cause msg to be written to the pipe if the pipe is empty. pipe_write() will not cause a message in the pipe to be overwritten.

pipe_overwrt()

Function Prototype: void pipe_overwrt (LONG fnum, BYTE *msg)

Parameters: **fnum** - This is the file handle of the named pipe that is to be written to.
msg - This is a pointer to the byte array which the contents of the pipe are contained in.

Return Value: None.

Description:
pipe_overwrt() is a preprocessor macro which calls the function **pipe_ovwrt(fnum, msg, MSGLEN)** , where MSGLEN is the defined length (in bytes) of the message (and pipe). This function will cause any message in the pipe to be overwritten.

pipe_rd()

Function Prototype: void pipe_rd (LONG fnum, BYTE *buffer, LONG bufsiz)

Parameters: **fnum** - This is the file handle of the named pipe that is to be read from.
buffer - This is a pointer to the byte array which the read contents of the pipe will be placed in.
bufsiz - This is the defined length of the message (and pipe) as specicified in the definition MSGLEN.

Return Value: None.

Description:
pipe_rd() is a function which reads **bufsiz** bytes from the pipe. The bytes read are consumed.

pipe_nonrd()

Function Prototype: void pipe_nonrd (LONG fnum, BYTE *buffer, LONG bufsiz)

Parameters: **fnum** - This is the file handle of the named pipe that is to be read from.
buffer - This is a pointer to the byte array which the read contents of the pipe will be placed in.
bufsiz - This is the defined length of the message (and pipe) as specicified in the definition MSGLEN.

Return Value: None.

Description:
pipe_nonrd() is a function which reads **bufsiz** bytes from the pipe. The bytes read are not consumed and remain in the pipe.

pipe_wrt()

Function Prototype: void pipe_wrt (LONG fnum, BYTE *buffer, LONG bufsiz)

Parameters: **fnum** - This is the file handle of the named pipe that is to be written to.
buffer - This is a pointer to the byte array which contains the message to be written.
bufsiz - This is the defined length of the message (and pipe) as specified in the definition MSGLEN.

Return Value: None.

Description:
pipe_wrt() is a function which writes **bufsiz** bytes to the pipe. The bytes are only written to the pipe if it is not full. (The function pipe_wrt() does not overwrite.)

pipe_ovwrt()

Function Prototype: void pipe_ovwrt (LONG fnum, BYTE *buffer, LONG bufsiz)

Parameters: **fnum** - This is the file handle of the named pipe that is to be written to.
buffer - This is a pointer to the byte array which contains the message to be written.
bufsiz - This is the defined length of the message (and pipe) as specicified in the definition MSGLEN.

Return Value: None.

Description:
pipe_ovwrt() is a function which writes **bufsiz** bytes to the pipe. The bytes are written to the pipe even if it is full. (The function pipe_wrt() does not overwrite.) The function first calls pipe_rd() which consumes the existing message, and it then calls pipe_wrt() which writes the new message into the pipe.

---

put_fst_msg()

Function Prototype: void put_fst_msg (LONG fnumc)

Parameters: **fnumc** - The handle of the pipe that is to be initialized.

Return Value: None.

Description:
put_fst_msg() is called by the function pipe_open(). When a pipe is to be opened for writing, a dummy message must be written into the pipe. This is the purpose of this function. If the pipe is left empty, then the system will block while the non-consuming read operation is waiting for the pipe to be written to. The pipe overwrite function however, performs a consuming read and a pipe-write (normal) as an atomic operation. This means that both the overwrite and non-consuming read operations read the pipe first. Thus if the pipe is empty, neither reads or writes will be able to proceed and the system will be deadlocked. Writing a dummy message into the pipe on creation avoids this deadlocked situation. The contents and expiry time fields are set to zero. This could be a valid message at some time, and so the precaution of setting the status byte to 0x01 is taken. This informs the system that the pipe has been created, but not yet used.

---

pipe_open()

Function Prototype: LONG pipe_open (char *name, UWORD msg_size, unsigned int numbrmsgs, int R_W

Parameters: **name** - This is the name of the pipe that is to be created. The name is prefixed by "pi:@" and placed in the local pipe table.

msg_size - This informs the operating system of the size of the records that are to be written to the pipe.
numbrmsgs - This is the number of records that must be buffered by the pipe. The length of the pipe in bytes is calculated as the product of **msg_size** and **numbrmsgs**.
R_W - This is a boolean variable having the the value READ (0) or WRITE (1). It informs the function as to whether it is being called for the purposes of opening a pipe for reading or for writing.

Return Value: The handle of the pipe that has been created and opened for writing, or opened for reading is returned.

Description:
If pipe_open() is called with R_W set to WRITE, then the function calls the SVC s_create() and attempts to create the pipe. If the call to s_create() returns an error indicating that the pipe already exists, then the function opens the pipe for writing and returns the handle. If the pipe does not exist then the pipe is created, and put_fst_msg() is called to write a dummy message ito the pipe. The pipe is then opened for writing, and the handle is returned. If pipe_open() is called with R_W set to READ, the function attempts to open the pipe for reading. If the pipe has not yet been created by the owner process, then an error is returned indicating that the pipe does not yet exist. An error message is printed out, and the function attempts to open the pipe once again. This continues until the open succeeds (i.e. when the pipe has been created.) The function then returns the handle.

---

pipe_close()

Function Prototype: void pipe_close (LONG fnum)

Parameters: **fnum** - This is the handle of the pipe that is to be closed.

Return Value: None.

Description:
This function closes the pipe specified by the handle fnum.

9.6 Interface to DSVFNS.C

No FlexOS specific functions have been used in the file **dsvfns.c**. Instead, all the FlexOS specific functions which need to be called are defined by another name, more generic in nature, using the preprocessor. These generic names are used in the file **dsvfns.c** . It is not only FlexOS specific functions which are masked using this generic naming system, but also FlexOS specific implementation details. For example, the FlexOS implementation of DSVs uses named pipes as the underlying IPC. Any user-written, FlexOS implementation specific function that the file **dsvfns.c** requires, is also masked by the generic naming system. This generic naming system provides **dsvfns.c** with a polymorphic interface to the operating system dependent layer. This makes **DSVLIB** easy to port to new operating systems.

