DESIGN OF A MONITOR FOR THE DEBUGGING AND DEVELOPMENT OF MULTIPROCESSING PROCESS CONTROL SYSTEMS

A thesis presented
in partial fulfilment of the requirements
for the degree of
Master of Technology in Computing Technology
at
Massey University

GILLIAN DOBBIE

February 1987

ACKNOWLEDGEMENTS

There are a number of people I wish to thank for the parts they have played, either directly or indirectly, in the production of this thesis.

I would like to sincerely thank my supervisors, Dr. Tim Hesketh and Dr. Bob Chaplin. Tim developed the RTS process control system and hence provided the incentive for the production of this thesis. Bob guided me through the masterate, offering much welcome criticism where necessary, throughout all the stages of the work reported in this thesis.

I am also extremely grateful to the other members of the Production Technology Department and to members of the Computer Science Department. They showed interest, gave support and listened. My particular thanks goes to Paul Lyons whose constructive criticism was invaluable in shaping this thesis.

Finally I would like to take this opportunity to thank my family and friends whose encouragement and support throughout my years at Massey University provided much incentive towards the completion of this thesis.

CONTENTS

List of Figures

ABSTRACT

This thesis describes the design of a general purpose tool for debugging and developing multimicroprocessor process control systems. With the decreasing price of computers, multimicroprocessors are increasingly being used for process control. However, the lack of published information on multiprocessing systems and distributed systems has meant that methodologies and tools for debugging and developing such systems have been slow to develop. The monitor designed here is system independent, a considerable advantage over other such tools that are currently available.

Chapter 1

Introduction

This thesis is divided into five chapters. This first chapter is the introduction, giving the background of the work to be done.
Chapter Two considers tools for debugging multiprocessing systems and looks at existing multiprocessing process control systems.
Chapter Three outlines multiprocessing considerations and decisions which were encountered in the design of the monitor.
Chapter Four outlines the monitor requirements and design specifications, giving a scenario of activities for which the monitor may be used.
Chapter Five descibes the implementation of the monitor. It outlines the hardware and software used and how they fit together.
Finally, Chapter Six contains the conclusions and a proposal for furthur research which has stemmed from this thesis.

1.1 Process Controllers

Digital industrial controllers, with control functions realised in software are now common place. They must meet stringent reliability and availability specifications. This has led to the design of real-time systems with software designed with the requirements of reliability, safety and management in mind.

However a control system encompasses much more than the process control functions themselves. Indeed it can be argued that the following support functions:

- process interface,
- user interface,
- alarm monitoring,
- data logging,
- start-up and shut-down,
- sequence control

are as important as the controller. Certainly they are inordinate consumers of computing resources.

The traditional approach to servicing the above functions has been to use a bigger and faster monoprocessor computer, but with the advent of low cost powerful microcomputers the multiprocessor solution can be economically explored. Many different machine architectures and software packages [3,8,24,25,26,30,39,42,47] have been proposed. The difficulties of deciding which structures are optimal or even suitable for a given control system are not only caused by implementation costs but also by the lack of suitable tools for fault tracing and performance assessment in multiprocessor systems.

RTS, a small digital controller will be used to illustrate the associated problems, and later to assist in the design and testing of a multiprocessor.

1.2 RTS

RTS (real-time system) is a real-time process control system which was designed and the program written by Dr. T. Hesketh [19]. It has been used in a number of practical applications including controlling the atmosphere in the climate rooms at the DSIR and the drum temperature at a wool scourer. The program runs on a single Z80 processor and is written mainly in PL/I with some routines in Z80 assembly language. It has been written so that it is relatively easy to alter.

RTS is a general purpose process controller which may be set up in a specialised configuration when it is to control a specific situation. It assumes multi-independent single variable loops. These are loops where a given manipulated variable affects only its own controlled variable and there is no interaction between the loops. RTS implements three of the five functions mentioned above:

- user interface function,
- process interface function,
- control interface function.

The other functions were not incorporated because they would overload the processor. An 8-bit microprocessor has a limited capability and so priorities must be given to the activities which are considered for implementation. The control function has the highest priority and the data logging the least.

Start-up and shut-down actions do not occur automatically in RTS. The sequences can be carried out manually by the operator. This is possible in this situation as the operator using the system would be skilled. This gives the operator more flexibility in the operation of the process.

1.2.1 User Interface Function

The user interface function has been minimised to reduce processor loading. This function in RTS includes:

- menu display,
- process, state and set-up displays,
- user responses,
- plotting and printing.

1.2.2 Process Interface Function

The process interface function includes:

- A/D and D/A operations,
- filtering operations,
- alarm monitoring.

1.2.3 Control Interface Function

The control interface function incorporates the following five sections:

- measurement of the process behaviour,
- determination of the model,
- design of the controller,
- subsequent definition of feedback gains by the controller,

- use of feedback gains to obtain a control signal.

Measurement of the Process Behaviour

In this phase, system-specific information must be gathered. This information comprises:

- sampling and control time intervals,
- the signals and their interconnections (loops).

There are three different time intervals:

- the interrupt time interval (usually in the order of milli-seconds),
- measurement sample rate, set by the user,
- actuation sample rate, also set by the user.

A signal in RTS is a data structure for the definition of external variables i.e. set-point, measurement, actuation. Loops are hierarchically superior data structures which are incorporated in the system to associate signals in groups.

Determination of the Model

The system identification involves a recursive procedure whereby the data which becomes available at the beginning of each sampling interval are used to improve the previous estimate of the system model. The technique of extended least squares [2] is used to determine the model and a Kalman Filter [23] is used to obtain parameter estimates also recursively.

The plant can be operated at user-selected operating points and have pseudo-random disturbances created by RTS. The subsequent measurements are used in the system identification to estimate model parameters.

Design of Controller

In RTS there are a number of different controller design methods used. They are designed for both on- and off-line operation and include the following controller designs:

- optimal controller,
- self-tuning controller,
- user defined controller,
- three term controller.

The design of an optimal controller uses the Ricatti Equation [27]. To determine the gains of the system it is necessary to iteratively solve an initial value problem for a Ricatti Equation.

The self-tuning controller works well but as it operates on-line, it can begin to model the noise if the system remains in this mode once the system has settled.

The user defined controller allows the user to define and design a controller.

It is necessary to consider the categories of variables which exist.

1.3 Variables in Process Control

There are four important categories of variables related to process control [41]:

- manipulated variables,
- disturbances,
- controlled variables,
- intermediates.

The values of manipulated variables can be adjusted by the control system. Examples include input raw material flow rate, steam pressure etc.

Variables whose values affect the operation of the process but are not subject to adjustment via the control system are known as disturbances (e.g. composition of raw material, ambient air temperature etc.)

Control variables are those whose values measure the performance of the plant, and as such are those which the control system must keep at some set-point (e.g. production rate, production quality etc.)

The intermediate variables appear at some intermediate stage in the process and can not normally be measured.

The general control problem is to adjust the manipulated variables so as to maintain the controlled variables at their set-point in the face of disturbances. The set-point is the value at which the user wants the control variables to operate.

The intermediates may be used advantageously, if they can be measured, in determining what control action should be taken. One major problem with process plants is the difficulty in deriving a mathematical model of the process from its characteristics. The characteristics depend firstly on the level of the plant operation (the plant often cannot be descibed using a linear model) and secondly, even under constant operating conditions the plant's characteristics change with time. The abilities of a computer make it attractive in this situation where it can collect large quantities of data, analyse it and make logical decisions based on the results.

1.4 Real-Time Programming

The term "real-time" has been described as "any information processing activity or system which has to respond to externally generated input stimuli with a finite and specifiable delay" [33].

For real-time process control systems the delay must not only be specifiable, it must also be constant as the theory of the identification and controller design is based on the assumption of regular measurement and actuation.

Software for real-time process control systems must respond to "clock tick" interrupts, user requests and changes in the process, so there are time critical components. The software must interact with the dynamic properties of the industrial system and must react to stochastically occurring events.

There are two different types of events:

1. events which must be serviced whenever a time interval boundary is reached,
2. events which are serviced as and when required.

With both types of event, the control software must react within a certain time and parallel tasks must also be synchronised within the software system.

1.5 Multiprocessing vs Multiprogramming

To alleviate the problems in RTS and indeed any other processor bound process control systems, it is possible to implement the system as either a multiprogramming system or a multiprocessing system.

A multiprogramming system appears to execute many tasks at the same time. However there is always only ever one set of instructions being executed at a particular time. A multiprogramming system time-slices, devoting a set amount of time to all the processes that are currently being run. In the case of RTS, the use of multiprogramming would not increase the speed of operation. RTS cannot have other functions added as it is presently processor bound (section 1.2).

Multiprogramming provides no relief from this situation as the implementation of multiprogramming would take up more processing power, and hence make RTS slower. In real-time process control systems, the interrupts must be serviced immediately. Interrupt systems like this would complicate a multiprogramming operating system and the software would be difficult to write.

The definition of multiprocessing differs from paper to paper. In many books only tightly coupled systems (section 3.6) are considered multiprocessing systems. In this thesis, a broader interpretation will be used, in which a multiprocessing system consists of both tightly and loosely coupled systems. A multiprocessing system consists of two or more processors linked together to perform one function. It was thought that RTS would run more efficiently as a multiprocessing system as it is presently processor bound, and there is a lot of code that could be executed in parallel, hence saving time. RTS can be divided into four functional subunits which could be executing at the same time. They are:

- the user interface function,
- the processor interface function,
- the control interface function where this could be implemented as an identification function and a control function.

1.6 Objectives of this Project

As mentioned previously (section 1.1), the advent of microprocessors has generated interest in alternative, multiprocessor-based methods of providing process control functions. The main objective of this project has been to design a monitor (a tool for debugging systems) for use in the development of such multiprocessor-based systems. This monitor was developed for the conversion of RTS from a single processor to a multiprocessor system. Every effort has been made to keep the monitor as general as possible. To this end, as few assumptions as possible about the nature of the multiprocessing system have been made. Any design decision that was made was considered from a general view-point and if a decision could not be reached then the logic and the structure of RTS was taken into account in reaching the decision.

Chapter 2

Current Research

This chapter covers some of the tools which are being used for the development and testing of multiprocessing systems and gives the reasons why a monitor was chosen for this task in this design exercise. It continues with the background information pertaining to multiprocessing control systems and a description of some multiprocessing process control systems which have already been developed.

2.1 Testing Multiprocessor Products

There are a number of possible ways to test and develop microprocessor based systems. They include analytical modelling, simulation modelling, system prototyping or using some kind of tool or testbed. The testbed approach is closely related to prototyping, since they both involve modelling the specific system being investigated. However the testbed approach can be used as a universal tool whereas prototyping is a one-off method.

Traditionally, a mathematical or statistical model was used to represent a process. The model would be run on a different computer from the one being designed and the output would be a listing of the computer code, a document describing the computer code, and a set of results that exemplified the solution to the problem being considered. This method was expensive in terms of both time and money, and was inflexible

to change. In comparison the testbed approach ensures performance is exact because the final system can use functionally equivalent hardware which is ideal for real-time systems, and while the software is varied in testing, it will be finalised to real system software [6].

The tools designed for single processor systems (often referred to as development systems) can be general purpose or universal investigative aids that support several microprocessor types, or dedicated aids restricted specifically to a single microprocessor type of family. Such systems can assist the designer in evaluating alternative hardware or software prototypes. Simultaneous testing of hardware and software produces powerful debugging capabilities. In-circuit emulators provide the most accurate method for testing and checking microcomputer systems hardware and software. They replace the central processing unit in the system under test and provide the ability to examine, change or modify CPU registers and storage memory. The monitoring of the program execution permits rapid and easy hardware and software testing [14].

The above applies equally to multiprocessing systems. However the tools designed for systems which have only one processor usually do not completely fulfil the needs of systems which have many processors working together.

A tool for a multiprocessor system is more complicated than one for a single processor system because it must test for the synchronisation of processors, for communication between processors and must record the data on traffic and the utilisation of any of the processors.

The software and hardware of each individual processor can be tested using methods used for single processors. However we are interested in the next stage of the testing, when the single processors are connected to form a multiprocessor system.

Berg [4] says "Many inherent problems need to be overcome before the benefits of distributed systems can be realised....Among the causes of difficulties with distributed processing are the lack of experience with, and data about, distributed systems and a lack of appropriate methodologies and tools for designing distributed systems and their application".

2.2 Summary of Multiprocessor Test Systems

Below is a summary of some of the tools which have been developed for testing multiprocessor systems.

2.2.1 Modular Multiprocessor System Design

Hirschman, Ali and Swan [20] devised a structural approach to multimicroprocessor system design. This approach uses computer modules based on the Texas Instruments TMS 9900

microprocessor as building blocks to configure systems. It can handle a range of microprocessor products, from limited standalone controllers, through loosely coupled networks to closely coupled multiprocessors.

The principle benefits of this approach are modularity, expandability, software simplification and low cost.

There are three limitations to this system. The first is the use of a single interprocesssor bus. This can constrain the volume of communication traffic that can be supported. To prevent impaired system performance, each processor's executable code must have a high degree of locality, requiring infrequent access to the interprocessor bus. The second limitation is that the rigidity of this system confines choices of the designer of the multiprocessing system and therefore the flexibility of the system being designed. The product must follow the principle of "circuit switching", rather than "packet switching" [43]; it must use specified hardware, and use the bus arbitration system specified. Finally it is also designed specifically for designing a product and not for testing the product once it has been designed, so it does not collect any data. If data was collected it could be analysed to estimate the best performance of the system and this often helps the designer make design decisions.

2.2.2 Concert

Halstead, Anderson, Osbourne and Sterling [17] describe a system called Concert, which is a shared memory multiprocessor testbed. It is intended to facilitate experimentation with parallel programs and programming languages. The designers did not design it as a direct prototype for highly parallel computers of the future. They designed it as a tool to teach enough about parallel processing to specify future designs. Concert is not designed for microprocessors. However, a lot can be learnt from the principles used in designing tools for larger systems and applying the same principles where applicable to tools for microprocessor products.

Concert was designed with a ringbus network. The advantages of this are its suitability as an interconnection medium that has adequate bandwidth and does not require high clock speeds or present other technical difficulties that might have made it riskier to build and debug.

Another decision the designers made was to use shared memory as the principle medium for interprocessor communication. This presents difficulties in the queueing for accessing memory, deadlocks, and ensuring the processors get the most current information.

To reduce design and construction time, "off the shelf" commercially available components were used wherever possible.

2.2.3 The Cm* Testbed

This testbed has been designed for testing a specific system, which consists of minicomputers. The Cm* testbed has taken a large part in determining whether multiprocessor architectures in general have an important role to play in providing computational power.

In writing about the Cm* testbed Gehringer, Jones, and Segall [13] indicate that a distributed system can be considered to have two components - an architectural component and a behavioural component. The first consists of hardware, firmware and software elements, and the relationship between them. The architectural component should be flexible. The second is characterised by the way the architecture acts in the presence of a workload. The behavioural component should provide controllable and measurable behaviour.

The architectural and behavioural components are incorporated in the Cm* testbed which has a programmable interconnection network for hardware architecture flexibility. The software includes the operating systems - StarOS and Medusa - they provide adaptable mechanisms and policies for running experiments with applications programs.

The behavioural component, the workload, the measurement tools and the experimentation control are integrated into an experimental environment, and they complement the other support programs for specifying, monitoring and analysing experiments.

There are three hook processors in the testbed which are special microcomputers. They run a debugging tool called KDP which can load, start, stop and single step a Kmap and examine its registers or RAMs. There is a Kmap per cluster, through which the clusters communicate. With KDP, microcode can be debugged on the Cm* itself - this is important as microcode bugs are often timing dependent and cannot be recreated.

The diagnostic processor is a PDP 11/10. It runs diagnostic programs on processors which are not used and maintains an error log.

The operating systems contain sensors to allow monitoring, and the utilities to run as user programs on top of an operating system. The top level of the experimental environment consists of three components, a schema manager, a monitor and a workload generator, which allow speedy construction of synthetic tasks to exercise various portions of the multiprocessor.

The Cm* testbed has allowed approaches which were heuristic to provide a basis for more extensive and rigorous investigation.

2.2.4 Rochester's Intelligent Gateway [28]

While developing this system Lantz, Gradischnig, Feldman and Rashid had no separate debugging tool as such. RIG supports Mantis which is a reasonably sophisticated system-level debugger designed to debug a stand-alone monoprocessor system. Using Mantis the system must swap from its present mode to debug mode so the entire RIG system on the processor with the debugger is suspended. It does however allow the user sitting at the system console to examine the state of all internal data structures of the RIG kernel and all processes using symbolic traces.

There have been changes made to the kernel and Mantis so that an Alto can act as a systems console for more than one RIG host simultaneously. The advantages of this are:

- that manual breaks to Mantis can be broadcast to all systems involved in the distributed activity,
- a software break on one system is immediately detected by the Alto, which then simulates a manual break for the other systems involved.

Process-level debugging remains a hit-and-miss proposition, relying primarily on the use of multiple virtual terminals and a user-level message switcher.

Realising that distributed operating systems are complex, and numerous logic and design problems must be solved during development, the designers of RIG incorporated a monitor. By monitoring the communication between processes, the kernel(s) can propagate the concurrency potential of processes directly involved with active system components, such as a file system process or a network server, to processes communicating with them directly or indirectly. The concurrency potential of a process shows its ability or likelihood of starting the activity of an active system component, defined as any device capable of acting without the attention of the CPU. The data collected enables the kernel to maintain a directed graph of processes that are blocking each other. This helps in making good replacement and scheduling policy decisions.

The monitor consists of two parts, a collector in the RIG kernel, and a supervisor residing on an Alto.

The collector records the event trace in a circular buffer. When the buffer becomes full the data is transmitted to the supervisor. When set up just to collect data without transmitting it, it is a useful tool for debugging.

The supervisor can compress the data in real-time and can collect some or all the desired statistics, or it can modify the tracing process dynamically, depending on the recorded events without interrupting the system.

From the data collected it should be possible to derive the following information:

- the state of any process at any time,
- the time when state changes occur,
- the state of all the system queues,
- the cause of state changes (particularly the reason for the suspension of a process),
- the scheduling sequence,
- the state of all the active components in the system,
- the swapping behaviour,
- the time spent in system mode, and the time spent in user mode,
- the messages being exchanged.

2.2.5 The μ* Multimicroprocessor

This system was built to aid in the setting up of tools and facilities for software development. It has a separate software development station which has the usual development tools for a microprocessor based machine (editors, assemblers, compilers, linkers etc.).

In this system the debugging was divided into the two phases, debugging a single task at monoprocessor level and debugging the cooperation of the tasks.

The first phase was carried out as usual and the second by a distributed set of system processes (DMS), that centralise the debugging activity of all user software to a unique working station. The user sees the process as a process similar to the usual multitasking system so it is irrelevant which processor a task is running on.
The functions of the system monitor are expanded to allow for monitoring of the process and debugging. The tasks are identified by name and number. This allows duplicated tasks on different processors, all with the same name. (This could be useful for system tasks).

The debug firmware has been built with two different layers which communicate with each other:

- the Display and Command Monitor (DCM), which interfaces interactively with the operator and consists of a single task running on a processor,
- the local monitors (LM). They are the lower level, residing on each processor.

The main tasks of the DCM are:

- to edit and parse operator commands, to format and send them to the destination environment (Local Monitors) for execution. The DCM therefore acts as a filter to the LMs.
- to receive messages from any task in the system (particularly from the LMs) and to display them on the CRT screen.

The LM always sends an answer back to DCM to synchronise its activities. The answer contains the acknowledgement of the message received, the information requested, and error flags.

The commands available to the operator include :

LOAD, RUN, SUSPEND, DELETE (cancels a tasks activation record), BREAKPOINT, DISPLAY REGISTERS, STATUS, DISPLAY MEMORY, SET MEMORY, SEND (sends a message to any processor in the system).

2.3 Simulation vs. Emulation vs. Monitor

In the above section some of the methods currently used for testing multiprocessor systems have been reviewed. It is now necessary to ask why a monitor has been chosen as the most suitable method for developing and testing a multiprocessing process control system. This is explained in this section with specific references to RTS. The three main practical test procedures would be simulation, emulation or the use of a monitor. Once the target system is completed, the test method should allow the comparison of different hardware and software combinations in the target system. The performance of a certain architecture is very much program dependent and can vary widely from one algorithm to another.

There are various multiprocessing simulation packages available - SIMON [11] and ALGAN [45] are two examples. Both of these simulation packages expect the system to be modified to fit into their philosophy. It is necessary to use the programming language provided and hardware that can run the provided software.

The benefits of simulation include improved or less costly design procedures for plant and machinery of various kinds, increased understanding of the problem, validation of hypotheses and educational benefits [7].

As the multiprocessing hardware is available, it is not going to be more costly to introduce a model system. RTS is a working system on a single processor thus the initial problem has already been understood and this has validated the hypotheses posed by RTS.

If this project was simulated the most effective method would be to run the simulation on the Cromemco, which was already available, so a special purpose simulation language would have to be written. The question must now be asked "Is it worth simulating RTS as a multiprocessing system on a monoprocessor?". The gains would be the understanding of the scheduling and synchronising of processes, understanding some of the timing problems, and the comparison of some of the hardware and software configurations. One disadvantage would be the time necessary to set up such a simulation. Also the final simulation would be quite different from the final system, and the resources that are available (i.e. hardware and

software) would not be used. So it would be very costly in terms of both computing and human resources.

Another method of testing a system is using emulation. Emulation is the replacement of a processor with a processor which implements the same instruction set and architecture as that which is replaced. The new processor can be linked to a host system, allowing the user to control the operation and the necessary signals of the target system. This allows the debugging and testing of software to be done in real-time. The target system is remote from the emulator allowing debugging in-situ on the target with full control from the emulator. An additional benefit is that many facilities of the emulator are now available to the target hardware. All the gains from the simulation apply here - the scheduling, synchronisation and timing are more easily understood as the process is running in real-time. If the programs can be down-loaded from the emulator and the connections between the hardware are not fixed, the comparisons of configurations can still be performed. The disadvantages of the simulation are also overcome as the existing hardware and software are being used. Emulation incorporates the final system, decreasing the time necessary to set up such an implementation.

Unfortunately problems of scale are difficult to study using either simulation or emulation. No emulation system can be large enough to contain all the situations of interest for test and analysis.

The final choice, the monitor, has the advantages of the other methods and also overcomes their disadvantages. A monitor is a separate processor which is included in the system as an observer without altering the target system. It allows the use of the most realistic data possible, and the impact of day-to-day events can be ascertained. It can be used as a development, test and debugging tool throughout all the stages of the implementation of RTS as a multiprocessor system.

2.4 Multiprocessing Control Systems

Computers have been used in process control activities for a long time (since the early 1960s [26]). Initially mainframes were used, but because of their cost, only large plants with hundreds of control loops could afford them. Mainframes were later replaced by general purpose minicomputers. Many of the techniques developed for mainframes were passed on to the development of minicomputer process control systems.

The use of minicomputers resulted in a single complex software program resident in main memory [24]. One of the general architectures for a minicomputer system is illustrated in Fig. 1. The minicomputer uses either interrupts or polling to service the I/O devices. Polling increases the I/O servicing latency period, however polling software is generally easier to write than interrupt servicing software.

There are still three areas where improvements could be made in the minicomputer system:

- The cost could decrease. Although such systems are cheaper than systems involving a mainframe, the cost of process control systems can be decreased further.
- The system could be more fault tolerant. When the computer breaks down, control of the process is lost. The second improvement is to stop this occurring.
- Finally the scanning and update rate can also be improved. In this system the computer has to access each loop and recompute its output signal.

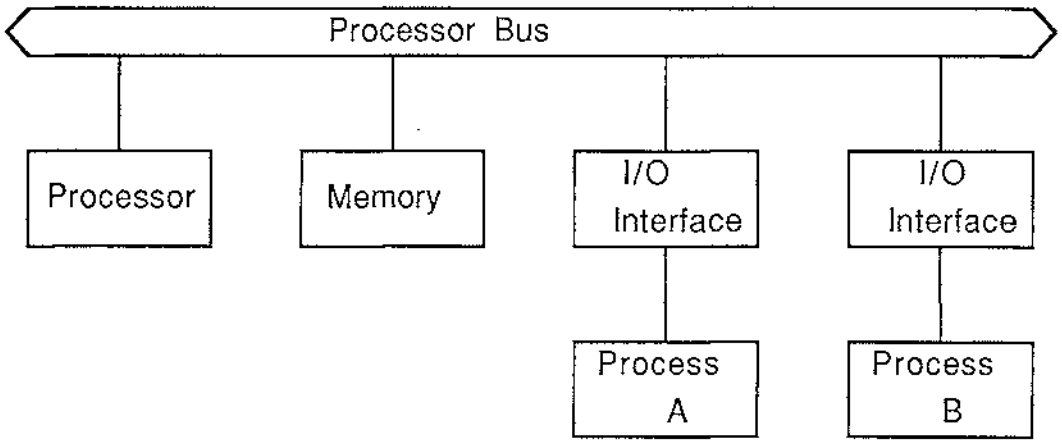


Fig. 1 General Architecture of a Minicomputer Process Control System

A system which overcomes the problems in minicomputer systems outlined above, is the multimicroprocessor system. The microprocessors are generally cheaper and more flexible. The system is more fault tolerant, because if one processor breaks down, another processor can take over for it or the rest of the system can continue without that processor. Each processor can be scanning at the same time, so if they are connected as illustrated in Fig. 2, each processor can do the input and output for the loop they are controlling.

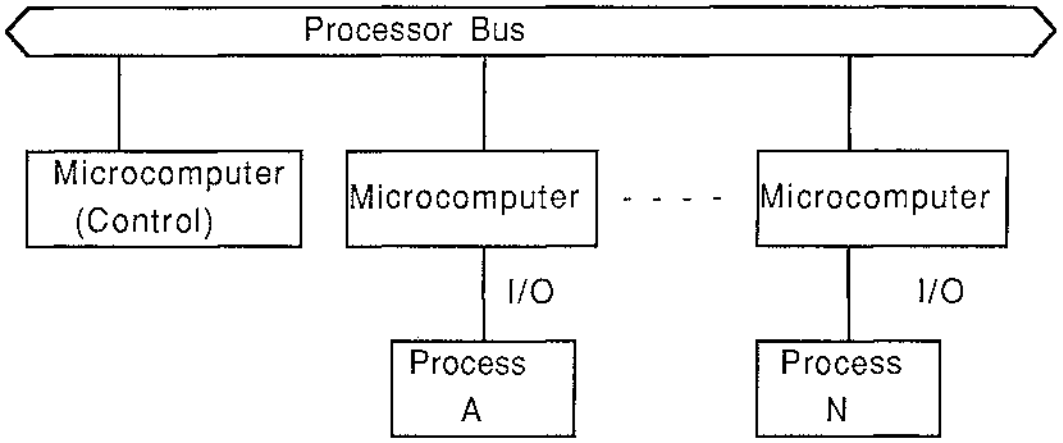


Fig. 2 An Architecture of a Multimicroprocessor Process Control System

This multimicroprocessor process control system has a microprocessor to process information from each I/O unit. The partitioning leads to the concept of a "smart" controller. This is only one type of architecture which is available for multiprocessor based process control systems. There would need to be interaction between the controllers for the loops for multivariable control so this system works only for single loop processes.

There are five fundamental differences between the two systems illustrated above:

- The processor bus in the multimicroprocessor based system can be run at a lower data rate than in the minicomputer based system. The processor memory bus need not be the same as the bus between the processor

and the peripheral devices, and therefore the electrical interfacing problem becomes simpler.

- In the minicomputer system the entire software for the system is concentrated, resident in a single memory. In the multimicroprocessor system the software is distributed i.e. each microprocessor memory contains software that is customised to that part of the system.
- The single high speed bus in the minicomputer is replaced by several lower speed buses in the multimicroprocessor system. This physical partitioning of the system into self-contained modules, which communicate at low speed, serves to simplify the bus design.
- The multimicroprocessor can be adapted to a variety of bus configurations and process interfaces.
- System software development is simplified due to the modular and standalone nature of the microprocessors.

2.5 Advantages of Multiprocessing

Multiprocessing is suitable when there are a large number of processes which can be executed at the same time or want to use the same software or a copy of it. In the RTS control system there are a large number of processes which want to execute the same software. This software involves a lot of sequential action. Modern design theory allows designing to be done on-line which demands a lot of computation quickly and one of the cheapest methods for servicing this need is multiprocessing.

2.6 Summary of Multiprocessing Process Control Systems

This section contains the description of seven multiprocessing process control systems.

2.6.1 NARCIS Project

Vernel, Lagier and Husson [47] felt there were inadequacies in the traditional minicomputers for real-time process control applications so they undertook a project called N.A.R.C.I.S. (Nouvelle Architecture de Calculateur Industriel Specialise) in the early 1970s. They felt the inadequacies lay in both the hardware and software. The inadequacies in the hardware were the lack of reliability, speed and capacity for expansion.

The software inadequacies were the inflexibility of the real time monitors and the lack of a real time language.

They decided to build a process control system which was reliable, fast and modular.

To realise these features NARCIS was designed as a multicomputer system with the hardware based on modules that are physically identical in order to simplify reconfiguration processes. Each module has a microprocessor, a ROM (to store programs), and a working memory.

The modules differ in their software. There are two different types:

- IOP or input-output processor for i/o operations.
- TP or treatment processor for computing operations.

The IOPs do the data logging, send commands, translate code and can monitor the process when necessary. The TPs take the data collected in the plant and treat according to control laws.

Any necessary exchanges between the different processors takes place through a common memory.
There is a control unit which is responsible for allocating the tasks the processors must perform.

The software can be thought of as two separate levels, the software used to control the computer, and the software used to control the application.
The first of these is the operating system, responsible for scheduling various other portions of the software system and maintaining communications between the various programs in use.
The second is divided into three levels which are characterised by the time requirements to respond to outside events. These levels are:

- data logging and the sending of commands,
- computing of the set-point control,
- adaptive control or optimisation algorithms.

Within NARCIS the aims of the project were fulfilled. The reliability is satisfied by the number of redundant modules in the system. There is also an automatic maintenance system (A.M.S.) which consists of a combination of failure detectors distributed across the various operation levels of the system. The failure detection no longer compromises the development of a program so the A.M.S. contributes to increasing the processing speed. Also distinct operations can be carried out simultaneously on various processors.
The use of the treatment processor (TP) and input-output processor (IOP) increase the modularity and give the potential to enlarge the system according to need.

2.6.2 A Distributed Computer Control System

The system described in this paper [25] was designed around an existing plant control system. The proposed multiprocessor system fitted naturally into an hierarchical structure.
In the first implementation interfaces such as the operator desks and their existing meters and auto/manual stations were retained. Another constraint was to utilise the existing plant-mounted transducers and actuators. The existing system was a control system for a 120MW boiler. The areas that were controlled were:

- feedwater control,
- superheater temperature control,
- reheater temperature control,

- furnace pressure control,
- air control,
- mill control,
- steam pressure control.

To control these areas, five subsystems were chosen:

- feedwater control,
- temperature control,
- air control,
- coal firing control,
- steam pressure control.

Each subsystem remains at a steady state when the interactions of the subsystems are being ignored. However it is known that there is some interaction over the full-load range so a management supervisory processor was included to investigate and define some of the higher level control functions.

Between the subsystem machines, and between the subsystem machine and the management machine, a modified form of the BISYNC data communications protocol [43] was used on 20mA [29,33] serial links. Data transfer between subsystem processors is routed through and handled by the management processor.

The management processor carries out such functions as data logging, operator display and supervisory functions. A disk is provided for medium term data storage for the display and alarm functions. Down-line loading to the subsystem machines is also provided through the management processor and a copy of the schemes in the subsystem processors is held on disc. In the event of a failure in the subsystem machine, automatic program reloading may be achieved.

Commercially available equipment was used to keep hardware development to a minimum. MEDIA plant interface equipment was used along with PDP11-03 microcomputers for the subsystems, and PDP11-34 for the management system. Most of the computers were placed in a single room.

The software for the system is written in CORAL 66. Design is such that any failure in communications is fail-safe and minimised.

The main design features included:

- provision for all common modulating control functions e.g. PID controllers etc.,
- provision for selectable control strategies in response to plant state,
- provision for inter-control loop communication when dealing with interactive control situations,

- communication facilities to other machines for distributed and hierarchical control systems,
- capability for efficiency or other plant performance driven control via communication with logging/display machines,
- smooth manual/auto transfers and control schemes switching,
- automatic loop start sequencing for cascade systems,
- automatic prevention of integral wind-up and related problems due to valve endstop or other constraints,
- built-in error checking and fail-safe response for communication link or local plant input/output failures,
- watchdog backup check for processor or other basic hardware failure,
- simple to use control language to set the system up providing on site configuration,
- security features for controlling access for controller settings, set-points etc.

The conclusions that were reached were that the control system of a large power generator unit, which this was, can be broken down into a distributed form using five subunits and that considerable advantages in terms of higher security and improved control performance could be achieved by adopting this distributed hierarchical structure.

2.6.3 A Hydrological Monitoring System

The hydrological data monitoring system in this paper [39] consists of clusters. The clusters are interconnected with the RWS packet-switched-data communication network in combination with public data communication services. This allows for distributed processing and distributed data storage.

The design of the clusters makes distributed processing possible. A cluster consists of a number n (where $4 < n < 50$) of monoprocessor microcomputers interconnected with the data communication system. Each microcomputer has an identical software data communications kernel and could in principle perform every existing function. However there are differences introduced with discs, plotters, sensors etc.

There are five basic types of processors:

- dictionary processor,
- library processor,
- executing processor,
- user processor,
- link processor.

There is one **dictionary processor** per cluster. It takes care of the allocation of processing power and of storage capacity. Every background storage medium will have a **library processor** to abstract its behaviour from the cluster. The distribution of files among library processors is taken care of by the dictionary processor.

Every processor without a process will execute without default its data communication kernel and can be loaded with any process. These empty processors are called **executing processors**. At start-up time all processors are of this type.
The **user processor** executes a high-level language interpreter to accept user commands from a terminal.
It is necessary to have a minimum of two **link processors** to interconnect two clusters via a packet-switched network. This processor abstracts the behaviour of the packet-switched network from the cluster. With abstraction the two clusters look like one. Only the inter-cluster messages have a time-delay.

A simulation was carried out based on the processors in the clusters being joined in a ring net with a twisted pair cable. All processors are connected to the bus through input/output processors (IOPs). The IOPs have enough memory to handle all incoming or outgoing traffic but they are limited in their accessability due to a restriction in the number of I/O ports. They keep track of the transmission control, time slot synchronisation and bus interfacing. A seventeen bit counter is synchronised by the first synchronisation character issued by the monitor on the first restart procedure, and provides the time slot counter with word counts. Transmissions are only acknowledged on a message base, so no handshaking takes place at the word level.

The simulation was carried out before implementing into hardware one of the ring network configurations. This simulation was naturally very specific to this system. The information to be obtained was:

- the relation between the ring response time and the number of time slots in the ring net,
- the relation between the connect time and the number of time slots in the ring net,
- the influence of the ring load and the connect time and ring response time,
- the influence of the number of I/O ports in each processor on the connect time and the response time and,
- the fact that it takes sometime before a processor can start the transmission of an output messsage can cause queueing of messages. The queue length was investigated for different configurations and ring loads.

The simulation program was written in Simula 67 and all activities were defined using Evaluation Nets (E-Nets) [35].
The results of the simulation showed that the introduction of time slots resulted in a deterioration of the connect time if the number of I/O ports in each processor was not increased at the same time. A double ring was introduced with two time slots and two I/O ports per processor, and this was preferred for reasons of connect times, throughput and reliability.

2.6.4 Modular Multiple Microprocessor System for Control Applications

This proposed modular multibus multiprocessor system [30] was designed to overcome resource contention and performance degradation, and also to take advantage of definitions in process control which slot naturally into the multiprocessing system.

There are three main parts to this system. They are the processor (P), the common memory (M), and the bus (B). Each processor has its own local memory (LM), I/O capabilities (I/O) and bus interface (BI). The common memory also has the I/O capabilities and bus interface. The bus module has arbitration logic (AL), which is used to manage the bus.

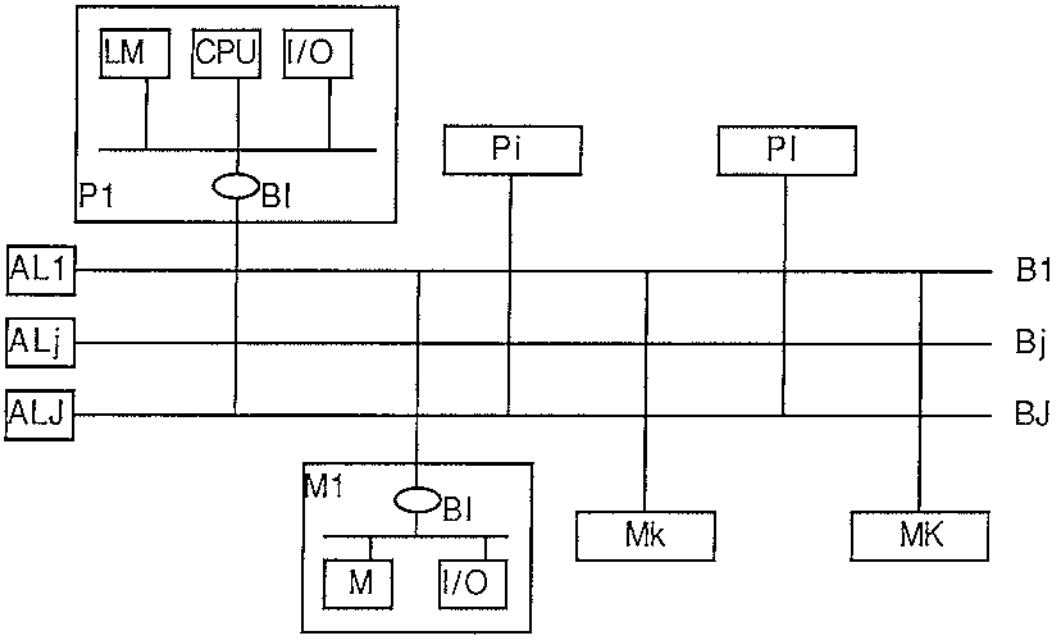


Fig. 3 The Structure of the Modular Multiple Microprocessor System

The choice of topology was based on the following factors:

- reliability,
- fault tolerance,
- modularity,
- bandwidth,
- number of processors,
- expandability,
- moderate cost.