- **ropen_dsv(name)** - This is defined to call the FlexOS implementation dependent function pipe_open() for reading, with a total length of one message, *MSGLEN* bytes long.
- **wopen_dsv(name)** - This is defined to call the FlexOS implementation dependent function pipe_open() for writing, with a total length of one message, MSGLEN bytes long.
- **state_read(dsv, msg)** - This is defined as the function pipe_nread().
- **state_write(dsv, msg)** - This is defined as the function pipe_overwrt().
- **close_dsv(dsv)** - This is defined as the function pipe_close().
- **obtain_time(dat)** - This function is defined to call the FlexOS SVC **s_get()**, with parameters set to extract the time and date from the time table.
- **synch_timer(delay)** - This function is defined to call the FlexOS SVC **s_timer(delay)**, which starts a synchronous timer for **delay** hundredths of a second. Note the factor of ten in the SVC, which is required because **s_timer** expects the delay to be specified in thousandths of a second.

10 Using DSVLIB

10.1 Including the File

The functions in DSVLIB can be used if the file **dsvfns.c** is included in the application source file. Before the include statement is specified, the constant **CONSLEN** must be defined. **CONTSLEN** is the size in bytes of the message, *excluding* the expiry time and status byte.

10.2 Configuration Files and Implementation Files

For one process to read a DSV created and written to by another process, the second process has to know the name by which the operating system refers to the DSV. This information is specified by means of configuration files. Each executable module has to have its own .cfg file, which relates the local names to operating system resource names.

For example, the module *TANK* might be a monitoring process which writes the temperature of the tank into a DSV called *TEMP* (local name). A module *MONITOR* may wish to access this value, but it might wish to refer to the value as *TANKTEMP* . The operating system could call the DSV by any name it wishes. It could for instance, call it *"XYZ"* . The file *tank.c* . would then have to include a file *tank.cfg* , which would define the following relationship:

#define TEMP "XYZ"

The file *monitor.c* would have to include a file *monitor.cfg* , defining the following relationship:

#define TANKTEMP "XYZ"

This system of configuration enables the programmer to give each DSV a local context name. The DSV name could have been the same in each of the modules and in the operating system. The files *tank.cfg* and *monitor.cfg* would both contain the following definition:

#define TANKTEMP "TANKTEMP"

10.3 Adding to the Library

The library has been designed to provide a full set of IPC primitives. The only extra functionality that may be required is a set of extended debugging functions.

10.4 Using DSVLIB as the basis for other Libraries

DSVLIB is ideally suited to applications which require reliable, fault-tolerant, real-time communication in a distributed system. Libraries of higher level functions, for example control and monitoring functions, can use DSVLIB as the underlying layer simply by including it in new source files.

References

[1] Ben-Ari, M (1982) *Principles of Concurrent Programming*, Prentice-Hall

[2] Cohen, J E and MacLeod, I M (1989) *Software Design for Real-Time Distributed Computer Control Systems*, Internal Report, Dept Elect. Engineering, Univ. of the Witwatersrand

[3] FlexOS Programmers Guide, (1986) *Documentation - FlexOS 386 Operating System, Digital Research Corp.*

[4] Hinden, R J and Rauch-Hinden, W B (1983) *Special Series on System Integration*, Electronic Design, January 1983

[5] Kopetz, H et al (1985) *A Message-Based DCCS*, IFAC Conference on DCCS, Sab. Sabi, South Africa, (1985)

[6] Kramer J, Magee J, and Sloman M (1984) *A Software Architecture for Distributed Computer Control Systems*, Proceedings IFAC Conference on DCCS (1984)

[7] Letwin, G (1988) *Inside OS/2*, Microsoft Press, pg 122

[8] Kernighan, B.W. and Ritchie, D.M. (1988), The State of C, *Byte*, Vol. 13 No. 8, pp 205-210

[9] Kernighan, B. W. and Ritchie, D. M. (1978), *The C Programming Language*, Prentice-Hall

[10] Harbison, S. P. and Steele, G. L. (1984), *A C Reference Manual*, Prentice-Hall, Englewood Cliffs, N.J.

[11] Adams, M. (1988), FlexOS - A DOS Compatible Real-Time OS, *.EXE Magazine*, pp 65-69

[12] Linowes, J S (1988) *The C Language in Depth - It's an Attitude!*, Byte, August 1988

[13] Macleod, I M and Rodd M G (1983) *Interprocess Communication Primitives for Distributed Process Control*, IFAC Symposium on DCCS, Sabi Sabi, South Africa, (1983)

[14] Morris R R and Brooks, W E (1988) *At the Core: An API Comparison*, PC-Tech Journal, Vol 6, No 12, December 1988, pg 62

[15] Pohl, I and Edelson, D (1988) *A to Z: C Language Shortcomings*, Computer Language, Vol 13, No 2, pp. 51-64

[16] Tannenbaum, A S and Van Renesse R (1985) *Distributed Operating Systems*, Computing Surveys, vol 17 No 4, December 1985

Appendix B

DSVLIB: Source Listings

This appendix contains the listings generic file dsvfns.c , as well as the operating system-specific files flexos.dsv, os2.dsv and dsvfns.dsv (which is used in conjunction with our DOS-based kernel). The DOS kernel code and FlexOS and OS/2 support programs are not included. The file dsvfns.c is portable, and only the .dsv file inclusion statement must be changed.