The systems bus management and the basic synchronisation functions are supported by hardware. The multibus interconnection scheme is used in preference to the crossbar because it is less expensive and more fault tolerant. It also gives the same effective bandwidth with an adequate number of buses. Even though the arbiter required in the multibus system is more complex, it was shown to be cost-effective and fast.

The major feature of the system buses is that the software does not need to know how the buses are configured or how many of them are working.
Each microprocessor has a "window" to look at the common memory module. Due to the memory-mapped I/O, some common memory addresses can be used to support the common peripherals. The I/O subsystem is organised in two levels, local I/O and global I/O.

The local I/O involves the interfaces and devices which are directly associated with the processors. This direct association gives minimum response time for interrupt driven tasks. The global address region contains the physical addresses of the peripherals used for global I/O.

A concurrent programming scheme was used, based on "processes" and "monitors" [22]. As a process is a self-contained sequential program with its own data, it can access its own private data but cannot operate on data belonging to other processes. The monitor allows certain data structures to be shared.
A monitor defines a shared data structure and all the operations which can be performed on it. As well as transmitting data between processes, monitors can be used as synchronisation structures, enforcing exclusive access to shared data by executing one process at a time.

The system allows the allocation of a process to a processor to be fixed or dynamic. The management of communication between processors is simpler with fixed mapping.
In the prototype, the processes are given priorities based on their function and specification in the application. Factors affecting priority are:

- whether the processes are interrupt driven,
- if a bounded response time to process activation is expected etc.

The processes with top priority are stored in local memories and have their own processor. The other processes are stored either in local or global memory. The processes in global memory can be executed on any processor when activated. However they can be interrupted by a process with a higher priority. The global processes can be executed on processors with other processes in local memory or on processors without other processes in local memory. If some processors are set aside especially for global processes, the run-time-processor availability is increased.

The interconnection scheme is based on the common memory modules. A processor can use these common memory modules in one of two ways. The processor can either block the entire memory module for its own use or it can share the memory module with other processors. In the latter case, special hardware and/or software is needed to ensure that a processor is not attempting to access a part of the memory that it does not have access rights to.

The global operation is carried out in three steps:

- the common memory module is selected,
- the access form is selected,
- the request is executed.

The access form can be an I/O port request or an instruction fetch request. The I/O port request involves executing an output instruction for a special I/O port. The instruction fetch request is carried out during the instruction fetch cycle. If the

address of the instruction is in the range of the microprocessor window then a positive response is received. Once the request has been acknowledged, the desired module is selected. The access is in two parts:

- an arbiter is needed to make allocation decisions, when a simultaneous multiple request is made for a common memory module.
- the arbitration logic assigns the buses at request time. A common memory module access is achieved only if the selected module is available and there is a free bus.

The arbiter is used for arbitration of simultaneous multiple requests for common memory modules and also for bus assignment.

A modular system was designed and the prototype implemented. This paper explains the system architecture, the programming scheme, the interconnection structure and the arbitration scheme. Although the paper sometimes refers to the prototype, the system was designed as a general purpose system with control applications in mind.

2.6.5 MODUMAT 800

Like the other systems described above Modumat 800 is a distributed system for the control of industrial processes [3]. It consists of functional processors (e.g. control units and operator units) linked by a communications structure. The communication structure has one interface processor per functional processor and is connected by a serial bus.
Distribution is achieved at the levels of function and implementation.
In the implementation level, there are multiple processors at each function level.

There is a programmable multiloop regulator which is designed to handle from one to eight loops. The regulator is made up of three basic blocks:

- process interface module (PIM),
- multiloop processing unit (MPU),
- serial bus interface board (SBI).

There is one process interface module (PIM) per loop. It supports two main functions. The first function is to acquire data from the process, including sampling and A/D conversions. It also sends output values to the process and carries out the D/A conversions. The second function is to carry out fault detection and recovery when the MPU fails.

The MPU also has two main functions. It runs the regulation algorithms for the possible eight loops and their possible interactions. The MPU also handles exchanges with the PIMs and the Serial Bus Interface. The exchanges include requesting data and sending results.

The SBI is in charge of handling the communication on the bus which includes the bus allocation and low-level transmission protocols using an HDLC frame based mechanism. It is connected directly to the PIMs when a faulty MPU is disconnected. This makes it possible to directly access the data in the PIMs and transmit/receive it on the bus. The SBI is made up of four parts:

- a processor interface,
- a computing unit,
- a line interface,
- a board watchdog.

The processor interface receives the functional processor's requests and sends back corresponding messages.
A Z80 is used as a computing unit. Its main functions are firstly to analyse the messages received, prepare answers and transmit them; secondly to transmit/receive to/from the functional processor, to ensure that the compulsory logical transformations between the two interface parts of the SBI occur. The counter-timer chip is used for time out generation and clocking in general.

The line interface deals with the biphase encoding/decoding.
If not periodically retriggered by software, the board watchdog resets the board.

The operator unit is the man/machine interface. It allows the operator to select a given number of views of the industrial process data from a preestablished list. This is also where the operator commands are interpreted.
The operator unit includes a processor to implement the above functions, two mini floppy disks containing the coloured views, graphs or drawings and an SBI board connecting the unit to the other processors. At least two display units need to be connected : one in charge of the synthetic view and the other(s) to observe specific parts.

The communication hardware consists of the serial bus and the serial bus interface boards. The bus is a shielded twisted electrical pair. It is duplicated for reliability. Each functional processor is connected to an SBI board.
The software methodology used was based on the use of Communicating Sequential Processes (CSP) [21] and Petri nets [36].
For detection software there is a list of access capabilities, any attempt to make illegal accesses is refused. There is also a processor which acts as an observer. For each exchange on the bus, it verifies that in that particular partial state of the system, that exchange is allowed by the net.

Part of the communication scheme is located on each SBI and is known as the local communication scheme (LCS). Cooperation of the LCSs results in sharing of the bus resource between SBIs, based on a virtual ring. The SBI which is the primary station at a particular time can address any other SBI. When its primary station time is over, it relinquishes primary status to the next one on the ring. A boss/slave system is introduced in case of fault in an LCS.

Each LCS is the same, whatever processor is connected to the SBI. Its primary functions are:

- transmitting/receiving to/from the bus
- primary status handling
- access list handling
- message passing type cooperation between processes inside the SBI.

The LCS is based on a multi-tasking kernel, supporting a message passing synchronisation scheme. A process is either SENDing or WAITing, this is similar to the traditional P and V operations semaphores.

In the first implementation the master status is centralised in a single boss known as the controller. The controller constitutes the global system timing mechanism. The controller is duplicated. The second controller runs, controlling itself and the other one but its transmission circuits are turned off. If a fault occurs in the first controller the operator manually switches controller 1 off and controller 2 on. They aim to develop the protocols so that they are more

sophisticated than the send-and-wait one now existing. New protocols must be supported to encourage evolved communication and global distributed control.

2.6.6 Microcomputer Systems for Chemical Process Control

The system descibed here [8] uses a modular approach. This makes the construction of special interfaces and control circuits possible. It also enables custom-fitting of the microcomputer controller to a given process control application. The logic modules are computer elements assembled on printed circuit boards of a specified size which plug into a chassis and carry data and control signals in a parallel format between the modules. The memory addressing and input/output control are carried out on supporting electronic circuits. The module design approach makes possible the construction of custom made dedicated microcomputers with a minimum of circuit wiring and peripheral control logic design. Additional impetus for extending the modular design concept to peripheral functions is the engineering goal of adapting the microcomputer to a large number of speciality applications.

The basic 8-bit microcomputer system built around the Intel 8008 consists of a CPU and the supporting LSI circuit components on three standard printed circuit boards; the central processor board, the input/output board and the

memory address board. They supply all the timing, tri-state buffering and address decoding needed to support the operations of the basic microcomputer.

Another basic microcomputer system has been set up based on the Intel 8080 microprocessor as a two card module. The microprocessor module contains the Intel 8080 CPU chip and logic circuits for timing and control signal processing. The 8080 control module contains the clock generator and generates all timing, control and status signals for the 8080 microprocessor module, the RAM and the PROM.

The program push-pop stack module has been designed as a peripheral circuit to the 8008-based microcomputer. The module is used in complex multilevel software applications to store the state of the machine at a program breakpoint, and to hold the parameters for subroutine calls and returns. The dual-restart board functions as a manual restart of the microprocessors, or as an automatic restart at a stored location in memory. It is useful in software debugging and also in executing two-level priority interrupts. The A/D and D/A converters are 12-bit resistance ladder network devices with a sampling time of >= 25μs. The D/A converter has a decoder circuit on the module which sequentially stores the high-order and low-order 8 bits on a tristate latch connected to the data bus.

A general interface was developed to interface the microprocessor to different devices. Other special interfaces were also devised on a one-off type basis. It was proposed that the CAMAC [34] system, with its major international backing, could become the standard for remote data transmission.

The special purpose systems which were based on this system included:

- process monitoring,
- gas-chromatography control,
- electrochemical process control,
- general purpose control applications.

2.6.7 Distributed Hierarchical Computer System

At City University where this paper [28] was written there are several pilot-scale processes which are required to be operational simultaneously. Each pilot-scale process has its own local digital controller, which performs such tasks as data logging, parameter estimation and direct digital control. The local controllers independently transmit data to a minicomputer for processing and graphical display, and, in turn receive local control parameters. A star network is used as the only communication necessary is between the local controller and the minicomputer, rather than between two local controllers.

The simultaneity of the plant has been met by a time-shared operating system on the central computer. This limits the time

taken for an interrupt signal to be processed, but this was not a disadvantage on this system.

There are two levels, the supremal level which includes the central computer, and the infimal level which are the local controllers.

The infimal level contains four I-MIC (8085 based) microcomputers and a DEC LSI11/02 minicomputer, which are used (depending on the application, for data acquisition), direct digital control and local optimisation. A DEC LSI11/23 is used at the supremal level to coordinate the computers at the infimal level.

At the time the paper was written, the I-MICs controlled a pilot scale freon vaporiser, a mixing process and an analogue computer simulation of a two subsystem interconnected plant. The LSI11/02 controls a pilot scale travelling load furnace.

Peripherals include twin 20Mbyte Winchester discs and a 1.2Mbyte floppy disc. Others available at the supremal level include a colour graphics terminal and a graphics terminal with hard copy unit.

Each computer at the infimal level is interfaced to its respective plant using plug-in memory-mapped interface cards. Communication between the I-MICs and the LSI11/23 takes place over 20mA current loop serial lines at 1200bd, while communication between the LSI11/02 and LSI11/23 takes place over a 16 bit optically isolated parallel link.

The software for the I-MICs is mainly written in CONTROL BASIC, with a few routines like the link communication

routines written in 8085 machine code for speed reasons. The LSI11/02 is programmed in FORTRAN IV, using system software for I/O programming and interrupt handling. The LSI11/23 is programmed mainly in FORTRAN IV using software written in MACRO-11 to drive the colour graphics terminal. At the start of each communication the local controller transmits the length of the frame of data to be transmitted from itself to the supremal level. This is stored and the length of the frame of data to be transmitted from the supremal level to the local controller is transmitted back to the local controller. The rest of the data is then transferred. On the completion of the exchange of data a check is performed to ensure that a transmission error has not occurred.

The sampling time used for the freon vaporiser is four seconds, which is adequate in view of the slow process dynamics. At each sampling time the process variables are read, the control signals are calculated and the digital signal to the valve actuators is output. The conversational program running at the supremal level enables the operator to change process set-points etc. while the process is running, to select the mode of printing, i.e. VDU, line printer or file, to call the colour graphic display terminal to display the process state on a mimic diagram; to call for process optimization; and to request process shut-down.

Adaptive control was used and new iterative optimisation techniques were tried. The iterative optimisation techniques are resident at the supremal level and use measurements supplied by the local I-MIC controller to update process set-points to be automatically transferred to the local controller for implementation on the process.

The control software for the mixing process follows similar lines to that described for the vaporiser. Due to the interactions occurring between the control loops, the process has been seen to be a valuable vehicle as a demonstration of multi-variable control techniques. The non-interacting control method [9] and the Inverse Nyquist Array Method [40] have been found suitable.

A steady state model consisting of two interconnected subprocesses, simulated by an analogue computer, is used to investigate different coordination methods for closed loop hierarchical control such as interaction balance and interaction prediction methods with global and local feedback. An EAL Pace TR48 analogue computer is used to simulate the interconnected process. First order constants are introduced to the interaction inputs and controls within the simulated real process. Synchronisation and interprocess communication problems arise at the decision unit level, so this has also been tested. Two methods have been considered. They are the elapsed time method and the semaphore method.

Future possible developments are likely to include upgrading of the LSI11/23 to accommodate research in robotics and the

replacement of the local controllers by more powerful microcomputers to enable local optimisation to be performed.

Chapter 3

Design Considerations

This chapter summarises properties which multiprocessor systems exhibit [5]. These differing properties had to be considered when the monitor was designed. The properties include:

- partitioning of functions,
- interconnection topology,
- the bus,
- the structure of the processors,
- bus protocol,
- the coupling of the processors,
- the information needed in the frames.

3.1 Partitioning of Functions

Good partitions lead to good systems. The definition of the system is partitioned into processes. Good partitions are produced when there is very little overlap between the processes. When there is little overlap, there is only minimal data to be passed between the processors. The system is then more tolerant of faults, because the processors are more independent.

The allocation of processes to processors may be dedicated or dynamic. The dedicated approach simplifies the control software but decreases the flexibility of the system. The alternative is obviously more complex to control and

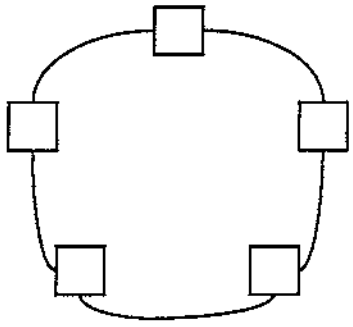
implement. However dynamic allocation increases the flexibility of the system.

3.2 Interconnection Topologies

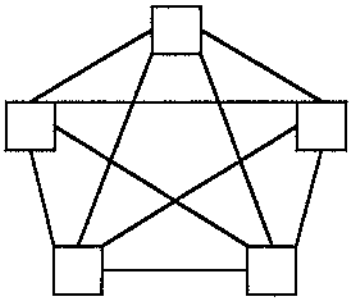
There are many interconnection topologies. Fig. 4 shows some topologies which are considered. The choice of topology used depends on the following features:

- performance bottlenecks,
- modularity,
- fault tolerance and reconfigurability,
- interconnection complexity.

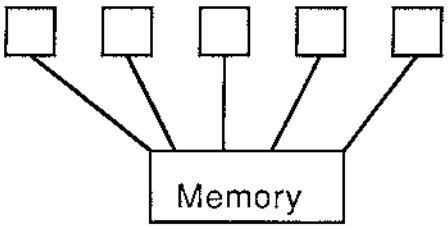
Not all these attributes are always important, however any interconnection schemes can be compared using this list as a reference.



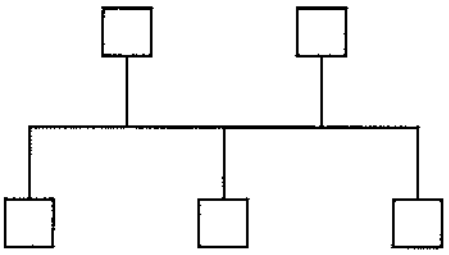
(a) Loop



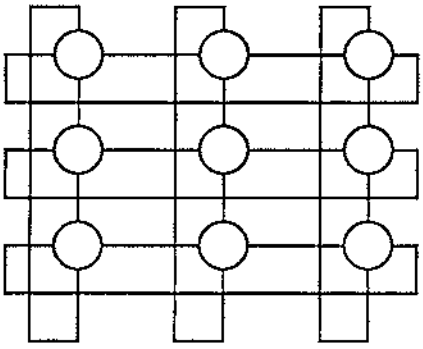
(b) Fully Interconnected



(c) Common Memory



(d) Multidrop



(e) Regular Network

Fig. 4 Interconnection Topologies

3.3 The Bus

The paths between the nodes can be implemented in many different ways. Along these paths pass all the data and control

information. These paths are the buses in the system and the connection is known as inter-processor communication.

Traditionally the bus structures have "occurred" in an ad hoc manner. However as systems become more complex, the design of the bus becomes more important [44]. The bus can often be the limiting factor in performance, modularity and reliability.

There are three major issues involved in the design of buses. They are:

- dedicated or shared lines,
- the communication technique,
- the data transfer conventions.

A dedicated line is always assigned to one physical pair of devices. A shared line can be shared by a number of physical devices and so needs special control hardware and/or software. A number of questions arise about the efficiency of protocols, and the interactions with other protocol design decisions, and system design decisions.

There are a number of basic data transfer conventions which determine how the data is sent along the bus e.g. single word transfers only, fixed length block transfers only, single word or variable length block etc. The width of the bus also has a large impact on the system.

The buses may be parallel or serial. In parallel mode the entire data word is transferred at one time. A serial bus is usually a single circuit interconnection between source and

destination. The data is transferred only one bit at a time. The single circuit is used for both control and data transfers.
The volume of data transferred during the connection of source and destination tends to influence decisions on the other factors. Data volume can be assigned as:

- single word,
- fixed length block,
- variable length block.

3.4 The Structure of the Processors

Processors may be logically equal or hierarchically structured. A hierarchical structure implies a master-slave relationship. These systems are usually easier to build but less flexible. Logically equal systems are more difficult to build but more flexible, reliable and capable of dynamic load sharing.

3.5 Bus Protocol

The main bus protocols are synchronous and asynchronous.
The total data transferred includes bits required by the protocol plus the message. Thus the actual transfer of n bits of data requires some m bits of protocol signals, and the effective bandwith in message bits per second is lower than the bit transfer rate of the bus. Obviously the protocols are a necessary overhead which determines the bandwidth of the bus. There is a trade off between costly protocols and bandwidth. The cost is the number of signal lines, drivers etc.

required to implement the protocol as well as the logic cost of the components required for implementation.

The asynchronous protocols fall into two categories, the one-way protocols and request/acknowledge protocols.

One-way protocols are either source or destination controlled. Timing between source and sink is implicit and no acknowledgement of data reception is given. The major problem is ensuring an appropriate delay so that data is valid before it is captured. The advantage is the simplicity.

A request/acknowledge protocol demands an acknowledge from the receiver and in some cases a negative acknowledge.

Three issues must be considered for synchronous communications, they are:

- the mechanics of synchronisation,
- data transmission,
- verification.

The synchronisation can be generated either locally or centrally. A centralised system has a clock signal which is generated to all units on the bus. This is subject to errors due to skew as distances increase. A framing signal can be propogated which is decoded by each unit and used to drive local counters which then identify the correct time. The framing sequence may be distributed by a separate bus or mixed with data.