1 DSVFNS.C

```
/*=============================================================================

                                DSVFNS.C

            DSV Functions for Real-Time Distributed Programs

                    Version 1.5     -     18 January 1990

                                Jack E. Cohen
                            University of the Witwatersrand
                                  Johannesburg

=============================================================================*/

#define  CONTSLEN   sizeof(double)
#include "os2.dsv"

typedef struct {
        BYTE conts[CONTSLEN];
        long int expire;
        BYTE status;
        } message;

#define DSV             LONG
#define MSGVALID        unsigned short int
#define VALID           1
#define INVALID         0
#define EXPTIME         long int
#define NOT_YET_USED    0x01
#define USED            0x02
#define ALARM           0x03
#define DELAY           LONG
#define TENDAYS         86400000  /* hundredths of a second in 10 days */

#define dsv_ropen(name)         ropen_dsv(name)
#define dsv_wopen(name)         wopen_dsv(name)
#define dsv_close(name)         close_dsv(name)

EXPTIME get_time()
    {
    dattime dat;
    LONG ret;
    obtain_time(dat);
    return(dat.time/10L + 24L * 3600L * 100L * dat.day);
    }

MSGVALID check_time(EXPTIME exptime)
    {
    EXPTIME timenow;
    timenow = get_time();
    if ((timenow < exptime) || ((timenow - exptime)) > TENDAYS) return(VALID);
       else return(INVALID);
```

```
    }

void put_state(DSV dsv_var, void *msg, LONG valid)
     {
     message messg;
     memcpy(&(messg.conts), msg, (size_t)CONTSLEN);
     messg.expire = get_time() + valid + 3;      /* 3 is clock tick resolution */
     messg.status = USED;                        /* in hundredths of a second. */
     state_write(dsv_var, (BYTE *)&messg);       /* i.e. resoultion is 32ms.   */
     }

void flag_status(DSV dsv_var, void *msg, LONG valid, BYTE newstatus)
     {
     message messg;
     memcpy(&(messg.conts), msg, (size_t)CONTSLEN);
     messg.expire = get_time() + valid + 3;.     /* 3 is clock tick resolution */
     messg.status = newstatus;
     state_write(dsv_var, (BYTE *)&messg);       /* in hundredths of a second. */
     }                                           /* i.e. resolution is 32ms.   */

MSGVALID get_state(DSV dsv_var, void *msg)
     {
  void wait();
     message messg;
     MSGVALID curvalid;
     state_read(dsv_var, (BYTE *)&messg);
     memcpy(msg, &(messg.conts), (size_t)CONTSLEN);
     if (messg.status == NOT_YET_USED)
        {
        printf("Waiting for the writer process to start writing...\n");
        return(INVALID);
        }
        else
           {
           curvalid = check_time(messg.expire);
           if (!curvalid)
                 {
                 printf("Message has expired !\n");
                 return(INVALID);
                 }
                 else return(VALID);
           }
     }

void wait(DELAY delay)      /* delay is in hundredths of a sec   */
     {
     LONG ret;
     synch_timer(delay);
     }

BYTE get_status(DSV dsv)
     {
     message msg;
     int ret;
     state_read(dsv, (BYTE *)&msg);
```

```
    return(msg.status);
    }

EXPTIME get_expiry(DSV dsv)
    {
    message msg;
    int ret;
    state_read(dsv, (BYTE *)&msg);
    return(msg.expire);
    }

void get_conts(DSV dsv, BYTE *contents)
    {
    message msg;
    int ret;
    state_read(dsv, (BYTE *)&msg);
    memcpy(contents, &(msg.conts), sizeof(CONTSLEN));
    }

void display_dsv(DSV dsv_var)
    {
    printf("                                          STATUS OF DSV\n");
    printf("                                          %ld\n", get_expiry(dsv_var));
    printf("                                          ");
    switch (get_status(dsv_var))
          {
          case  NOT_YET_USED     :   printf("NOT_YET_USED");
                                     break;
          case  USED             :   printf("USED");
                                     break;
          case  ALARM            :   printf("ALARM");
                                     break;
          default                :   printf("INDETERMINATE STATUS");
          }
    return;
    }
```

2 FLEXOS.DSV

```
/*=============================================================================

                                FlexOS.dsv

   Functions which tailor IPCs into a form suitable for use by dsvfns.c

                Version 1.5     -       18 January 1990

                                Jack E. Cohen
                            University of the Witwatersrand
                                Johannesburg

=============================================================================*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "flags.h"
#include "error.h"

#define READ     0
#define WRITE    1
#define MSGLEN   CONTSLEN + sizeof(LONG) + sizeof(BYTE)

#define PIPE LONG
#define pipe_read(fnum, msg)      pipe_rd(fnum, msg, MSGLEN)
#define pipe_nread(fnum, msg)     pipe_nonrd(fnum, msg, MSGLEN)
#define pipe_write(fnum, msg)     pipe_wrt(fnum, msg, MSGLEN)
#define pipe_overwrt(fnum, msg)   pipe_ovwrt(fnum, msg, MSGLEN)

typedef struct{
        UWORD year;
        BYTE month;
        BYTE day;
        LONG time} dattime;

EXTERN LONG s_create();
EXTERN LONG s_open();
EXTERN LONG s_read();
EXTERN LONG s_write();
EXTERN LONG s_close();
EXTERN LONG s_get();
EXTERN LONG s_timer();

void pipe_rd(LONG fnum, BYTE *buffer, LONG bufsiz)
     {
     LONG  nbytes;

     nbytes = s_read(A_FLUSH|A_EOFOFF, fnum, buffer, bufsiz, (LONG)0);
     if (nbytes < 0L) check_error(nbytes);
     }
```

```
void pipe_nowrd(LONG fnum, BYTE *buffer, LONG bufsiz)
     {
     LONG  nbytes;

     nbytes = s_read(A_FLUSH|A_NODESCT|A_EOFOFF, fnum, buffer, bufsiz, (LONG)0);
     if (nbytes < 0L) check_error(nbytes);
     }


void pipe_wrt(LONG fnum, BYTE *buffer, LONG bufsiz)
     {
     LONG  nbytes;

     nbytes = s_write(A_FLUSH|A_EOFOFF, fnum, buffer, bufsiz, (LONG)0);
     if (nbytes < 0L) check_error(nbytes);
     }

void put_fst_msg(LONG fnumc)
     {
     message dummy;
     int index;
     for(index = 0; index < CONTSLEN; index++)
        dummy.conts[index] = 0;
     dummy.expire = 0x0000;
     dummy.status = 0x01;      /* NOT_YET_USED */
     pipe_wrt(fnumc, (BYTE *)(&dummy), MSGLEN);
     }

LONG pipe_open(char *name, UWORD msg_size, unsigned int numbrmsgs, int R_W)
     {
     char pipename[20];
     UWORD security = 0x0FFF;
     LONG fnumc, fnum, nbytes;

     sprintf(pipename, "local::pi:0%s", name);
     if (R_W == WRITE)
        {
        fnumc = s_create( 0_FILE,
                          A_WRITE|A_READ|A_SHARE|A_TEMP|A_CONTIG|A_SECURITY,
                          (BYTE *)pipename,
                          msg_size,
                          security,
                          (LONG)(msg_size * numbrmsgs));
        if ((UWORD)fnumc != E_EXISTS) put_fst_msg(fnumc);
        fnum = s_open( A_WRITE|A_READ|A_SHARE|A_REDUCE, (BYTE *)pipename);
        }
     if (R_W == READ)
        {
        for(;;)
           {
           fnum = s_open(A_READ|A_SHARE|A_REDUCE, (BYTE *)pipename);
           if ((UWORD)fnum == E_NO_FILE) printf("DSV has not been created !\n");
              else break;
           }
        }
```

```
void pipe_nonrd(LONG fnum, BYTE *buffer, LONG bufsiz)
     {
     LONG  nbytes;

     nbytes = s_read(A_FLUSH|A_NODESCT|A_EOFOFF, fnum, buffer, bufsiz, (LONG)0);
     if (nbytes < 0L) check_error(nbytes);
     }


void pipe_wrt(LONG fnum, BYTE *buffer, LONG bufsiz)
     {
     LONG  nbytes;

     nbytes = s_write(A_FLUSH|A_EOFOFF, fnum, buffer, bufsiz, (LONG)0);
     if (nbytes < 0L) check_error(nbytes);
     }

void put_fst_msg(LONG fnumc)
     {
     message dummy;
     int index;
     for(index = 0; index < CONTSLEN; index++)
        dummy.conts[index] = 0;
     dummy.expire = 0x0000;
     dummy.status = 0x01;      /* NOT_YET_USED */
     pipe_wrt(fnumc, (BYTE *)(&dummy), MSGLEN);
     }

LONG pipe_open(char *name, UWORD msg_size, unsigned int numbrmsgs, int R_W)
     {
     char pipename[20];
     UWORD security = 0x0FFF;
     LONG fnumc, fnum, nbytes;

     sprintf(pipename, "local::pi:0%s", name);
     if (R_W == WRITE)
        {
        fnumc = s_create( 0_FILE,
                          A_WRITE|A_READ|A_SHARE|A_TEMP|A_CONTIG|A_SECURITY,
                          (BYTE *)pipename,
                          msg_size,
                          security,
                          (LONG)(msg_size * numbrmsgs));
        if ((UWORD)fnumc != E_EXISTS) put_fst_msg(fnumc);
        fnum = s_open( A_WRITE|A_READ|A_SHARE|A_REDUCE, (BYTE *)pipename);
        }
     if (R_W == READ)
        {
      . for(;;)
           {
           fnum = s_open(A_READ|A_SHARE|A_REDUCE, (BYTE *)pipename);
           if ((UWORD)fnum == E_NO_FILE) printf("DSV has not been created !\n");
               else break;
           }
        }
```

```
    return(fnum);
    }

void pipe_ovwrt(LONG fnum, BYTE *buffer, LONG bufsiz)
    {
    LONG  nbytes, position;
    BYTE *killmsg;

    killmsg = malloc(bufsiz);
    pipe_rd(fnum, killmsg, bufsiz);
    nbytes = s_write(A_FLUSH|A_EOFOFF, fnum, buffer, bufsiz, (LONG)0);
    if (nbytes <= 0L) check_error(nbytes);
    free(killmsg);
    }


void pipe_close(LONG fnum)
    {
    s_close(A_PCLOSE, fnum);
    }

/*=============================================================================*/
/*                  DEFINING THE DSV / FlexOS INTERFACE                        */
/*=============================================================================*/

#define   ropen_dsv(name)           pipe_open(name, MSGLEN, 1, READ)
#define   wopen_dsv(name)           pipe_open(name, MSGLEN, 1, WRITE)
#define   state_read(dsv, msg)      pipe_nread(dsv, msg)
#define   state_write(dsv, msg)     pipe_overwrt(dsv, msg)
#define   close_dsv(dsv)            pipe_close(dsv)
#define   obtain_time(dat)          s_get(0x0002, 0L, &dat, 8L)
#define   synch_timer(delay)        s_timer((UWORD)0, 10 * delay)
```