Having reached an agreement on time, each unit can be allocated a time interval (slot) during which it utilises the bus. These slots are allocated on a dedicated or non-dedicated basis. Verification of the data by the receiver is usually required to ensure the reliability of the transfer. This requires an acknowledgement to be transmitted back to the sender in the appropriate time slot. Verification by default is often used, which implies a request for retransmission in a designated time slot only if an error occurs.

3.6 The Coupling of the Processors

Systems can be tightly coupled or loosely coupled. A tightly coupled system has a common memory. The loosely coupled systems have a memory for each processor. A lot of systems are somewhere between tightly and loosely coupled. Loosely coupled systems cut down on queueing to access memory and decreases the occurrence of deadlock situations. Deadlock can occur however when one processor is waiting for some information before it continues executing. This can be eliminated if the information is held in a variable and the last available information is always used (i.e. it is not necessarily the most current information but it does prevent deadlock from occurring).

If a system is loosely coupled and the only communication between the processors is through messages then the processors are independent of one another. This makes a

system fault-tolerant because when one processor malfunctions it has very little effect on the others in the system.

3.7 Information Contained in the Frames

The final matter to be considered is the way the frames [43] are to be constructed. This is of course dependent on all the factors discussed above. A frame may contain any of the following [33]:

- a synchronisation bit (one or two),
- a field to indicate the type of the frame,
- a count indicating the number of bits in the message,
- start of header marker,
- a header containing the address of the receiver, address of sender etc.,
- start of text marker,
- end of text marker,
- the message,
- the cyclic redundancy code.

It is necessary to keep information to a minimum while retaining all the appropriate information, especially when the messages are short (section 3.5).

3.8 Systems which can be Monitored

Many systems do not fit tightly within the categories outlined above. In fact, it has been shown in Chapter Two that multiprocessing techniques are not clearly defined, so some

limitations must be placed on the systems we are able to monitor. To make the monitor as general as possible it was necessary to break it into two parts, the main monitor section and the communications handler.

The main monitor section remains constant for any system that is being monitored. It is necessary to change the communications handler however, to suit the features of the system being monitored.

So generally this monitor will work for any system which has a common bus, some form of common clock, and some means of halting all system processes eg. for the monitor designed for RTS a signal of the bus is used for the NMI.

The number of processors this monitor can analyse is not limited by the hardware or software. The system is built so that any number of processors could be added. The limiting factor would be the speed.

It is necessary to decide on a system to test the monitor on, so the rest of this thesis refers to the monitor as it was designed for RTS, keeping the monitor as general as is practical.

3.9 RTS

At this point it is necessary to look at how RTS is partitioned as this may influence some decisions which are made when designing the monitor.

RTS can be partitioned into four separate functions:

- I/O,
- Keyboard/Display,

- Identifier (Id),
- Control.

When the partitioning was initially considered, one processsor was dedicated to a system observer. However the monitor will now do this task.

The I/O involves the A/D and D/A conversions, and the communications with other peripherals. Each task would have to be given a priority, and the A/D and D/A conversions would be top priority because these operations must be done in real-time.

The Keyboard/Display function performs the man/machine interface.

The Id identifies the model and state from A/D parameters. With this executing at the same time as the control function, the speed should be increased.

The control function is there to determine new actuator values at each time step.

With this partitioning the data to be passed between processors is at a minimum.

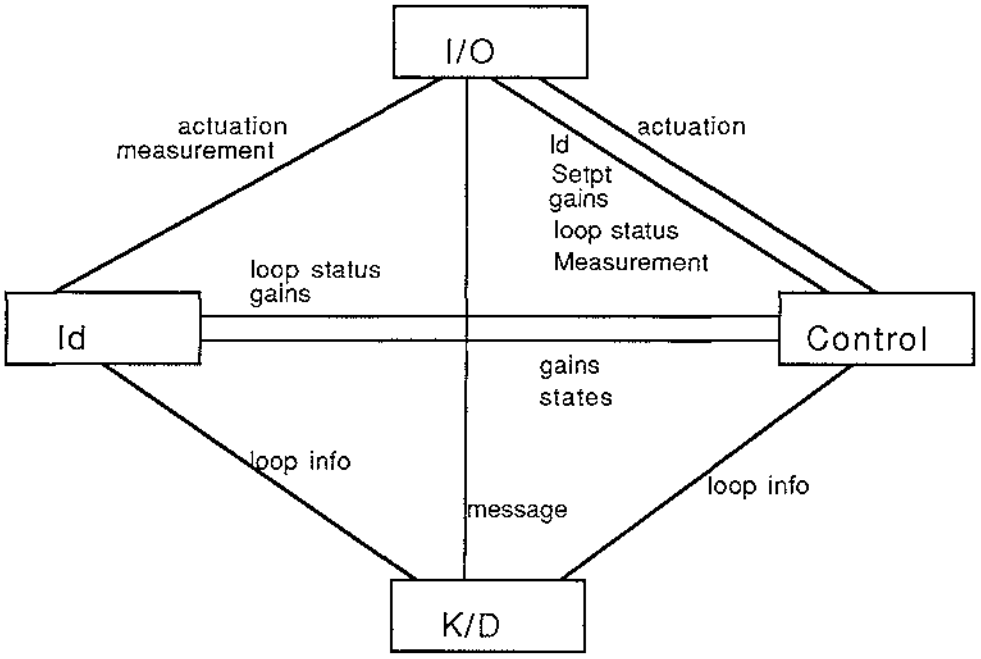


Fig. 5 Communication Between Processors

Fig. 5 illustrates the connections which are necessary between the functions and the information which must be passed. Notice that this structure is quite different than the structure in fig. 2 where each processor acts as a local controller. Fig. 5 is not the physical topology of RTS. RTS would be set up as a loop or ring network (see fig. 4) and the messages would be passed around the system from processor to processor until the message reached its destination.

Unfortunately the amount of information passed between processors is not always quantifiable. Some figures depend on the numbers of loops in the system while others are dependent on the number of system states or the order of the process. As

RTS stands the amount of the information passed is as follows:

I/O to Id

actuation	2 bytes
measurement	2 bytes
total	4 bytes/loop

This is the amount of information passed for one loop. A system can have a range from one to several hundred loops. However for a small system we are envisaging between one and ten loops.

I/O to Keyboard/Display

message This may only be 1 byte, it is dependant on whether a key is used or not.

I/O to Control

Id	2 bytes
setpoint	2 bytes
measurement	2 bytes
gains	45 bytes
loop status	1 byte
total	52 bytes

The gains are dependant on the order of the process. The number of states in most systems is between three and five. The loop status and gains need only be passed if the ID goes down.

Control to I/O

actuation	2 bytes

Control to Id

gains	45 bytes
loop status	1 byte
total	46 bytes

Id to Control

gains	45 bytes
states	180 bytes
total	225 bytes

Keyboard/Display to Control

loop info	this can be of variable length

Keyboard/Display to Id

loop info	this can be of variable length

The figures above illustrate that there is not a great deal of information passed between the processors. The other important point about partitioning is that when one processor fails the rest of the system should be able to carry on or close all necessary activities down rather than simply allow them to crash. That is, the system should be fault tolerant and should degrade gracefully. Looking at the system as it stands, if the I/O processor fails it would be necessary to stop the entire process or keep it at some steady state. A message would be displayed on the operator's terminal indicating that a fault had occurred.

If the Keybord/Display processor crashed, it would only be necessary to stop that processor and the rest of the system could continue in a limited form.

RTS is designed around the possibility of the Id processor failing, so this would have no effect.

If the Control procesor failed, the outputs could be set at some safe condition values.

It was considered that serial communications would be necessary for such a system as all processors may not necessarily be resident in the one box and they could indeed be distributed over a long distance. It would be impractical to use parallel communication from the point of view of both the noise and the cost. For a serial connection no provision has been made in the hardware for an NMI. However the RS-232C signal DCD can be used to generate an interrupt on all

processors. This doesn't have the priority of an NMI, however it is the best that can be done with serial communications. A high priority interrupt is necessary to force the processor to take some form of action no matter what it was doing before being interrupted.

Chapter 4

Design Specifications

Many papers have been written expressing the difficulties of debugging a real-time system [15, 37, 48] but few include the complication of debugging a real-time distributed system. This thesis outlines the design of a monitor for the development and debugging of a real-time distributed process control system. Some of the papers which discuss debugging a real-time distributed system [10,12] describe a system where special software must be included in each processor. These monitoring system are not as general as they could be, therefore they are not as easily adapted for use with other distributed systems.

Earl Van Horn [46] outlines three criteria for designing computer systems to facilitate debugging:

- record input streams,
- specify the input stream,
- reproduce an output stream.

These criteria do facilitate debugging of a batch-type system. However reproducing an output stream is impossible in real-time systems because the output not only depends on the input stream but also on the timing. Van Horn indicates allowances can be made so that the third criterion can be met, but this adds unnecessary complications to the system involved.

The word "monitor" is used throughout computing literature. There are a number of software and hardware items which are all called monitors e.g. those used in operating systems [38, 39, 43]. The monitors in this thesis have the same definition as those referred to by Witschorik [48]. A monitor is a tool designed for the debugging of systems. In this case the system is a real-time distributed process control system.

The properties of a performance monitor as listed by Allworth [1] are:

- it should be possible to extract the necessary performance characteristics from the system being measured,
- it must cause minimum interference to the system being measured,
- it must be convenient to use.

These three properties also apply to the debugging monitor designed below.

4.1 Monitor Requirements

This section outlines assumptions which this monitor is built around, and then lists the requirements of such a monitor.

An outline of activities a user may wish to carry out is given. Using the requirements as defined, a description is given showing how the activities can be satisfied.

The requirements are listed again outlining their importance by showing the priority of their implementation.

Finally the details of how these requirements can be implemented is described.

4.1.1 Initial Assumptions

Before the target system (i.e. system to be monitored) can be run under the control of the monitor the following assumptions must be met:

- the individual processors must be tested as far as possible (e.g. with an off-the-shelf ICE (in circuit emulator)), before interconnecting them and using the monitor,
- it should be possible to download programs to all processors,
- the states in the target system must be able to be set using the target system.

4.1.2 Requirements

The monitor must be able to meet the following requirements to be viable:

- Start and finish times must be able to be synchronised both at the beginning of the program and at breakpoints.

- It should be possible to set breakpoints based on time or on the beginning or end of message transmission.

- An interrupt mechanism must exist so that the user can push a control key (e.g. BREAK) to stop the system.

- A user should be able to view the values of certain variables. He should be able to list which variables he

wants to view the values of and perhaps set up a screen format for these variables. To make this possible the user would have to enter both the processor id and the address of the variable within that processor.

- A user should be able to view the data at i/o ports. The messages to be viewed would have to be qualified by destination, source or message type.

- Records timing data (i.e. times when messages are sent and received); and times when processors are active and inactive.

- Statistics should be calculated from the above when the processors are halted.

- The relevant log data should be stored (i.e. data that is necessary in the running of the target system (eg. RTS)). The availability of logs allows designers to make the best decisions.

- Get info on processor states when processing has been halted.

- The monitor should be able to act as a process-to-be-controlled. The monitor stores the messages sent from the target system when instructed. The monitor can then

act as the target system by sending all these messages at the appropriate time. This is the closest to reproducing an output that is possible for a real-time system.

4.2 Possible Scenarios Using the Monitor

Assuming each processor has been checked separately (e.g. using an ICE), the testing now concentrates on the interaction between the processors ie. message transmission, timing, synchronisation and the effect of the processors on the other individual processors.

The following descriptions are possible problems and scenarios applied to debug or decipher the system, using the activities described above.

Problem : The target system stops.

Action : 1. set breakpoints

2. at each breakpoint, look at all messages that are passed and see if they are as expected.

3. at each breakpoint, look at all values in the log and estimate if they are as expected.

4. at each breakpoint, view values of specified variables.

5. the above process can be run in an iterative manner, narrowing down the problem area all the time.

6. the interrupt mechanism can be used to stop the system as desired.

Problem : The target system is in an infinite loop.

Action : 1. same as above.

Problem : The target system has a runtime error e.g. variable is uninitialised (this is possible if it is waiting to be initialised from another processor), read/write error etc.

Action : 1. set breakpoints
2. view values of specified variables

Problem : The target system has run for days and suddenly crashes.

Action : 1. look at log data
2. look at timing data
3. look at info on the present states of the processors
4. attempt to set-up RTS as it was when it crashed (it would take too long to go all the way through the processing again).
5. run the monitor as the process-to-be-controlled.

Problem : The data displayed on the screen is not changing (this could be because the processors are not communicating). One or more of the processors may have crashed.

Action : 1. look at the messages being passed between certain processors.

2. look at the timing data - when messages are sent or received, and when the processors start and stop.

Problem : The target system is working but the values displayed are unlikely.

Action : 1. set breakpoints.
2. view values of certain variables.
3. look at messages being passed to certain processors.
4. run the monitor as the process-to-be-controlled.

Problem : Values in memory are wrong (something could be overwriting memory).

Action : 1. same as above.

Problem : Find the performance of the target system or compare different systems.

Action : 1. look at the timing data - when messages are sent and received, when processors start and stop.
2. analyse the statistics.

4.3 Priorities of Requirements

The requirements listed above fall into three categories:

- the necessary requirements,
- those which are useful for the user to have available,
- those which are a luxury, but are still nice to have.

This section indicates priorities and hence an order of implementation (i.e. "key" requirements should be implemented before "make life easier" requirements).

4.3.1 Key

- breakpoints, at set times,
- view values of specified variables,
- view i/o port information,
- record timing data - sending/receiving messages, active/inactive processors,
- synchronise start/finish,
- run the monitor as the process-to-be-controlled.

4.3.2 Make Life Easier

- breakpoints, after a qualified type of message transmission,
- view i/o port information either to or from set processors,
- log the relevant data,
- info on procesor states at present.

4.3.3 Nice to Have

- set up screen formats for viewing variables,
- statistics on the timing data.

4.4 The Implementation of the Monitor Requirements

The monitor must be user-friendly or people will not use it. To this end the requirements must be implemented so that they are readily available to the user and not difficult for the user to administer.

The NMI line or a top priority interrupt and a common clock must be connected to all processes. For parallel communications this is easily achieved. However, for serial communications this implies synchronous transmission where the baud rate or the bit rate clock recovered from the transmission will serve as "clock ticks".

In the implementation an NMI is used for synchronisation whenever the system is started or stopped so the NMI line can not be used in the target system. The use of the NMI is the preferred mechanism for the start/stop synchronisation, however if it is not possible then an interrupt with the highest priority will do. In this monitor design INT is used for the synchronisation of "clock ticks". The synchronisation using NMI and INT could be given using some other mechanism in other implementations. This would make it necessary to change something in the communications handler - no changes would be necessary in the central monitor at all.

In defining the requirements, the monitor is recognised as being target-system dependent. It is dependent on the message representation, and the physical system structure. One of the main aims of the monitor was to keep it as general as possible. This is accomplished by dividing the monitor into two sections

- a communications handler and the central monitor. All the changes that occur affect the communications handler so this is the part of the program which is changed. The central monitor and the interface between the communications handler and the central monitor are standard and are unaffected by the differences between implementations.

The size of the logs will be dependent on the amount and allocation of memory available in the monitor. This can be increased simply by changing values in the declarations in the programs and recompiling them. If they are likely to change often, it would be possible to ask the user how many values he wants logged for various histories.

The rest of this section indicates how the requirements listed in section 4.1 can be implemented.

4.4.1 Synchronised Start and Stop.

There would be one line connected to the NMI on all the processors, used for starting and stopping the processing. If all the processors are in the STOP state then an interrupt on the NMI line will put them in the START state, and vice versa. During the stopped phase all processors read the line for messages and/or commands from the monitor.

4.4.2 Breakpoints

It is relatively easy to base breakpoints on time. The user enters n (the number of clock ticks between breaks) and the system counts n ticks before sending an NMI (section 4.1.1).

Setting breakpoints, based on messages, involves viewing each message and inspecting a list to see if that message should cause a breakpoint. The message passing differs for different system topologies.

Physical Structure and Message Passing

If the topology of the system is a loop (fig. 4a), the messages are transmitted from the source and are removed from the loop when they return to the source. As each message passes past the monitor, it can be investigated.

If the topology of the system is multi-drop (fig. 4d), the messages are effectively broadcast so any message sent will arrive at the monitor (as well as at the destination node), and can then be analysed. Each node removes the copy of the message which they receive.

Once the monitor is able to recognise the messages, it can stop the processing after the appropriate message (section 4.4.1).

4.4.3 Interrupt Mechanism

A key must be recognised by the monitor as an interrupt (e.g. BREAK). The monitor would then make the NMI line active (section 4.4.1).

4.4.4 View Values of Variables

The user must know which processor, and where in that processor the variables are stored (i.e. the processor number and address of the variable). The processor, the address and

the name of the variable can be entered and a table will be set up in the monitor. The name is entered so the data can be printed out on the screen along with the name.

The monitor can investigate the values of these variables in the processors after an NMI occurs i.e. the values of these variables are not logged in real-time: they are logged when a breakpoint (section 4.4.2) or interrupt (section 4.4.3) from the monitor occurs. The monitor sends a message to the relevant processor requesting the value and the processor sends a return message giving the value. The use of messages in this manner allows the monitor to maintain its generality. The message mechanism already exists in the target system.

4.4.5 View Data at I/O Ports

The user enters information indicating which messages he wants logged, based on type, source or destination. He also indicates if the whole message should be stored or only the header. The messages can be investigated when they reach the monitor (section 4.4.2).

4.4.6 Record Timing Data

The monitor can simply store the necessary information when it identifies a message (section 4.4.2).

To detect if the processors are active or inactive it would be necessary for each processor to have two variables - active start and inactive start. The time when the processors become active and inactive can be stored, and when the processor

This would take up very little processing time (unless a processor is starting and stopping a lot).

4.4.7 Statistics

The monitor designer for the specific system must decide what statistics are necessary. Most of the necessary information is already logged (section 4.4.6).

4.4.8 Buffering of Log Data

Most of the information can be read from messages of a certain type so it would be necessary for the monitor to know how to identify these messages and get the relevant information from them (section 4.4.2). It could be possible for the user to enter templates for each message on a floppy disk at the beginning of a monitor session - this would allow the monitor to be less target-system dependent.

4.4.9 Activity State of the Processor

This can be read from the activity flag or the timing information which is logged (section 4.4.6).

4.4.10 Monitor as Process-to-be-Controlled

The user must initially say that the messages to be logged are those from the A/D converter. The monitor will then be able to feed these values back into the system as though they were coming from the process-to-be-controlled. Ideally the monitor should be able to operate as normal as well (i.e. be able to log

various variables values etc.) but this may not be possible as it may cause timing problems.

Chapter 5

Details of the Implementation

Although the central monitor has been designed to be independent of the hardware implementation, a decision must be made about the structure of the monitor system. This decision is influenced by the hardware implementation of the multiprocessor system to be tested. This chapter is a description of a multiprocessor design implementation for a system with only two processor boards. The reasons for the choice of vehicle for this implementation is also described. The same processor boards will be used for the multiprocessor implementation of RTS, so RTS is also considered throughout this chapter. There are a lot of decisions outlined which are decisions of the multiprocessor designer.