3 OS2.DSV

```
/*=============================================================================

                                DS2.DSV

    DSVs for OS/2 v 1.2

      This file contains functions which support a shared-memory
      implementation of DSVs. The Program DSVINIT must be run in
      a separate session. The network and inter-node communication
      is performed by DSVINIT.

                Jack E. Cohen          -          12 July 1990

=============================================================================*/

#include  "os2.h"
#include  "errors.c"

#define   NUL           (USHORT)0
#define   size_t        USHORT
#define   DONE          1
#define   NOTDONE       0

#define   MAXDSVS           100       /* Maximum number of DSVs allowed */
#define   MSGLEN            CONTSLEN + sizeof(LONG) + sizeof(BYTE)
#define   WAITFOREVER     -1L       /* no timeout if semaphore is owned */

#define   DSVINIT     dsv_initialization()
#define   SHRSEGNAME "\\SHAREMEM\\SHARESEG.DAT"   /* shared seg name */
#define   SEMNAME         "\\SEM\\MEMCTRL"             /* semaphore name  */


typedef struct dsvmsg{
char name[9];
BYTE conts[CONTSLEN];
LONG expire;
BYTE status;
}DSVS;

typedef struct dsvrec{
        DSVS dsv;
        char name[9];
        }dsv_data;


#define    DSV     long int

struct ShareRec {
dsv_data dsvtable[MAXDSVS+1];
};

int numbrdsvs = 0;
struct ShareRec * ShPtr;
```

```
HSYSSEM Semhandle;
PIDINFO proc_id;

#define   LOCK          DosSemRequest(&Semhandle, WAITFOREVER)
#define   UNLOCK        DosSemClear(&Semhandle)

#define   TASK(tak)                   main()

#define   gotoxy(x, y)      VioSetCurPos(x, y, 0)

dsv_initialization()
{

        SEL Selector;
unsigned rc; /* return code */

/* allocate the shared memory segment */
if (rc = DosGetShrSeg((PSZ)SHRSEGNAME, &Selector
{
            printf("Access to shared memory failed, error: %d\n", rc);
            DosExit(EXIT_PROCESS, 0);
            }

/* Get a far pointer from a 16 bit selector */
ShPtr = (struct ShareRec *) MAKEP(Selector, 0);

/* allocate the shared memory segment semaphore */
if (rc = DosOpenSem(&Semhandle, (PSZ)SEMNAME ))
{
            printf("Access to memory control semaphore failed, error: %d\n", rc);
            DosExit(EXIT_PROCESS, 0);
            }
UNLOCK;
DosGetPid(&proc_id);
DosSetPrty(PRTYS_PROCESS, PRTYC_TIMECRITICAL, 31, proc_id.pid);
printf("Linked to Shared Memory \n");
}


DSV init_a_dsv_record(int numbr)
{
int index;

strcpy((ShPtr->dsvtable[numbr]).name, "          ");
for(index=0; index < CONTSLEN; index++)
   (ShPtr->dsvtable[numbr]).dsv.conts[index] = 0;
(ShPtr->dsvtable[numbr]).dsv.expire = 0x0000;
(ShPtr->dsvtable[numbr]).dsv.status = 0x01;
numbrdsvs++;
return((DSV)numbr);
}


DSV look_up_dsv(char *dsvname, BOOL destroy)
{
```

```
int index;
int comp;
index = 0;
        LOCK;
for (index = 1;  index < (MAXDSVS+1); index++)
            {
            comp = strcmp((ShPtr->dsvtable[index]).name, dsvname);
            if (!comp)
{
if (destroy)
   {
   init_a_dsv_record(index);
                    }
return((DSV)index);
}
            }
        UNLOCK;
return((DSV)(NUL));
}


DSV init_dsv_record(char * dsvname)
{
int index;

strcpy((ShPtr->dsvtable[numbrdsvs+1]).name, dsvname);
for(index=0; index < CONTSLEN; index++)
   (ShPtr->dsvtable[numbrdsvs+1]).dsv.conts[index] = 0;
(ShPtr->dsvtable[numbrdsvs+1]).dsv.expire = 0x0000;
(ShPtr->dsvtable[numbrdsvs+1]).dsv.status = 0x01;
numbrdsvs++;
return((DSV)(numbrdsvs));
}



DSV wopen_dsv(char * dsvname)
{
int index;
DSV exist;
exist = look_up_dsv((char *)dsvname, (BOOL)0);
        printf("Exist = %ld\n ", exist);
if ((BOOL)exist) return((DSV)exist);
if (numbrdsvs > 100) printf("DSV table is full.\n");
else
{
                        LOCK;
return(init_dsv_record(dsvname));
                        UNLOCK;
                        }
return(NUL);
}


DSV ropen_dsv(char * dsvname)
```

```
{
DSV dsvhandle = NUL;
        dsvhandle = (DSV)look_up_dsv((char *)dsvname, (BOOL)0);
while(!dsvhandle)
     {
             printf("DSV has not been created !\n");
     DosSleep(100L);
         /*      error_message(DSVNCRTD);     */
             dsvhandle = (DSV)look_up_dsv((char *)dsvname, (BOOL)0);
     }
return(dsvhandle);
}

int close_dsv(dsvname)
{
if (!(DSV)look_up_dsv(dsvname, 1)) return(DONE);
   else
{
printf("Cannot close DSV! \n");
return(NOTDONE);
}
}

void state_write(DSV dsvhandle, BYTE *newdata)
   {
   UNLOCK;
   LOCK;
   memcpy((void *)(&((ShPtr->dsvtable[dsvhandle]).dsv.conts)),
                                        (void *)newdata, MSGLEN);
   UNLOCK;
   return;
   }

void state_read(DSV dsvhandle, BYTE *olddata)
   {
   UNLOCK;
   LOCK;
   memcpy((void *)olddata,
        (void *)(&((ShPtr->dsvtable[dsvhandle]).dsv.conts)),
                                                    MSGLEN);
   UNLOCK;
   return;
   }

typedef struct dat_time
{
long int time;
int day;
}dattime;

#define obtain_time(dat)   dos_obtain_time(&dat)

Void dos_obtain_time(dattime * dtstruct)
{
DATETIME dstruct;
```

```
DosGetDateTime(&dstruct);
dtstruct->time = (long int)((dstruct.hours*3600LU*1000LU)
                                    + (dstruct.minutes*60LU*1000LU)
                + (dstruct.seconds*1000LU) +
                                            (dstruct.hundredths*10LU));
dtstruct->day = (int)dstruct.day;
}


void synch_timer(LONG hsecs)
     {
     DosSleep(hsecs*10L);
     }
```

4 DSVSYS.DSV

```
/*=============================================================================

                                DSVSYS.DSV

        DSVs for DOS-based DSVKERN

      This file contains functions which support a shared-memory
      implementation of DSVs.

                Jack E. Cohen           -           1 March 1990

=============================================================================*/

#define   MAXDSVS     100
#define   DONE        1
#define   NOTDONE     0

#define   MSGLEN      CONTSLEN + sizeof(LONG) + sizeof(BYTE)

typedef struct dsvmsg{
char name[9];
BYTE conts[CONTSLEN];
LONG expire;
BYTE status;
}DSVS;

#define    DSV      struct dsvmsg *

DSV dsvtable[MAXDSVS];
int numbrdsvs;


void set_dsvs_nul()
{
int index;
for (index=0; index < MAXDSVS; index++)
    dsvtable[index] = NULL;
return;
}


DSV wopen_dsv(char *dsvname)
{
int index;
if (numbrdsvs >= 100) printf("DSV table is full.\n");
else
{
dsvtable[numbrdsvs] = (DSV)malloc(sizeof(DSVS));
if (dsvtable[numbrdsvs])
{
strcpy(dsvtable[numbrdsvs]->name, dsvname);
for(index=0; index < CONTSLEN; index++)
   dsvtable[numbrdsvs]->conts[index] = 0;
```

```
dsvtable[numbrdsvs]->expire = 0x0000;
dsvtable[numbrdsvs]->status = 0x01;
numbrdsvs++;
return(dsvtable[numbrdsvs-1]);
}
  else
    {
    printf("Error in allocating DSV.\n");
    return(NULL);
    }
}
return(NULL);
}

DSV look_up_dsv(char *dsvname, BOOLEAN destroy)
{
int index;
index = 0;
for (index = 0;  index <= numbrdsvs; index++)
if (!strcmp(dsvtable[index]->name, dsvname))
{
if (destroy)
   {
   free(dsvtable[index]);
   dsvtable[index] = NULL;
   }
return(dsvtable[index]);
}
printf("DSV not exist.\n");
return(NULL);
}

DSV ropen_dsv(char *dsvname)
{
DSV dsvhandle;
while(!(dsvhandle =(DSV)look_up_dsv(dsvname, 0)))
     {
     printf("DSV has not been created yet !\n");
     yield();
     }
return(dsvhandle);
}

int close_dsv(dsvname)
{
if (!((DSV)look_up_dsv(dsvname, 1))) return(DONE);
   else
{
printf("Cannot close DSV! \n");
return(NOTDONE);
}
}

void state_write(DSV dsvhandle, BYTE *newdata)
   {
```

```
   memcpy((void *)(&(dsvhandle->conts)), (void *)newdata, MSGLEN);
   return;
   }

void state_read(DSV dsvhandle, BYTE *olddata)
   {
   memcpy((void *)olddata, (void *)(&(dsvhandle->conts)), MSGLEN);
   return;
   }

typedef struct
{
long int time;
int day;
} dattime;

#define obtain_time(dat)   dos_obtain_time(&dat)

void dos_obtain_time(dattime *dtstruct)
{
struct time gtime;
struct date gdate;
gettime(&gtime);
getdate(&gdate);
dtstruct->time = (long int)((gtime.ti_hour*3600LU*1000LU) +
                  (gtime.ti_min*60LU*1000LU) + (gtime.ti_sec*1000LU)
                                             + (gtime.ti_hund*10LU));
dtstruct->day = (int)gdate.da_day;
}

void synch_timer(LONG hsecs)
     {
     long int reftim, get_time();
     unsigned short int check_time();
     reftim = get_time();
     while (check_time(reftim + hsecs)) yield();
     }
```

Appendix C

A Language for Programming Real-time Control Systems

AN EXTENSIBLE LANGUAGE FOR PROGRAMMING AND CONFIGURING DCCS

I. M. MacLeod J. E. Cohen
University of the Witwatersrand
Johannesburg, South Africa

Abstract

The requirements for a process control programming language for next-generation computer control systems are examined. A language based on the the C programming language and a theory of process interaction is proposed. The language incorporates facilities for the management of time at the data representation level. Program examples for a distributed adaptive controller are presented and discussed to show how the language meets its requirements.

1 Introduction

The growing complexity of computer control systems has led to a realization that better techniques for programming and configuring such systems are required. The need for specialized interprocess communication primitives for use in distributed real-time control systems was identified several years ago. A number of proposals for such primitives have been put forward [1], however these proposed primitives have typically not developed into languages which can be used to program modern computer control applications.

With the advent of computing hardware which is capable of executing modern control algorithms in real-time, a need has arisen for programming languages which can be used to program low-level set-point control applications, higher level optimizing control applications, as well as supervisory control applications. In future computer control systems these applications will be distributed.

Real-time distributed computer control systems require the development of low-level and high-level software. The ideal programming language for computer control systems must facilitate the development of software at various levels of abstraction. This is necessary because many modern control techniques require the availability of high level programming concepts for the implementation of what would previously have been regarded as low-level control functions. Thus any next-generation control programming language must be extremely flexible.

In this paper we review the structure of traditional control software, and examine the requirements for future control programming languages. A control programming language which is based on an interprocess communication primitive is proposed. The primitive provides inherent real-time support, and facilitates the dissemination of real-time information in distributed systems. We show how this language meets the requirements identified at the outset.

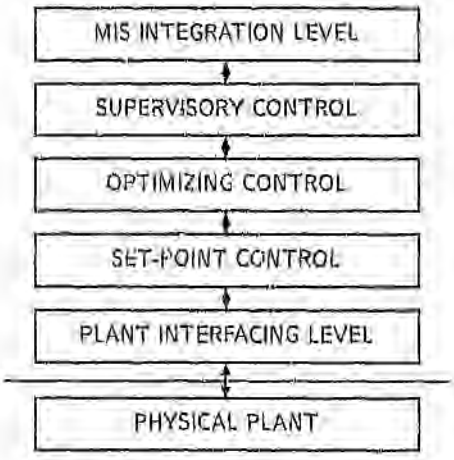


Figure 1: Structure of control software

2 Structure of control software

Computer control systems are capable of performing a number of different control functions. At the lowest level, a computer control system is required to interact with the physical plant by means of suitable hardware. At the uppermost level, the computer control system is required to make information available for integration into management information systems. A number of *levels of control* have been identified between these two extremes [2] (see fig. 1).

The boundaries between these layers are becoming blurred, and languages for programming next-generation computer control systems must reflect this.

3 Language requirements

As a result of the evolution of computer control systems, programming languages for modern computer control systems must meet a number of requirements.

Resource distribution capabilities A distributed computer control system consists of a number of loosely-coupled processing modules which may communicate with one another only by message-passing over some shared communications network. Each processing module is capable of executing one or more processes (concurrently). The language must be enable these distributed processes to communicate with one another in a way which masks the fact that the processes are executing on different processing modules.

Temporal correctness In real-time computer control systems information is only correct if it is quantitatively *and* temporally valid. The use of modern control techniques has further highlighted the need to ensure *correct* real-time system behaviour. Stankovic [3] and Kopetz [4] have recognized that management of time should be incorporated into real-time systems at the lowest possible level.

Horizonal and vertical extensibility It must be possible to add both lower-level set-point and direct digital control functions and higher level optimizing and adaptive control functions to the language at any time.

Intuitive programming A programmer developing software at any particular level using the language should not be required to possess any programming skills other than those which would be intuitive to him.

Redundancy management and fault detection The language should provide mechanisms which support redundancy management and facilitate failure detection.

System verification It should be possible to verify application software by means of simulation or some other mechanism.

Computer-aided software engineering At the highest level of programming, programmers should be able to make use of a computer-based program design and configuration tool.

4 The language

4.1 Interprocess communication

The first two requirements listed above can be met if the language is based on a suitable interprocess communication primitive. A language based on Distributed State Variables (DSV) is proposed. DSVs are an IPC primitive implementation which is based on our *p_var* theory [5]. The primitives were specifically designed for use in distributed real-time computer control applications, and they inherently incorporate facilities for the management of time whilst still supporting communication of data in distributed systems.

4.1.1 The p_var model

A *p_var* is used to represent a property of the plant. It consists of a name field, a sample value, and a sample lifetime. Sample value-lifetime pairs are treated as atomic units. Time management features are thus incorporated into the system at the data representation level.

Plant properties are represented in *p_vars* as state values. *P_vars* are persistent data entities which are overwritten periodically. System faults and communication failures can be detected when *p_var* sample values expire. Thus real-time constraints can be catered for if *p_var* validity times are carefully chosen.