This implementation involves two microprocessor boards and a Cromemco linked together in a loop, with a serial bus. The Cromemco acts as the monitor. The serial bus is used because the number of messages passed between processors is small and the use of the serial bus minimised the hardware requirements for RTS. As the communication is over very short distances with very little noise, there is no handshaking protocol introduced between the processors [31]. This would be a decision of the multiprocessor designer.

5.1 Hardware of the Target System

The microprocessors (Z80s) were chosen for their availability, cost effectiveness, simplicity and reliability. A box had been constructed for a controller and it was decided that the same box could be used for a multiprocessing version of RTS. This system was chosen to reduce design and development time.
A Cromemco was used as the monitor as it was readily available and had the necessary support software.

5.1.1 The Board for RTS

The boards have a processor, memory and input/output hardware. The hardware is structured in two levels so processors can be identical or designed to suit any particular requirements. The connections between boxes have a serial connection.

5.1.2 The Relationship of the Monitor to the Target System

The monitor simply fits into the target system as though it is another processor in the loop network. In a system where there is no central controller, and no dedicated lines, the monitor can be inserted at any point. There is a screen and keyboard connected to the monitor processor to enable the monitor user to communicate with the monitor.

5.2 Language

The languages used to code the software for the monitor were PL/I and Z80 assembly language. They were used in the development of RTS and all the necessary facilities were available to test and debug the system. PL/I also:

- allows the user to construct the necessary data structures,
- allows the user to incorporate assembly language subroutines. This is especially necessary in the case where a command must take exactly one machine cycle (section 5.3).
- has exception handling facilities which allows the user to trap errors without crashing the system,
- is very readable,
- is portable from one machine to another,
- leads to a structured approach to program design.

5.3 Design of the Communication Handler/Monitor Interface

The monitor itself is divided into two processes, the communications handler and the central monitor. They operate as foreground/background processes where the central monitor is the foreground process and the communications handler runs in the background. The background function is interrupt driven and has higher priority. The communications handler will differ from implementation to implementation. However within an implementation the communications handler for this processor (i.e. the monitor) is a superset of the processor communications

of the other processors in the target system. The processor communications are independent of the monitor. While the communications handler differs for every different processor communications, the central monitor remains the same. Fig. 6 shows the structure of this processor and indicates where the communications interface occurs.

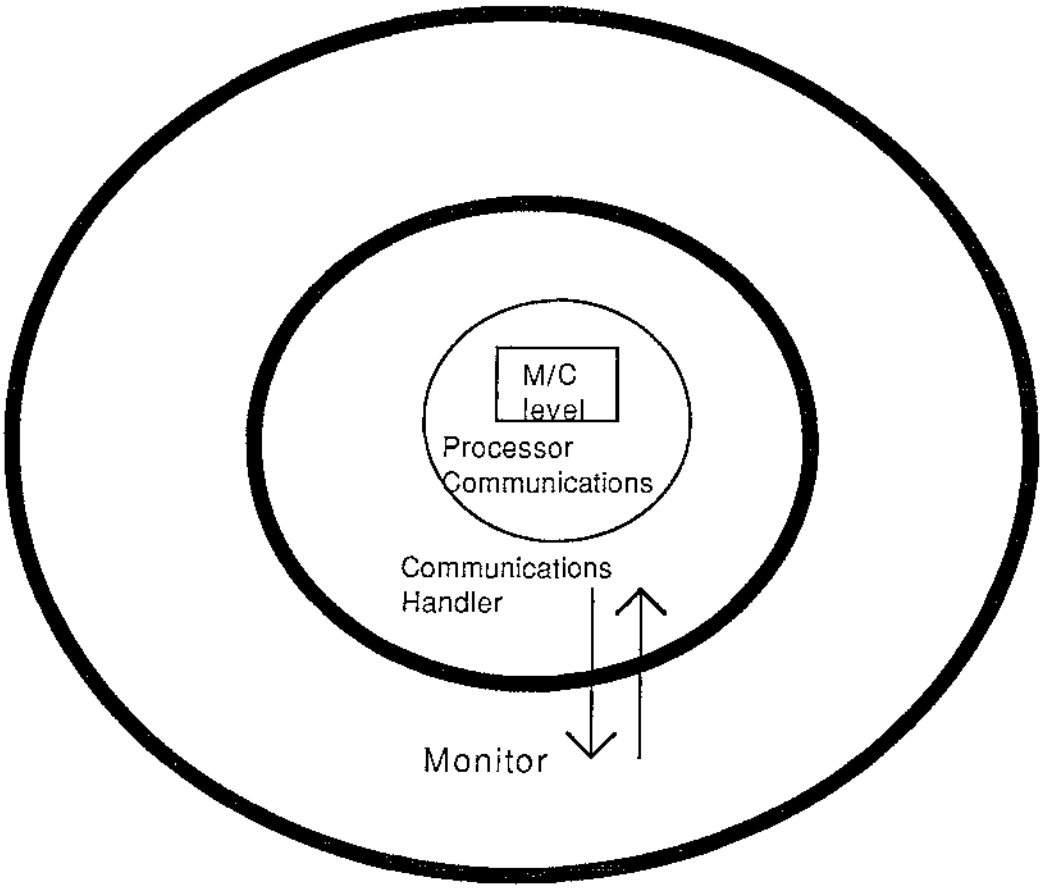


Fig. 6 Structure of the Monitor Processor and its Interface with the Target System

One of the aims of the design is to keep the interface between the communication handler process and the central monitor process as simple and as general as possible.

Bowen and Buhr [5] describe a possible method using two queues of buffers and two semaphores. One queue is for full buffers and the other for empty buffers. The semaphores indicate when all buffers are full and all buffers are empty.
They extend the example to where either the receiver or sender is an interrupt service routine (ISR). The difference is that the ISR is operating in real-time so it cannot wait on a semaphore. Problems can be avoided by continuing to use semaphores and disabling the interrupts at the start of the critical region and enabling them at the end.
This method is effective but it would involve both processes sharing all buffers so the interface between the processes would not be as well defined as it possibly could be and the processes would depend on each other. This is a reasonable method for a fixed design. However for a monitor where the communications handler is necessarily redesigned for every implementation, the boundaries between the communications handler and the monitor should be precisely defined and their operation easily observed.

Fig. 7 illustrates the proposed communications handler/monitor interface. When the topology of a network is a loop (section 4.4.2), the messages which are received by this processor must be passed on to the next processor and a copy is taken for this processor to analyse. The connection between the "receive process" and "transmit process" achieves this. If the network is a multidrop system, this connection is not necessary because all

messages are broadcast to all the processors so all the messages would be sent to the monitor and would terminate at this point. The message which is received may be in a different format from that which the monitor is expecting. In this case the message is reformatted in the communications handler section. It is then stored in the ring buffer and the variable **rxcnt** is incremented for each character passed. Every "clock tick" the monitor checks to see if **rxcnt** is greater than zero. If it is, the monitor receives the message, decrementing **rxcnt**, and stores the message in an array of messages.

When a message is sent from the monitor to the communications handler, it is initially stored in a ring buffer and the variable **sxcnt** is incremented. **Sxcnt** is also checked on the "clock tick". The message is stored in a buffer in the communications handler, reformatted if necessary, and transmitted.

Rxbase, rxlength, sxbase and **sxlength** are variables which are set at compile time so if there are a lot of ring buffer overflows, these values can be changed and the programs recompiled.

Deadlock [5] could occur if both the communications handler and the monitor were attempting to access either **rxcnt** or **sxcnt** at the same time. This can be overcome by using an instruction of only one machine cycle to do the addition and subtraction, because interrupts occur only at the end of machine cycles. This was achieved by two Z80 assembly language subroutines.

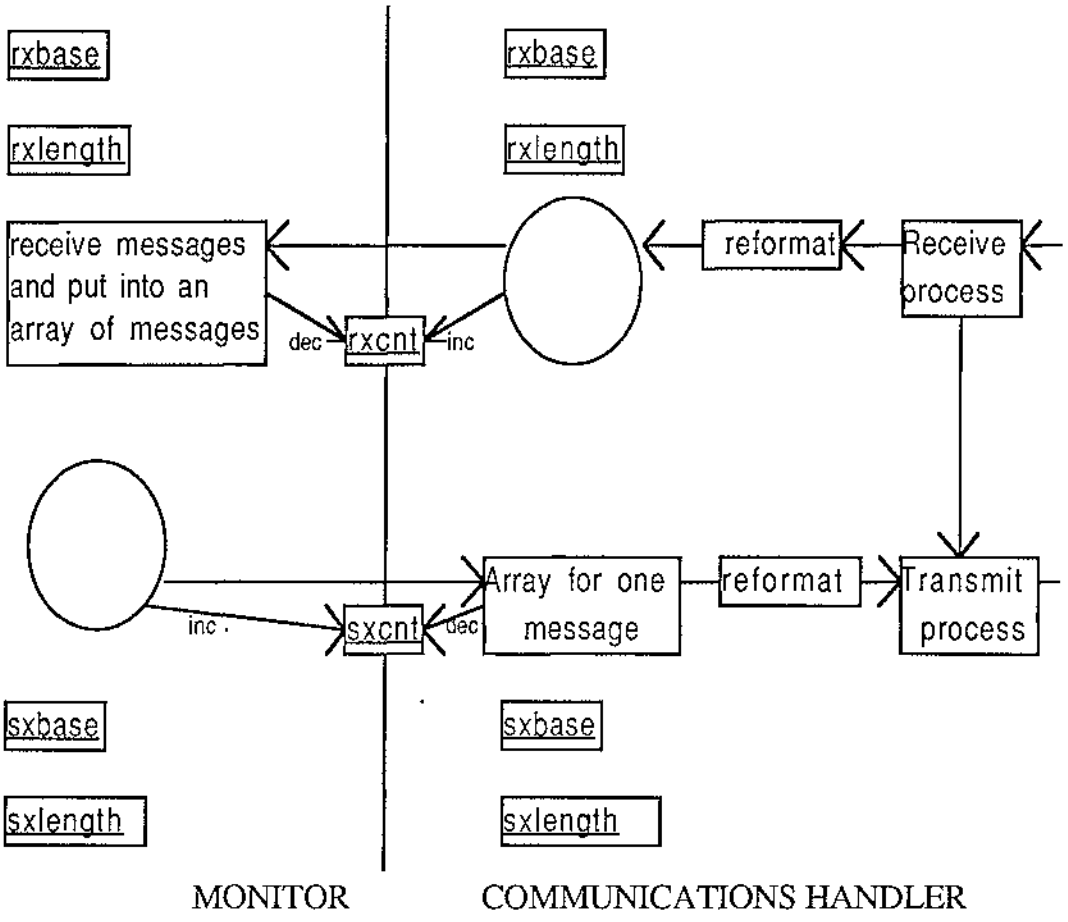


Fig. 7 Communications Handler/Monitor Interface

Another feature which the designer of the multiprocessing system must account for is the synchronisation of messages. Synchronisation of messages on the line can be a problem, i.e. if a processor is putting a message on the line when there is another message passing by, it could insert the message in the midst of the passing message. This is not a problem in systems with a loop structure because no messages go straight past a processor. Every message is taken into a ring buffer and the appropriate action is taken (i.e. it is passed on, or killed, or it is analysed).

All the information transmitted from this processor (i.e. the messages with a source number of this processor) is sent when the system has been halted. These messages are very short requesting some information from an addressed processor.

5.4 The Information in the Frames for RTS

Section 3.7 gives a general review of the information which is generally contained in the frames [43]. In the initial implementation the message template system was not implemented so a standard message was defined. As these messages are travelling over very short distances in an environment with very little noise, it was decided to keep the messages simple and each message contains:

- start of text marker (STX),
- type of message (section 5.5.1),
- the source of the message,
- the destination of the message,
- the data,
- an end of text marker (ETX).

The central monitor expects messages in the circular buffer of the communications handler to be formatted as above. If the messages which are passed around the multiprocessor system are in a different format, they must then be reformatted in the communications handler before being put into the circular buffer.

5.5 Data Structures Involved

It is necessary to have some data structures to hold:

- the information about which messages to log,
- the messages themselves when they arrive,
- the information about the variables to be logged,
- the values of those variables,
- the message templates.

This is as well as those data structures mentioned above for the temporary storage of messages when they are being transmitted or received.

5.5.1 Information About Messages

The program will ask the user if he wants all messages of a certain type stored (T), all messages from a certain destination (D) or all messages to a certain source (S). This information will be stored in a `type` field. An integer must then be given to identify the particular type, source or destination. The user can then choose what action is to be taken when a message of this type is encountered:

- log the header only (H),
- log the header and the information in the message (D) or
- cause a breakpoint (ie. halt the system) (B).

The data structure used for this is a linked list of records and is shown in fig. 8.

Type (ch)	Id (int)	Action (ch)
T S D		H D B

Fig. 8 Table Used to Store Information about Messages to be Logged

The structure where the messages are to be stored can be viewed as a sequential block of memory. The messages are stored sequentially in the order in which they arrive. The only method of accessing these messages is sequentially, and is therefore rather slow. This structure was chosen on a time/space tradeoff. The space requirement had priority over speed as logged messages are not accessed in real-time.

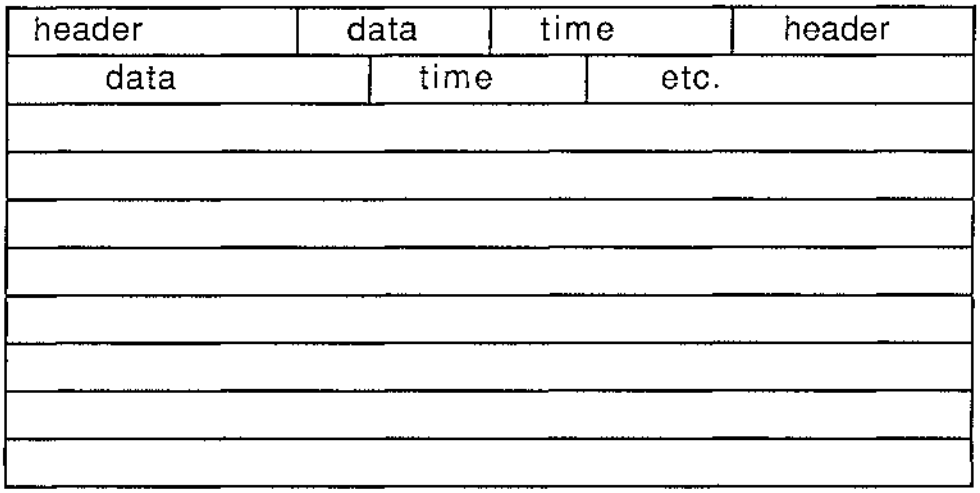


Fig. 9 How the Messages are Stored when Logged

Fig. 10 shows the structure in which the message templates are stored. It was necessary to have such a table to allow for different systems to have different message formats.

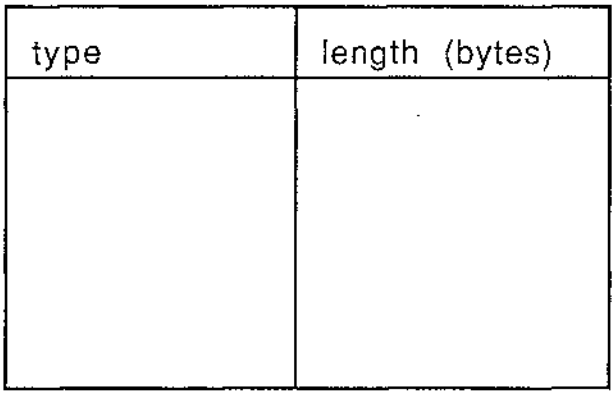


Fig. 10 Structure of Table for Message Templates

5.5.2 Information about Variables to be Logged

The information about variables to be logged and the values of the logged variables are stored in a single data structure. There is a linked list of records which stores the information about the variables to be logged. One of the fields in the structure is for a pointer, which can point to a dynamic linked list. When the table is constructed the user chooses how many values are to be logged and that number of nodes is created for the linked list. This linked list contains the values which the variables has had in a circular list. When the circular list has no more empty nodes, the oldest value is overwritten by the new value to be logged.

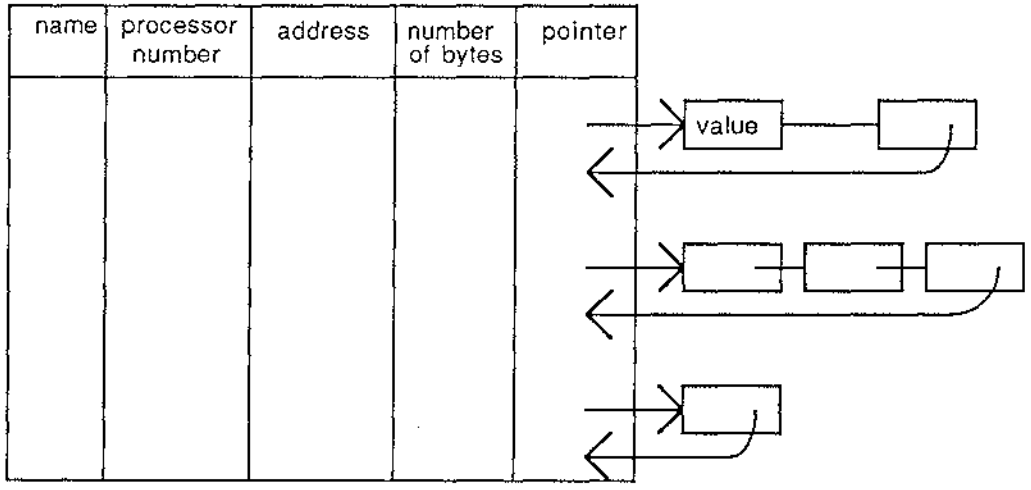


Fig. 11 Structure where the Information about Variables to be Logged and their Values is Stored.

5.6 Menus

The system is menu based. This enables the user to use the tool without any programming skills or specialist computing knowledge. When a tool is easy to use, there is no distraction

from the problem at hand i.e. developing an efficient multiprocessing control system. However if the tool is difficult to use a lot of time can be spent learning to use it without any direct results.

Looking at the menus and their connections can give some idea of the way the control of the program is passed.