Because each *p_var* represents a single plant property, each *p_var* can only be updated by a single process (physical or otherwise). We say that the process which updates the *p_var* "owns" the *p_var* [5]. The *single ownership principle* must be strictly adhered to if data integrity is to be assured.

When multiple sensors are used to sample the same property, each measurement must be represented in a unique *p_var*. The act of measurement separates multiple measurements of the same quantity into several separate and distinct quantities. No two sensors have identical characteristics, and each sensor may fail independently. Thus the *p_var* model requires that each *measurement* be represented as a unique *p_var* in the system. It may seem as if this approach compromises redundancy management—it will be shown that, on the contrary, it supports multiple redundancy configurations whilst providing inherent facilities for failure detection and fault isolation.

Each process may have *p_vars* which it owns, and *p_vars* which it requires read-only access to (*image p_vars*). Each process periodically broadcasts the *p_vars* which it owns onto the network. It also reads the *p_vars* which it requires read-only access to, off the network. Real-time information is disseminated throughout the distributed system in this way, and a real-time distributed database exists as a set of *p_vars* on the network.

When a value is inserted into a *p_var* it is time-stamped automatically. Similarly, the operation of reading a *p_var* value causes the validity time to be inspected automatically.

The *p_var* model thus allows us to design a set of IPC primitives which allows the first two requirements to be met.

4.1.2 The DSV implementation

The DSV implementation of the *p_var* process interaction model consists of the following primitives:

dsv_wopen()

Function prototype: DSV dsv_wopen (char *name)
This function is called by a process wishing to create a DSV. The calling process is given ownership of the DSV and rights to write to the DSV.

dsv_ropen()

Function prototype: DSV dsv_ropen (char *name)
This function is called by a process wishing to instantiate an image DSV. The calling process is given read-only access to the DSV. If the DSV does not exist in the system, an error message is returned and the process will continue trying to instantiate the image DSV.

put_state()

Function prototype: void put_state (DSV dsv_var, void *msg, LONG valid)
This function is used by the owner process to write data to the DSV. The data is automatically time-stamped by the function.

get_state()

Function prototype: MSGVALID get_state (DSV dsv_var, void *msg)
This function is used by subscriber processes to read a value from a DSV. The function automatically inspects the lifetime associated with the DSV value, and returns an error if the DSV value is invalid. Action taken when an invalid DSV value is encountered is application specific and is therefore left to the programmer to define.

dsv_close()

Function prototype: void dsv_close (DSV dsv_var)
This function is called by a process wishing to close a DSV or image DSV.

The primitives have been written in C and they are portable (we have ported them to three platforms). Provision of primitives such as these eliminates the need for the programmer of control applications to have any knowledge of systems programming, interprocess communication, or real-time programming.

4.2 Language extensibility

Low-level tasks which perform set-point control and direct digital control (DDC) functions can be developed. The IPC primitives defined above can be used to facilitate exchange of information between processes and to ensure correct real-time behaviour.

Set-point control and DDC fuctions are coded in C. The DSVs which the process owns and image DSVs which the process reads from define the interface between that process and other processes in the system. Set-point control and DDC processes are designed to be generic, so that they can be applied in any system. DSV names can be passed to the processes as parameters, allowing the processes to be written using local DSV names which are meaningful in the local context. Alternatively, as is done in one of our implementations, the C preprocessor and configuration files can be used to link the process's internal DSV names to DSVs in the system.)

4.2.1 Development of a low-level process

We will show how set-point control and DDC functions make use of the DSV IPC primitives by means of an example. The code given in this example is the implementation of an averager which continually averages three variables in the system and writes the result to a fourth variable. This function would be useful where multiple measurements of a single noisy property are being made by three sensors.

```
TASK(average1)
    {
    double stop, first_val, second_val,
              third_val, ave, valid_time;
    char dsv1[10], dsv2[10], dsv3[10],
              dsvaverage[10], cmdline[80];
    DSV c1, c2, c3, c4;

    GETCMDLINE(cmdline);
    sscanf(cmdline,"%s%s%s%s%lf", dsv1,
                  dsv2, dsv3, dsvaverage,
                   dsvstop, &valid_time);
    c1=dsv_ropen(dsv1);
    c2=dsv_ropen(dsv2);
    c3=dsv_ropen(dsv3);
    c4=dsv_wopen(dsvaverage);
    for(;;)
       {
       get_state(c1, &first_val);
       get_state(c2, &second_val);
       get_state(c3, &third_val);
       ave = (first_val + second_val +
                          third_val)/3;
       put_state(c4, &ave, valid_time);
       yield();
       }
    }
```

First a number of local variables are declared, including variables into which the processes places the DSV contents for local use (`first_val`, `second_val`, `third_val` and `ave`). DSV handle variables are also declared which are linked to DSVs by the `dsv_open()` and `dsv_close()` functions. The DSV handles are used in subsequent calls to the `put_state()` and `get_state()`. The

GETCOMMANDLINE() statement is used to accept command line parameters which include the names of DSVs in the system. The system DSVs specified in the command line are linked to local DSV names.[1] The function then obtains a value from each of the three source DSVs using the get_state() command. It then computes the average and writes it to the destination DSV using the put_state() command. The put_state() command assigns a validity time to the DSV.[2]

This process can be initialized using the process initialization macro INIT(). The syntax of this macro is INIT("TASK_NAME",task_name,parameter_list). Specific macros can then be written to initialize generic processes. The "average1" process above could, for example, be initialized using a command of the form doaverage("parameter_list"). This command would invoke the INIT macro.

4.2.2 Development of a higher-level process

Higher level control applications are programmed as a sequence of these macro statements. Macro statements which call generic processes have the following syntax:

```
dotaskname("image DSVs separated by spaces,
            owned DSVs separated by spaces,
            other process parameters")
```

Generic set-point control and DDC functions constitute process control objects (PCOs), and they are placed in the PCO library. PCOs from the library can be pieced together to form more complex PCOs, which may themselves be stored as higher level PCOs in the PCO library. The language is therefore horizontally and vertically extensible.

For example, a complete adaptive control system can be implemented as a combination of PCOs from the library.

```
dosample("period 6 600")
docounter("period")
dotaskmanage("")
dosysclock("")
doreadai("period level1 level2 level3 600")
doaverage("period level1 level2 level3
                                ave_level 600")
dofilter("period ave_level filt_lev 600")
dosynth("period filt_lev fil fig2 fig4 th2
     th4 mz ck1 ck2 ck3 ybar azone a 600")
doadaptcont("period filt_lev ck1 ck2 ck3 u
                        ubar mz uc 20 5 600")
doestimate("period azone a ybar ubar fil
                                fig2 fig4 th2 th4 600")
doscale("period u u_scale 1 0 600")
dograph("period u_scale filt_lev uc 12 12")
dowriteao("u_scale period)
```

Not all the processes specified in this statement list are essential in the adaptive controller implementation. dotaskmanage() provides a run-time command interpreter, dosysclock() provides an on screen calender and clock display, and dograph() provides graphical display of the controller performance. A macro statement doadaptcontsys("...,") could be defined to initiate all these processes and the complete adaptive control system could be added to the PCO library as a single PCO.

4.3 Intuitive programming

This language allows intuitive programming, because a programmer developing software at a particular level of control is not required to possess any skills which would be foreign to him. For example, programmers developing set-point control and DDC functions are not required to have any knowledge of low-level systems programming and interprocess communication.

Similarly, programmers of higher level optimizing control functions are not required to have any understanding of the internal workings of set-point control and DDC functions. Higher level functions are programs or configurations consisting of lower level functions.

4.4 Redundancy management and fault detection

Each *p_var* is owned by a single process, and the result of each measurement is maintained in a separate *p_var*. Where a number of *p_vars* represent redundant measurements of a single property, an *effective p_var* must be produced from some combination of the redundant *p_vars*. An arbitration or contention-handling process is responsible for producing the *effective p_var* from the redundant *p_vars*. Different redundancy management strategies can be implemented by the *arbitration process*.

Ideally, the arbitration process should not know how many redundant *p_vars* are in use in its particular subsystem. It should also not have to know the names of the individual redundant *p_vars*. It is possible to meet these requirements by using a suitable *p_var* naming scheme together with a special primitive for reading redundant *p_vars*. Each redundant measurement is thus represented in the system by a unique *p_var*. Redundant *p_vars* can be given the same name prefix, however each individual *p_var* must have a unique suffix. A primitive could then be implemented to read a number of redundant *p_vars* by specifying only the name prefix.

[1] The GETCOMMANDLINE() statement is only necessary in one of our implementations which uses a non-preemptive round-robin scheduler, and does not support the argv() and argc() functions.

[2] The yield() statement causes an explicit task-switch in our non pre-emptive round-robin scheduler.

For example, in some implementation an arbitration process may be responsible for producing an effective DSV temp from five redundant temperature measurements represented by temp1, temp2...temp5. A primitive get_redundant_state(temp*, values) could be developed to instantiate image DSVs of all temp DSVs in the system. temp* refers to all DSVs with the name prefix temp, whilst values is an array into which the contents of the DSVs temp* must be inserted. The arbitration process can then produce an effective DSV temp from the image DSVs temp* using some redundancy management technique.

This redundancy mechanism enhances failure detection facilities. Because each measurement is represented by a unique *p_var* in the system which indicates its temporal correctness, failures of redundant subsystems can be traced by the computer system. A failed subsystem's *p_var* contents will be invalidated by the passage of time. Because *p_vars* map directly to single subsystems, it is not difficult to identify the failed subsystem.

4.5 System verification

Computer control systems require extensive testing before they are commissioned. Often software verification is performed by simulation. The proposed language aids software verification because it allows the incorporation of real-time simulation PCOs. The control configuration given in section 4.2.2 can be verified by removing the statements doreadai() and dowriteao(), which link the control system to the plant via an analog input and analog output, and by inserting the statement dosimulate().

```
dosample("period 6 600")
docounter("period")
dotaskmanage("")
dosysclock("")
dosimulate("period u_scale level1 level2
                    level3 1.67 572 600")
doaverage("period level1 level2 level3
                          ave_level 600")
dofilter("period ave_level filt_lev 600")
dosynth("period filt_lev fi1 fig2 fig4 th2
     th4 mz ck1 ck2 ck3 ybar azone a 600")
doadaptcont("period filt_lev ck1 ck2 ck3 u
                    ubar mz uc 20 5 600")
doestimate("period azone a ybar ubar fi1
                  fig2 fig4 th2 th4 600")
doscale("period u u_scale 1 0 600")
dograph("period u_scale filt_lev uc 12 12")
```

4.6 CASE

Interactive and/or graphical "system configurators" could be developed to generate code similar to that given in section 4.2.2. It is therefore possible to extend to this programming language to the point where no actual programming is involved in the configuration of high level control strategies. Such strategies would be "configured" intuitively using an appropriate computer-aided software engineering tool.

5 Conclusion

This paper has examined the requirements which should be met by any language specifically aimed at process control programming applications. It was shown how many of these requirements can be met using the C language and primitives based on our *p_var* theoretical model of process interaction. The language addresses the programming of real-time distributed computer control systems at all levels. Real-time performance is achieved through the use of a specialized real-time interprocess communication mechanism (DSV) which forms the backbone of the language. Because provision of features for the management of time and distribution of information are intrinsic to the language, the programmer of a control application module need not be concerned with the communication of data to and from other modules. It was shown how this language facilitates intuitive programming of control systems, how it provides support for redundancy and failure detection, and how it supports real-time simulation for software verification.

6 Acknowlegements

We are grateful to the FRD and the University of the Witwatersrand for their support. We are also grateful to Shaun Zagnoev for his assistance.

References

[1] Kramer J., Magee J. and Sloman M. (1984) A Software Architecture for Distributed Computer Control Systems, *Automatica*, Vol. 20 No. 1

[2] Waller, S. (1990) Control Systems: The Key to Flexible Manufacturing, *Siemens Review*, No. 1 1990, pp 16-19

[3] Stankovic, J. A. (1988) A Serious Problem for Next Generation Systems, *IEEE Computer*, October 1988, pg 10

[4] Kopetz, H. et al (1989), MARS: A Maintainable Real-Time Distributed Computer Control System, *IEEE Micro*, pp 25-40

[5] Cohen, J. E. and MacLeod, I. M. (1991) Distribution of Real-Time Information in Computer Control Systems, Submitted Trans. SAIEE

[6] Zagnoev, S. (1991) The Development of a Modular Real-Time Process Control Language, *MSc Dissertation*, University of the Witwatersrand