A map of the menus is outlined below:

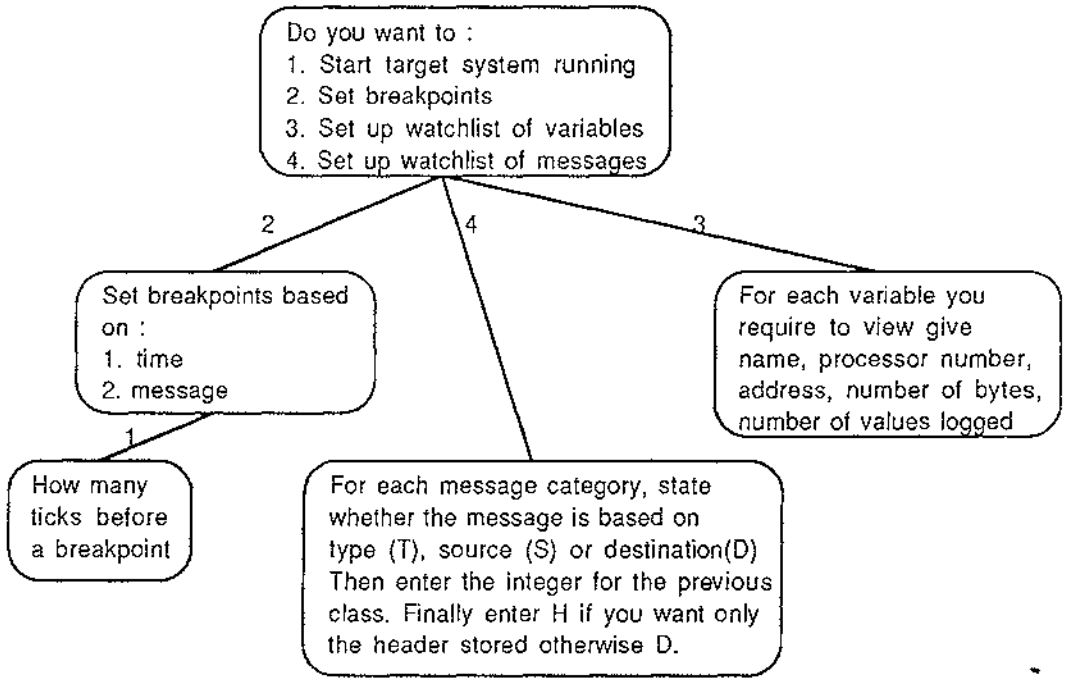


Fig. 12 The Relationship between the Menus

After a breakpoint has occurred, the following screens are displayed:

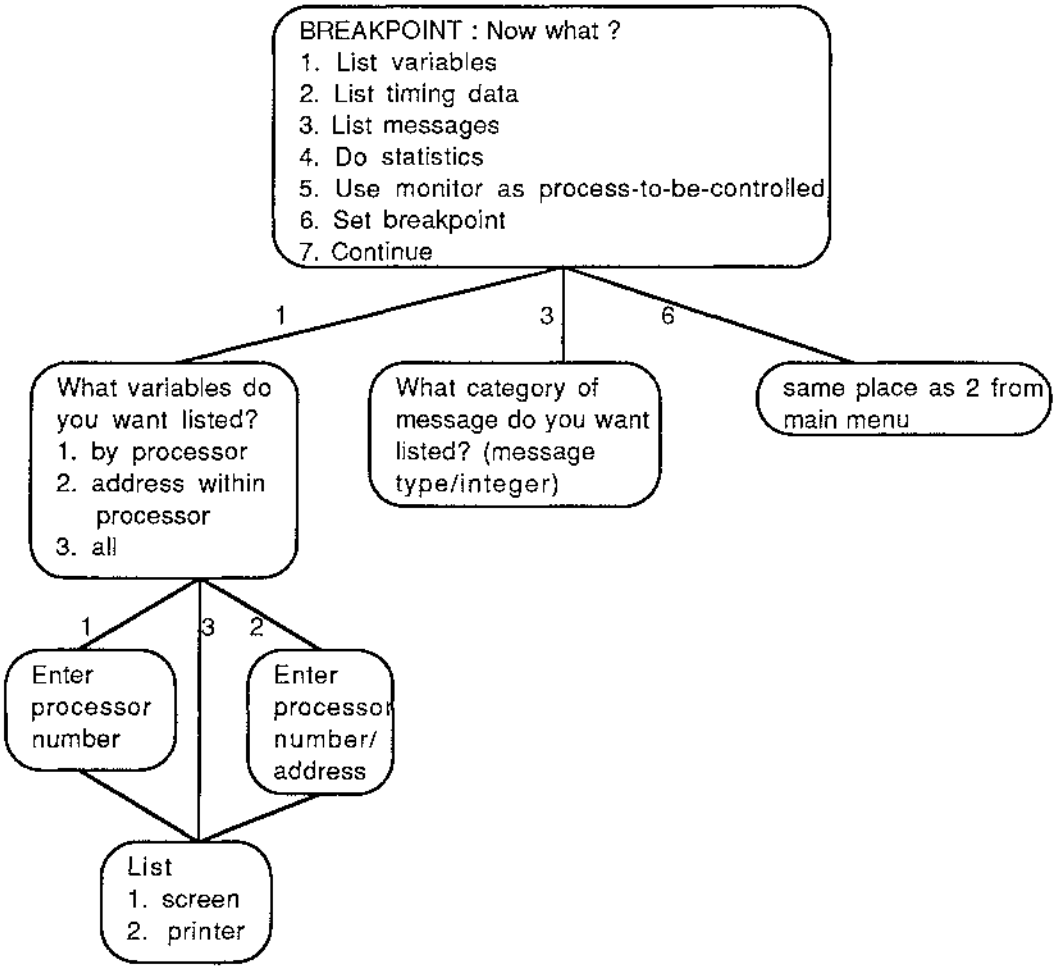


Fig. 12 cont. The Relationship Between the Menus

Chapter 6

Conclusions and Future Developments

6.1 Conclusions

The main objective of this project was to design a monitor to enable the development and debugging of multiprocessor systems for process control. This has been achieved and partially implemented.

While the monitor was kept as general as possible, it was decided that any monitor of this type had to be tailored to the system being designed. The effect of tailoring the monitor to whatever process control system was being designed was minimised by dividing the monitor into two sections: the communications handler and the central part of the monitor. All the necessary changes for different target systems could be made to the communications handler. So the central part of the monitor and the interface between the communications handler and the central monitor remain standard.

The use of the Cromemco to contain the monitor meant that the programs could be downloaded, and all the software which is available on the Cromemco could be used for development, i.e. the assemblers, compilers, linkers etc.

The final conclusion was that, although there were very few assumptions made initially, it is best to first have the monitor (i.e. a tool for development and testing) before the target system is designed as it ensures that the system is designed so

as not to preclude the use of the monitor and not upset the structure of the multiprocessing system.

This system, as it is designed, meets the criteria outlined by Allworth [1]:

- the necessary performance characteristics can be extracted,
- there is very little interference with the target system,
- it is convenient to use.

6.2 Future Developments

6.2.1 Complete the Monitor

The monitor is presently only partially finished so it does need to have other features added. The priorities set out in Chapter Four should be adhered to.

6.2.2 Man/Machine Interface

There are further refinements which could be added to the monitor, e.g. improving the man/machine interface.

6.2.3 Implement RTS as a Multiprocessing System

Now the tool has been designed, the task of implementing RTS as a multiprocessing system is simplified. RTS can be partitioned as described in Chapter One. With the use of the monitor, the multiprocessing will not appear such an awesome concept and the programmer can concentrate on equally important issues such as user-friendliness, fault tolerance, and the efficiency of the system.

6.2.4 Performance Criteria

The monitor was also designed for the analysis and prediction of specific aspects of software performance. To test the performance of the system it is necessary to lay out a criteria of how the performance is being measured. There are a number of papers and books written on testing system performance [18, 32].

6.2.5 Implement RTS with Different Partitioning

It would be interesting to implement RTS as a multiprocessing system, with the partitioning as shown in fig. 2, and then to compare the relative efficiency and speed of the two different process control systems.

6.2.6 Implement another Process Control System Using the Monitor

This would be an interesting exercise as it would indicate how RTS biased any necessary decisions in the design of the monitor. The designer of the new system would need to decide on their own communications handler. One could design a system using the monitor and also design the system without using the monitor, to test its usefulness. The author of this thesis thinks the results would show that the process control system designed using the monitor was designed in a more logical fashion and be running before the other system. A logically designed system is a lot easier to debug and change.

6.2.7 Exhaustive Testing

Introduce some method of programming the monitor so that steps are set out for it to run as the process-to-be-controlled i.e. the process-to-be-controlled is programmed to test different situations. In this way the multiprocessor product can be exhaustively tested without doing any damage to plant or equipment.

Bibliography

[1] Allworth, S.T.
Introduction to Real-Time Software Design. 2nd ed. The Macmillan Press Ltd., 1983.

[2] Astrom, K.J. and Wittenmark, B.
Computer Controlled Systems. Prentice Hall, 1984.

[3] Ayache, J.M. et al.
A Distributed Control System for Industrial Plants. Microprocessor Systems : Software, Firmware and Hardware. North-Holland Publishing Company, London, 1980, 243-250.

[4] Berg, H.K.
Distributed System Testbeds. IEEE Computer 18:2 9-11, 1982.

[5] Bowen, B.A. and Buhr, R.J.A.
The Logical Design of Multiple-Microprocessor Systems. Prentice-Hall Inc., 1980.

[6] Brayer, K. and Lafleur, V.
A Testbed Approach to the Design of a Computer Communication Network. IEEE Computer 18:2 14-23, 1982.

[7] Crosbie, R.E.
Simulation - Is It Worth It?
Simulation of Systems, 8th AICA Congress Delft. 15-19, 1976.

[8] Donaghey, L.F.
Microcomputer Systems for Chemical Process Control. Proceedings of the IEEE. 975-987, June 1976.

[9] Ergin, P.I. and Ling, C.
Development of a Non-Interacting Controller for Boilers. IFAC Conference Proceedings. 1960.

[10] Fagerstrom, J., Larsson, Y. and Stromberg, L.
Distributed Debugging - Collected Ideas.
LITH-IDA-R-86-21.
Department of Computer and Information Science, Linkoping University.

[11] Fujimoto, R.M.
Simon : A Simulator of Multicomputer Networks
UCBE -83-0140
Electrical Engineering and Computer Sciences, University of California, Berkeley.

[12] Garcia-Molina, H., Germano Jr., F. and Kohler, W.H.
Debugging a Distributed Computer System.
IEEE Transactions on Software Engineering 10:2 210-219, 1984.

[13] Gehringer, E.F., Jones, A.K. and Segall, Z.Z.
The Cm* Testbed. IEEE Computer 18:2 40-53, 1982.

[14] Gladstone, B.E.
Comparing Microcomputer Development System Capabilities. Computer Design 18 83-90, 1979.

[15] Glass, R.L.
Real-time: The "Lost World" of Software Debugging and Testing. Communications of the ACM, 23:5 264-271, May 1980.

[16] Gregoretti, G.
Software Development and Debug Aids for the μ* Multimicroprocessor System. Microprocessor Systems : Software, Firmware and Hardware. Euromicro Symposium London, 149-156. Sept. 1980.

[17] Halstead Jr., R.H., Anderson, T.L., Osborne, R.B. and Sterling,T.L.
Concert : Design of a Multiprocessor Development System.
Computer Architecture News 14:2, 40-48, June 1986.

[18] Heildelberger, P. and Lavenberg, S.S.
Computer Performance Evaluation Methodology. IEEE Transactions on Computers C-33:12, 1195-1220, 1984.

MEXTERN.PLI

This is a file of external declarations, which are used in a number of routines. They are kept here for easy access and to aid in documentation.

GETFIX.PLI is a procedure which displays a message on the screen asking a user for a value of type **fixed**.

GETFLT.PLI is a procedure which displays a message on the screen asking the user for a value of type **float**.

INTER.Z80 are assembler routines, taken mainly from RTS for various activities. Two other routines have been added. One of them increments a parameter, in one machine cycle. A command which takes one machine cycle is an indivisable operation, hence cannot be interrupted. The other routine decrements a parameter, in one machine cycle.

SEND.PLI outputs a character (if there is one to send) to a specified channel.

XFER.COM is used to transfer files from one disc to another.

ADDONE.PLI is used to increment the array indices. ie. the column count is incremented and if it is equal to 80 then it's set to 1, and the row count is incremented. If row is equal to 80 then both the row and column counts are reset to 1.

Maxwell files :

MONITOR.PLI is the main program that controls the monitor and directs the action.

There are a number of procedures which all fit together and should be useful when testing to see if two processors communicate with one another. Unfortunately as the multiprocessor system was not available none of these procedures have been tested:

TWOPROC.PLI,
INPROC.PLI,
SEND.PLI,
OUTPROC.PLI,
DISPLAY.PLI.

TWOPROC is the main procedure for this group. It gets a message from the screen, formats it, and puts it in the output buffer ready for another procedure to collect.

INPROC is the input interrupt procedure. It receives a character off the bus.

OUTPROC is the output interrupt procedure. It indicates that an output interrupt has been received by setting outint to true, then calls send.

DISPLAY indicates whether everything happened as expected. It should display the message on the screen of the receiving processor.

There is also a suite of programs for inserting messages into a data structure, asking the user which messages to log, logging them and listing them.

This includes:

MESSAGE.PLI,

WATMESS.PLI,

LISTMESS.PLI,

LOGMESS.PLI,

LMSDATA.PLI,

MESSAGE is the main program for testing these procedures.

WATMESS asks the user what messages he wants logged. They can either be of a certain **type**, to a certain **source**, or from a certain **destination**. This information is stored in the necessary linked list.

LISTMESS asks the user what messages he wants listed, and whether they are to be listed to the screen or printer. It then lists them.

LOGMESS takes the messages from **rxar**, checks to see if they should be logged, and logs them if necessary.

LMSDATA was simply used to test WATMESS.

There are a number of dummy procedures, which simply print out a line saying that they are executing. These are stubs to be replaced later by procedures that do the specified tasks. They are:

GETDISC.PLI,

START.PLI,

BPT.PLI,

DOWNLOAD.PLI.

GETDISC is designed to read information from a disc.

START is the loop which the monitor processes until a clock tick is reached.

BPT is activated when a breakpoint occurs. It displays a message on the screen asking the user what to do next.

DOWNLOAD is the program which downloads the programs through the monitor into the appropriate processors.

MESS.CMD is the batch file which tells the system to link the message suite together.

T.CMD is the batch file which links the monitor procedures together.

SETBRK.PLI asks the user whether he wants to set breakpoints on time or on a certain message. It gains the rest of the necessary data and either puts it in **bptime**, or in the message table.

TEMPL.PLI allows the information on how the messages are constructed to be entered from a disc. This information is then stored in the table for this data.

Var Files :

There is a similar suite to the programs for message storage. This suite is used for the storage of variables and their values:

VARIAB.PLI,

WATVAR.PLI,

GETVAR.PLI,

RECVAL.PLI,

LISTVARS.PLI,

PRINTVAR.PLI,

LVAR.PLI.

VARIAB is the main program for this test session. It calls watvar, getvar, recval and listvars.

WATVAR asks the user what variables to log, and how many values he wants logged. The data structure is then constructed for storing these variables.

GETVAR sends messages out to all the processors for each of the variables in the list, asking them to send the value of the particular variable.

RECVAL is used only for testing these procedures. It takes a message which includes a variables value from the screen, and places the value in the appropriate data structure.

LISTVARS lists all the variables which the user has requested to be logged. He can specify them either by processor number, by address within processor or request that all the logged variables be listed.

PRINTVAR is called from listvars. It prints a line of information about the variable then lists its values.
LVAR lists the variables to be logged. It tests watvar.

VAR.CMD is the instruction which allows all the variable handling programs to be linked.

TESTREC.PLI is used to test that the receive procedure works.
TESTSEND.PLI is used to test that the send procedure works.
TEST.PLI is used to test if the assembler routines add and sub are working.

RECEIVE.PLI receives a message from the circular buffer in the communications handler.

A2. Structure Diagrams

Structure diagrams have been drawn for the procedures which will be part of the monitor. There is also a structure diagram drawn for the procedures which test that the two processors are communicating, however there are no structure diagrams for the other test procedures.

To gain an appreciation of the workings of the monitor it is important to read the thesis as well as this appendix. The most important chapter is " **Chapter 5 : Details of Implementation**".

Structure Diagrams for Monitor

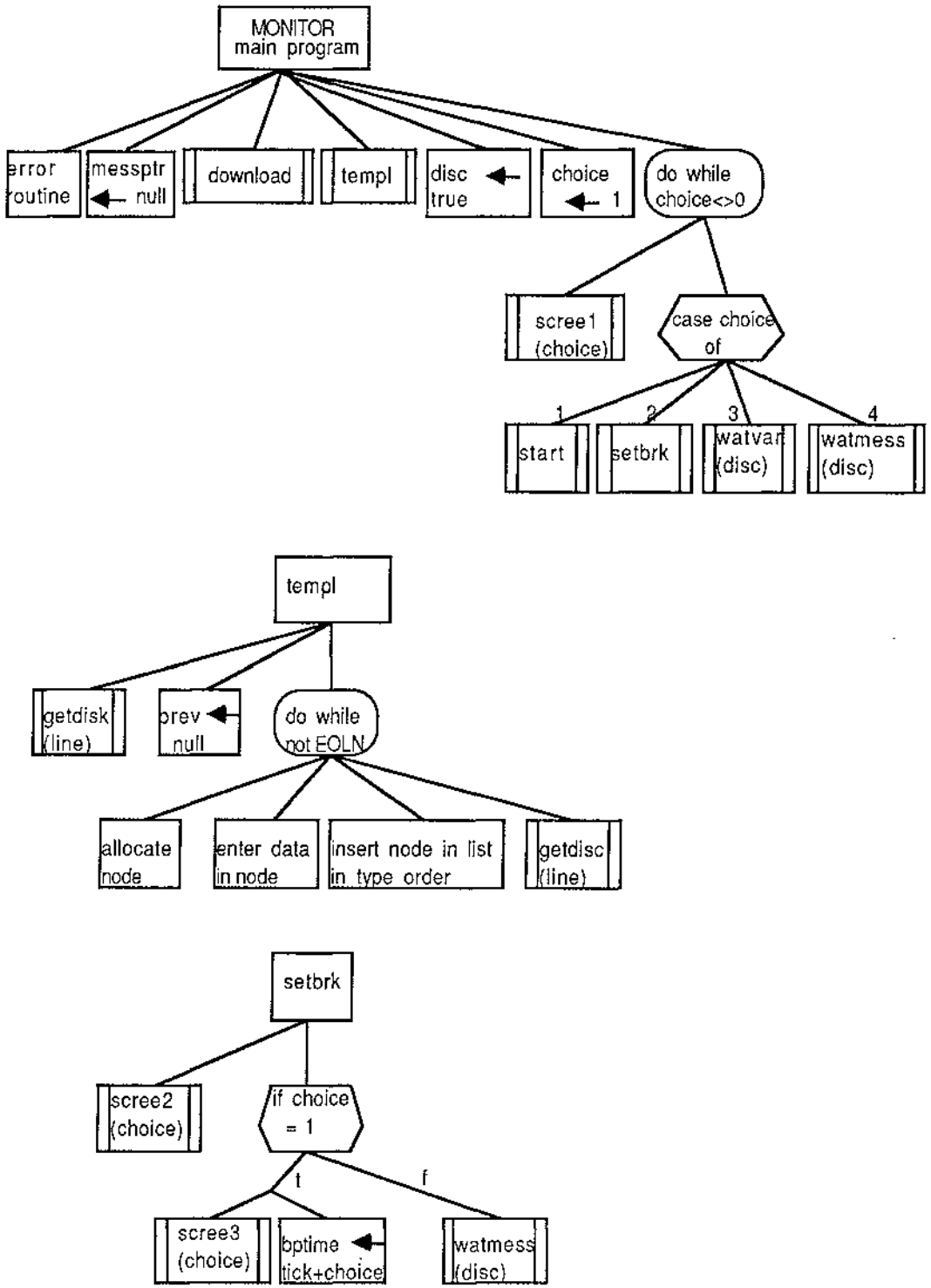


Fig. A1 The Relationship of the Monitor Procedures

Structure Diagrams for Monitor cont.

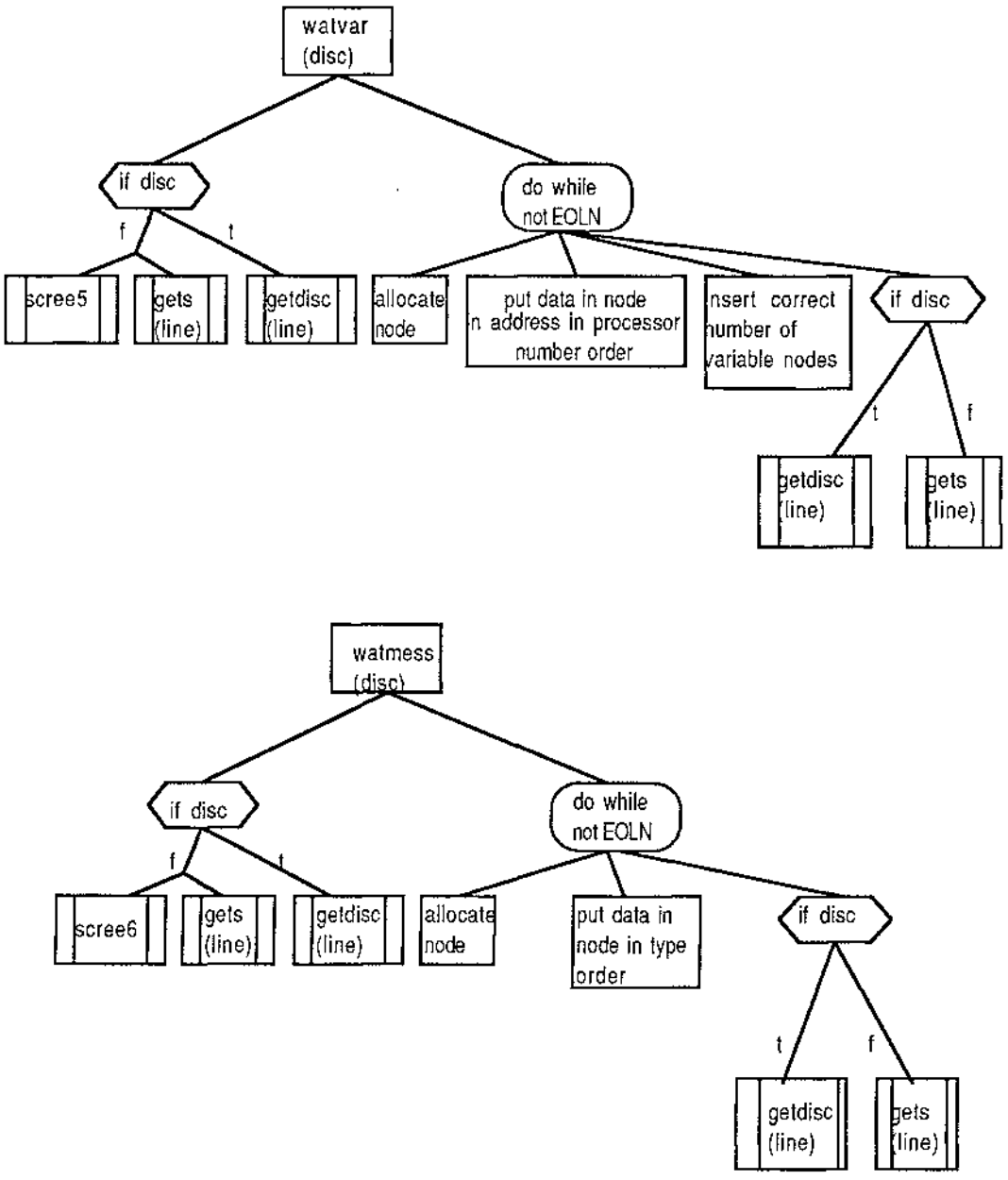


Fig. A1 cont. The Relationship of the Monitor Procedures

Structure Diagrams for Monitor cont.

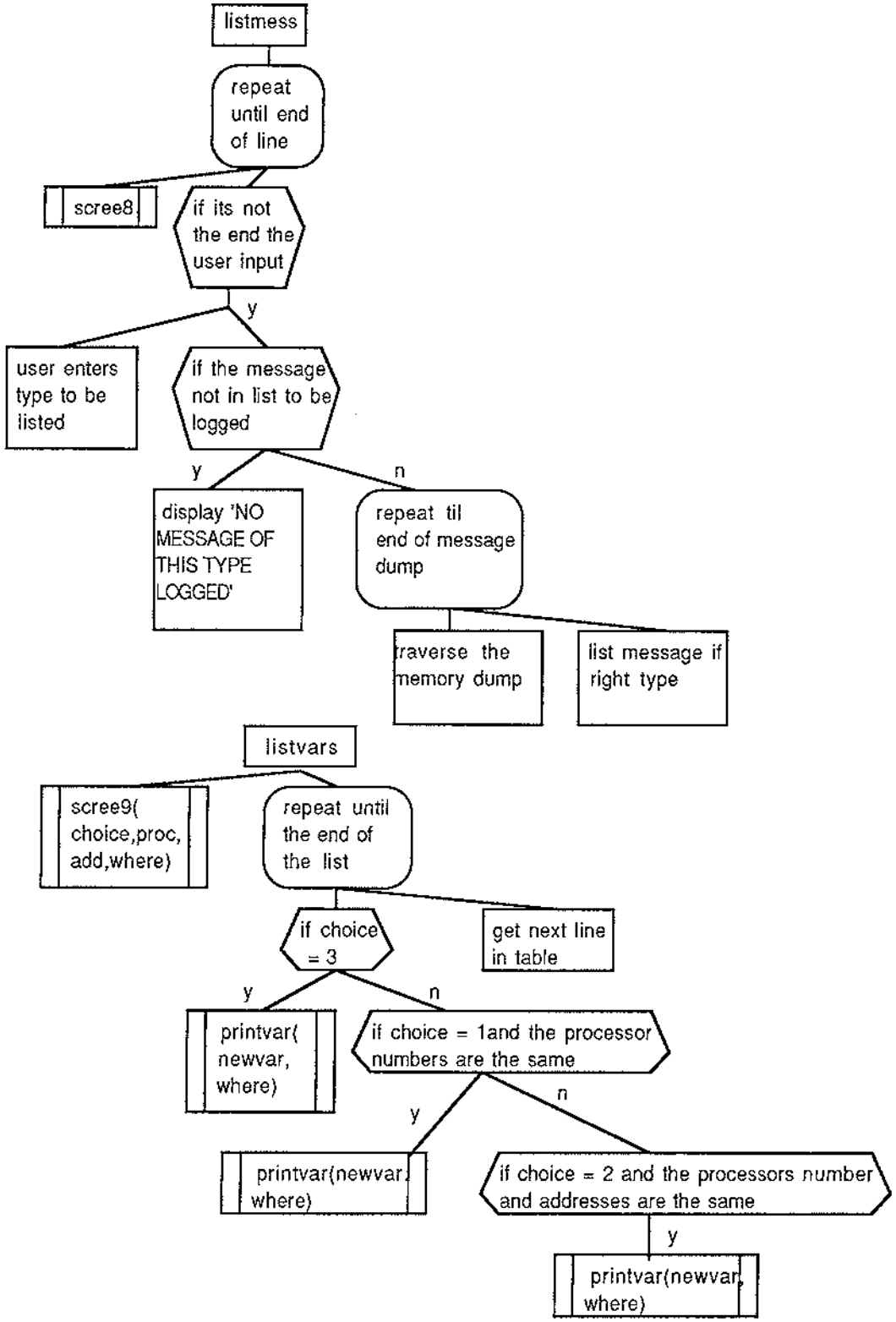


Fig. A1 cont. The Relationship of the Monitor Procedures

Structure Diagram for Monitor cont.

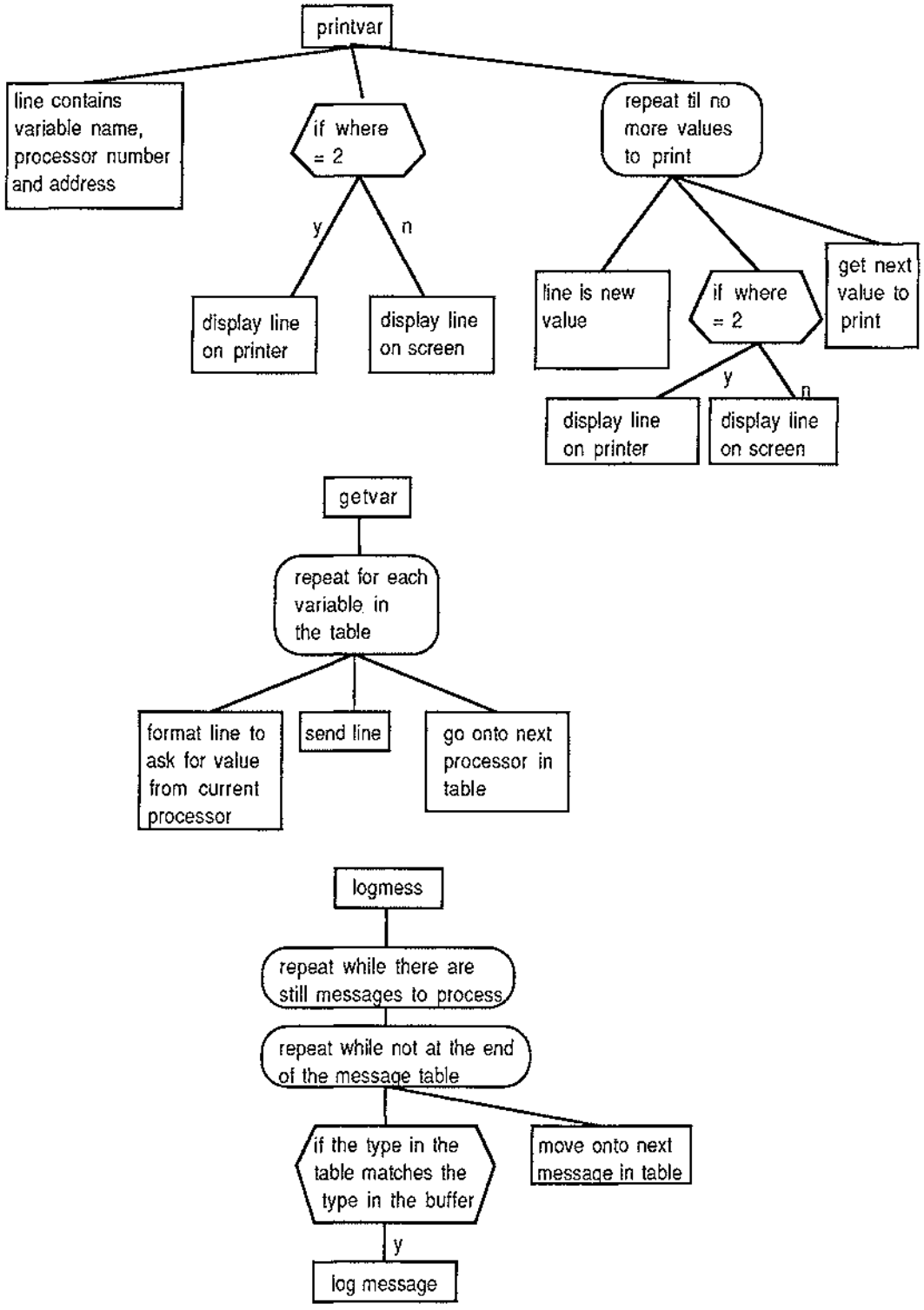


Fig. A1 cont. The Relationship of the Monitor Procedures

Structure Diagrams for Unwritten Procedures

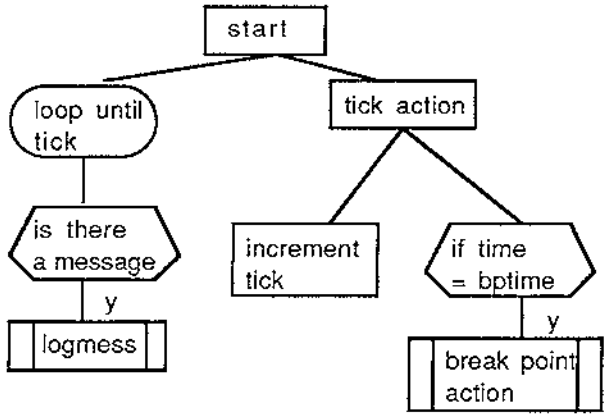


Fig. A2 Main Processing Once Monitor has Started

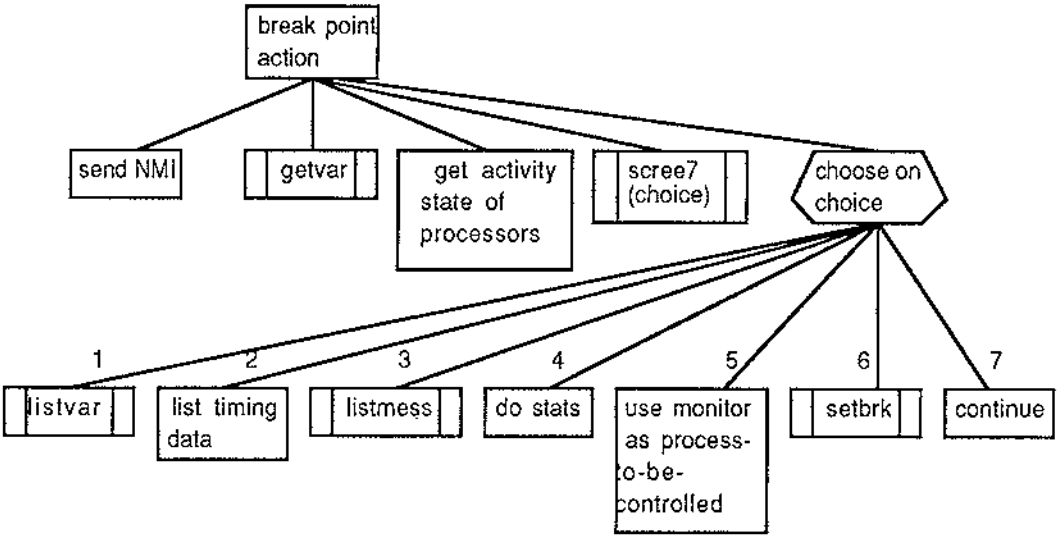


Fig. A3 Action After Breakpoint has Occurred

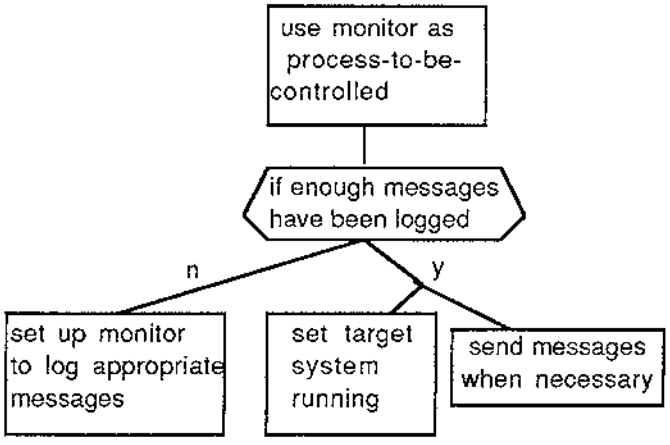


Fig. A4 Monitor as Process-to-be-Controlled

Structure Diagrams of Procedures that Test that the Processors are Communicating

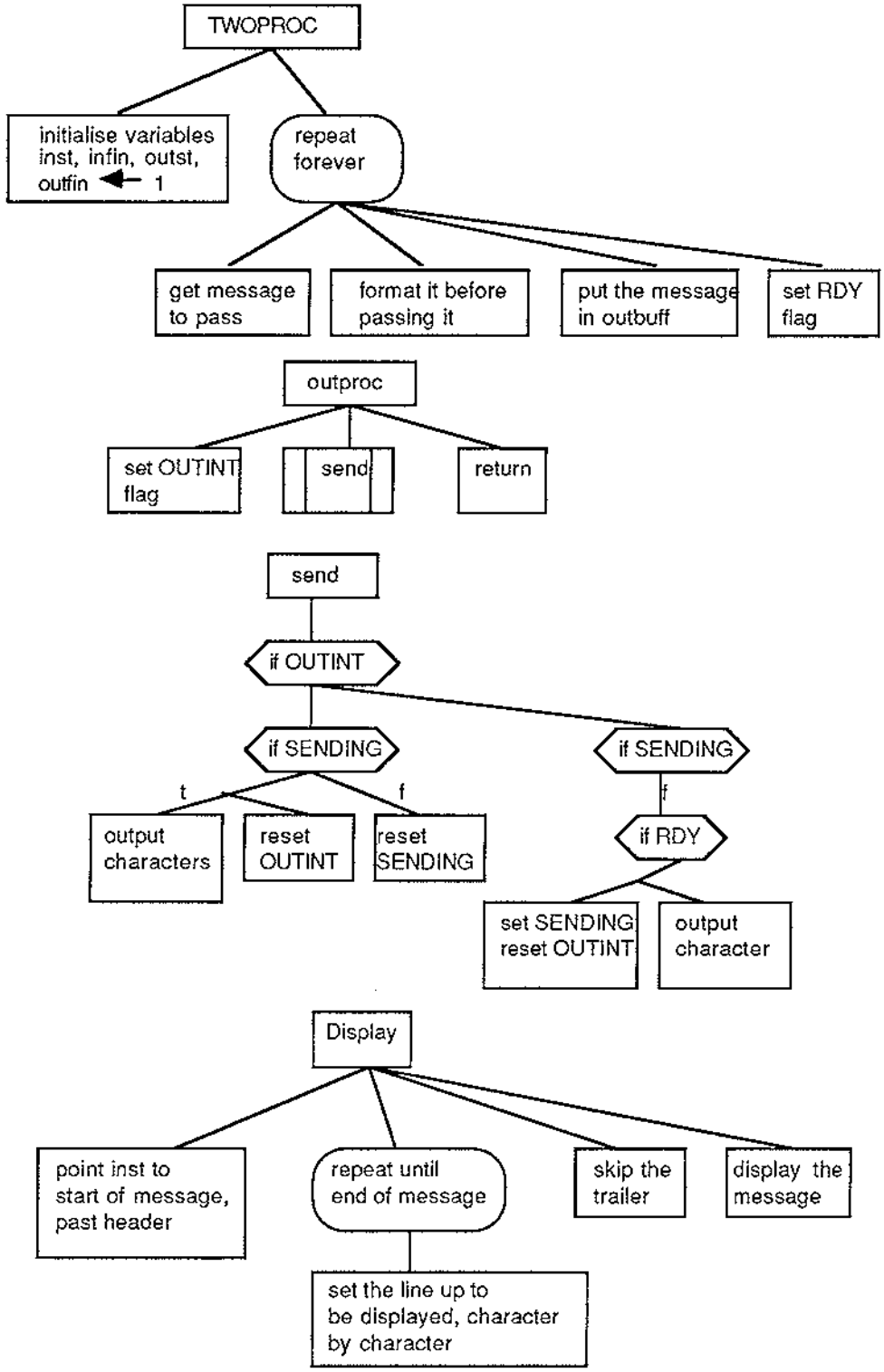


Fig. A5 The Relationship between the Test Procedures

Structure Diagram of Procedures that Test that the Processors are Communicating cont.

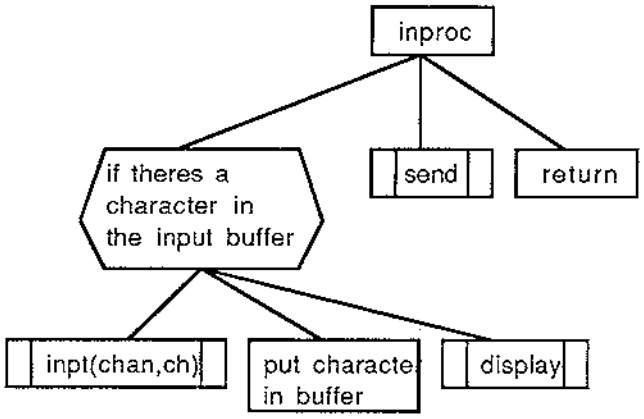


Fig. A5 cont. The Relationship between the Test Procedures

3. PROGRAM LISTINGS

These programs are listed in the same order which they were referred to in the program description.

Files on both the MAXWELL and VAR Discs

MEXTERN.PLI

```
/* external declarations or common routines and variables */
%replace home by '^[H';
%replace clear by '^[v';
%replace true by '1'b;
%replace false by '0'b;
%replace STX by '^B';
%replace ETX by '^C';
%replace monitornum by 3;
%replace vartype by 1;        /* message type when monitor sends message to get
                          variable information */
dcl (gets,puts,getdisc) entry (char(80) varying),
        send entry(char(80) varying,fixed),
        getfix entry (char(80) varying,fixed,fixed,fixed),
        watmess entry(bit(1)),

        disc bit(1) external,
        (finrow,fincol,bptime,tick) fixed external,
        line char(80) varying,
        choice fixed,
        logit bit(1),
        (tail,messptr,newmess,prev) pointer external,
        (temptr,newtemp,varptr,newvar) pointer external,

        (sxbase,sxchcnt,rxchcnt,rxbase,cycpos) fixed external,
        (sxcyc(200),rxcyc(200)) char external,
        rxar(80,80) char external,
        (logcnt,arcnt) fixed external,
        messdump(80,80) char external,
```

```
1 messnode based,        /* ordered on id within type */
   2 type char,
      2 id fixed,
   2 action char,
   2 nextmess pointer,

1 varnode based,        /* address in processor order */
   2 name char(20) varying,
   2 prnum fixed,
   2 address fixed,
   2 numbytes fixed,
   2 current pointer,
   2 nextvar pointer,
   2 infoptr pointer,

1 infonode based,        /* sequential */
   2 info float,
   2 nextinfo pointer,

1 messtemp based,        /* message type */
   2 messtype fixed,
   2 lgth fixed,
   2 nextemp pointer;
```

GETFIX.PLI

```
getfix: procedure(prompt,number,min,max);
/* displays prompt on screen, receives number from user and returns it */
dcl
  (gets,puts) entry(char(80) varying),
  line  char(80) varying external,
  prompt  char(80) varying,
  (number,min,max)  fixed;

  on error begin;
    line='RANGE'||min||max;
    call puts(line);
    goto onerr;
  end;

  prompt=prompt||'('||number||') ';
  onerr:
  call puts(prompt); call gets(line);
  if length(line)>0 then number=line;
  if (number<min)|(number>max) then signal error(255);
end;
```

GETFLT.PLI

```
getflt: procedure(prompt,number,min,max);
/* displays prompt on the screen, accepts number from the user and returns it */
dcl
  (gets,puts) entry(char(80) varying),
  line  char(80) varying external,
  prompt  char(80) varying,
  (number,min,max)  float;

  on error begin;
    line='RANGE'||min||max;
    call puts(line);
    goto onerr;
  end;
```

```
  prompt=prompt||'('||number||') ';
  onerr:
  call puts(prompt); call gets(line);
  if length(line)>0 then number=line;
  if (number<min)|(number>max) then signal error(255);
end;
```

INTER.Z80

```
        entry kbwait,print,inpt,output,gets,puts,keyhit,sub,add
getpar: ld e,(hl)
        inc hl
        ld d,(hl)
        inc hl
        ex de,hl
        ld c,(hl)
        ex de,hl
getone: ld e,(hl)
        inc hl
        ld d,(hl)
        ex de,hl
        ld a,(hl)
        ret
random: call getone
        ld b,a
        inc hl
        ld a,(hl)
        scf
        ccf
        bit 0,b
        jr z,un
        ccf
un:     bit 2,b
        jr z,deux
        ccf
deux:   bit 3,b
        jr z,trois
        ccf
```

```
trois:  bit 4,b
        jr z,quatre
        ccf
quatre: rr a
        rr b
        ld (hl),a
        dec hl
        ld (hl),b
        ld h,b
        ld l,a
        ret
gets:   call getone
        push hl
        ld c,0
        inc hl
getin:  call kbwait
        ld b,a
        sub 0dh
        jr z,crlf
        ld a,b
        sub 07fh
        jr z,delete
        ld a,b
        sub 08h
        jr z,delete
        ld a,b
        sub a,020h
        jp m,getin
        ld a,b
        call outch
        ld (hl),b
        inc hl
        inc c
        jr getin
```

```
delete: ld a,0ffh
        and a,c
        jr z,getin
        ld a,08h
        call outch
        ld a,020h
        call outch
        ld a,08h
        call outch
        dec hl
        dec c
        ld a,0
        ld (hl),a
        jr getin
crlf:   ld a,b
        call outch
        ld a,0ah
        call outch
        pop hl
        ld (hl),c
        ret
puts:   call getone
        ld b,a
        inc hl
getout: ld a,(hl)
        inc hl
        call outch
        djnz getout
        ret
outch:  out 1,a
waitou: in a,0
        and a,080h
        jr z,waitou
        ret
inpt:   call getpar
        in a,(c)
        ld (hl),a
        ret
```

```
output: call getpar
        out (c),a
        ret
print:  call getone
        ld b,a
        inc hl
more:   ld a,(hl)
        inc hl
        and a,07fh
        out 054h,a
        or a,080h
        out 054h,a
still:  in a,054h
        and a,020h
        jr nz,still
        djnz more
        ret
kbwait: in a,0
        and a,040h
        jr z,kbwait
        in a,1
        and a,07fh
        ret
keyhit: call getone
        in a,0
        and a,040h
        ld (hl),a
        in a,1
        ret
```

```
onst:   macro
        push af
        push bc
        push de
        push hl
        push ix
        push iy
        exx
        ex af,af'
        push af
        push bc
        push de
        push hl
        mend
offst:  macro
        pop hl
        pop de
        pop bc
        pop af
        ex af,af'
        exx
        pop iy
        pop ix
        pop hl
        pop de
        pop bc
        pop af
        mend
nokb:   offst
        ei
        reti
getnum: ld e,(hl)
        inc hl
        ld d,(hl)
        ex de,hl
        ld e,(hl)
        ld d,(hl)
        ex de,hl
        ret
```

```
sub:    call getnum
        dec hl
        ret
add:    call getnum
        inc hl
        ret
        end
```

SEND.PLI

```
send : procedure;
/* outputs a character if there is one to send, to a special port */
%include 'extern.pli';
   ch = ' ';
   if outint then do;
      if sending then do;
         if outst ~= outfin then ch = outbuff(outst);
         if ch ~= '^B' |  outst ~= outfin then do;
             call output(chan,ch);
            if outst+1 > outmax then outst = 1; else outst = outst + 1;
            outint = '0'b;
         end;
         else sending = '0'b;
      end;
      if ~sending then do;
         if rdy then do;
               sending = '1'b;
            outint = '0'b;
            ch = outbuff(outst);
            call output(chan,ch);
            if outst+1 > outmax then outst = 1; else outst = outst + 1;
         end;
      end;
end;
end;
end;
```

ADDONE.PLI

```
addone : procedure(row,col);
/* increments array indices of an 80X80 array */
%include 'mextern.pli';
dcl (row,col) fixed;

     if (col ~= 80) then col=col + 1;
     else do;
        if (row~=80) then row = row + 1; else row = 1;
        col = 1;
     end;
end addone;
```

MAXWELL Disc

MONITOR.PLI

```
monitor : procedure options(main);
/* main program which controls monitor and directs its action */
%include 'b:mextern.pli';
dcl (download, templ, start, setbrk) entry,
        lmsdata entry,
        watvar  entry(bit(1));

screel : procedure(choice);
dcl choice fixed;

        call puts(clear);
        call puts('^M^JDo you want to ? ');
        call puts('^M^J        0 Exit from the System');
        call puts('^M^J 1 Start Target System Running ');
        call puts('^M^J        2 Set Breakpoints ');
        call puts('^M^J        3 Set Up Watchlist for Variables ');
        call puts('^M^J        4 Set Up Watchlist for Messages ');
        call getfix('^M^JEnter Choice ',choice,0,4);
end screel;

        on error begin;
           call puts('^M^J error, push return to continue');
           call gets(line);
           goto thend;
        end;

        messptr = null;

        call download;
        call templ;
        disc = '1'b;
        call watvar(disc);
        call watmess(disc);
        disc='0'b;
        choice = 1;
```

```
        do while (choice ~= 0);
           call screel(choice);
           if choice = 1 then call start;
           else if choice = 2 then call setbrk;
           else if choice = 3 then call watvar(disc);
           else if choice = 4 then call watmess(disc);
        end;
thend:
end monitor;
```

TWOPROC.PLI

```
twoproc : procedure options(main);
/* main program for the two processor test. It receives message, */
/* formats it, and puts it in an output buffer for another procedure */
/* to collect */
%include 'extern.pli';
dcl i  fixed;

  inst = 1;
  infin = 1;
  outst = 1;
  outfin = 1;
  do while('1'b);
     line = '^M^JEnter message ';
     call puts(line);
     call gets(line);
     datatype = 1;
     line = '^B'||dest||source||datatype||line||'^C';
     i = 1;
     do while(substr(line,i,1) ~= ' ');
             outbuff(outfin) = substr(line,i,1);
             i = i + 1;
             if outfin+1 > outmax then outfin = 1; else outfin = outfin + 1;
     end;
     rdy = '1'b;
  end;
end;
end;
```

INPROC.PLI

```
inproc : procedure;
/* input interrupt procedure. It receives characters from the bus */
%include 'extern.pli';
dcl (display,send) entry;

   if infin ~=  inst-1 then do;
      call inpt(chan,ch);
      inbuff(infin) = ch;
      if ch = '^C' then call display;
   end;
   if infin+1 > inmax then infin = 1; else infin = infin + 1;
   call send;
   return;
end;
end;
```

OUTPROC.PLI

```
outproc : procedure;
/* output interrupt procedure. Indicates that an output interrupt has been */
/* received by setting outint to true, then calls send */
%include 'extern.pli';
dcl send entry;

   outint = '1'b;
   call send;
   return;
end;
```

DISPLAY.PLI

```
display : procedure;
/* displays message on the screen of the receiving processor */
%include 'extern.pli';
dcl i fixed;
  /* skip header */
  if inst+3 > inmax then inst= 3-(inmax - inst); else inst = inst+3;
  i = 1;
  do while(inbuff(inst) ^= '^C');
     substr(line,i,1) = inbuff(inst);
     if inst+1 > inmax then inst = 1; else inst = inst + 1;
     i = i + 1;
  end;
  if inst+1 > inmax then inst = 1; else inst = inst + 1; /* skip trailer */
  call puts(line);
end;
end;
```

MESSAGE.PLI

```
message : procedure options(main);
/* main program for directing the message test suite */
%include 'mextern.pli';
dcl ch char,
        (i,j) fixed,ADDONE ENTRY(fixed,fixed),
        arpos fixed,
        (logmess,listmess) entry;

        putmess : procedure;
        dcl addone entry(fixed,fixed);

           logcnt = 0;
           arcnt = 0;
           call puts('^M^JEnter Messages^M^J');
           line = ' ';
```

```
            do while (line ~= '.');
               arpos = 0;
               call gets(line);
               if (line ~= '.') then do;
                 arcnt = arcnt + 1;
                 call addone(arcnt,arpos);
                 rxar(arcnt,arpos) = STX;
                 ch = ' ';
                 do while (ch ~= '.');
                    ch = substr(line,1,1);
                    line = substr(line,2);
                    call addone(arcnt,arpos);
                    rxar(arcnt,arpos) = ch;
                 end;
                 call addone(arcnt,arpos);
                 rxar(arcnt,arpos) = ETX;
               end;          /* if */
            end;         /* do */
         end putmess;

         on error begin;
            line = 'out of range';
            call puts(line);
            call gets(line);
            goto ntline;
         end;

         call watmess;
         call putmess;

         ntline:
         call logmess;
         call listmess;
end message;
```

WATMESS.PLI

```
watmess : procedure(disc);
/* asks user what messages are to be logged. The entered info is stored */
/* in a linked list */
%include 'b:mextern.pli';
dcl found bit(1),
   temp char (80) varying,
   num char;

scree6 : procedure;

        call puts(clear);
        line = '^M^JFor each message category, state whether ';
        call puts(line);
        line = 'the message is based on type (T), ';
        call puts(line);
        line = '^M^Jsource (S), or destination (D). Then ';
        call puts(line);
        line = 'enter the integer describing the ';
        call puts(line);
        line = '^M^Jtype/source/destination. ';
        call puts(line);
        line = '^M^JFinally enter (H) to store the header, (D) to store the ';
        call puts(line);
        line = 'header and data, or (B) to cause a breakpoint. ';
        call puts(line);
        line = '^M^Jeg. T/5/H/ says the message is TYPE 5 ';
        call puts(line);
        line = 'and you simply wish to store the header.^M^J';
        call puts(line);
         call puts('^M^JFor the final line simply enter ''.'''^M^J');
end scree6;
```

```
getlet : procedure(let);
   dcl let char;

   temp = substr(line,1,index(line,'/')-1);
   let = substr(temp,1,1);
   if (rank(let)>=97) & (rank(let)<= 122) then
    let = ascii(rank(let)-32);
end;

on error begin;
   line = 'out of range';
   call puts(line);
   goto ntline;
end;

if ~disc then do;
   call scree6;
   call gets(line);
end;
else call getdisc(line);
prev = null;
```

```
do while (line ~= '.');
   allocate messnode set(newmess);
   temp = substr(line,1,index(line,'/')-1);
   call getlet(newmess->type);
   line = substr(line,index(line,'/')+1);
   if ((rank(newmess->type) = 0) &  (prev ~= null)) then
      newmess->type = prev->type;
   temp = substr(line,1,index(line,'/')-1);
   num = substr(temp,1,1);
   if rank(num) = 0 then newmess->id = 0;
   else newmess->id = decimal(substr(temp,1));
   line = substr(line,index(line,'/')+1);
   if (newmess->id=0) & (prev ~= null) then
      newmess->id = prev->id;
   temp = substr(line,1,index(line,'/')-1);
   call getlet(newmess->action);
   line = substr(line,index(line,'/')+1);
   if (rank(newmess->action)=0) & (prev ~= null) then
      newmess->action = prev->action;
   newmess->nextmess = null;
   prev = newmess;
   if messptr = null then messptr = newmess;
   else do;
      found = '0'b;
      newmess = messptr;
      do while ((prev->type >= newmess->type) & (~found)
         & (newmess->nextmess ~= null));
        if (prev->type = newmess->type) & (prev->id <= newmess->id)
         then found = '1'b;
        else do;
           tail = newmess;
           newmess = newmess->nextmess;
        end;
      end;
```

```
            if (newmess = messptr) then do;          /* insert at start */
              prev->nextmess = newmess;
              messptr=prev;
            end;
            else if (newmess->nextmess = null) then  /* insert at end */
              newmess->nextmess = prev;
            else do;                                 /* insert in middle */
              tail->nextmess = prev;
              prev->nextmess = newmess;
            end;
          end;        /* else */

          ntline :
          if disc then call getdisc(line);
          else call gets(line);
        end;        /* while */
end watmess;
```

LISTMESS.PLI

```
listmess : procedure;
/* asks user what messages to list, whether they are to be listed to the */
/* printer or screen and then lists them */
%include 'mextern.pli';
dcl (int,trow,tcol,row,col) fixed,
        temp char(80) varying,
        found bit(1),
        addone entry(fixed,fixed),
        (act,typ) char;

        test : procedure(typ,int,found,act);
        dcl (act,typ) char,
           messtmp pointer,
           int fixed,
           found bit(1);

           messtmp = messptr;
           found = false;
           do while ((~found) & (messtmp ~= null));
              if ((messtmp->type = typ) & (messtmp->id = int))
                  then found = true;
              else do;
                messtmp = messtmp->nextmess;
              end;         /* do */
           end;         /* do */
           if found then act = messtmp->action;
        end test;
```

```
scree8 : procedure(line);
dcl line char(80) varying;

   do while ((substr(line,1,1)~='T') & (substr(line,1,1) ~= 'S') &
      (substr(line,1,1) ~= 'D') & (substr(line,1,1)~= '.'));
      call puts(clear);
      call puts('What category of message do you want listed? ');
      call puts('^M^JEnter message type/integer/ eg. D/5/ for ');
      call puts('messages whose destination is 5^M^J');
      call gets(line);
   end;          /* do */
end scree8;

line = ' ';
do while (line ~= '.');
   call scree8(line);
   if (line ~= '.') then do;
      typ = substr(line,1,1);
      line = substr(line,3);
      int = decimal(substr(line,1,index(line,'/')-1));
      act = ' ';
      call test(typ,int,found,act);
      if (~found) then
        call puts('^M^JNO MESSAGE OF THIS TYPE LOGGED^M^J');
      else do;
         row = 1; col = 1;

         do while ((row <= finrow) & (col <= fincol));
           do while ((messdump(row,col) ~= STX) & (col <= fincol));

               call addone(row,col);
           end;
           found = false;
           if (messdump(row,col)) = STX then do;
               /* looking for typ and int */
               call addone(row,col);
               if (messdump(row,col)=act) then do;
                 call addone(row,col);
                 trow = row; tcol = col;
```

```
                        if ((typ = 'T') & (messdump(trow,tcol) = int)) then
                    else do;
                      call addone(trow,tcol);
                      if (typ = 'S') & (messdump(trow,tcol) = int) then
                          found = true;
                      else do;
                         call addone(trow,tcol);
                         if (typ = 'D') & (messdump(trow,tcol))= int
                           then found = true;
                      end;
                    end;
                  end;
                  if found then do;
                    do while ((messdump(row,col)~=ETX)&
                      (messdump(row,col)~=STX));
                       call puts(messdump(row,col));
                       call addone(row,col);
                    end;
                  end;
               end;        /* do */
            end;        /* do */
         end; /* if */
         call puts('^M^JPush <RETURN> to continue ');
         call gets(temp);
      end;        /* do */
end listmess;
```

LOGMESS.PLI

```
logmess : procedure;
/* messages are stored one per line in rxar */
/* the messages that need to be logged are stored in messdump */

%include 'mextern.pli';
dcl messtmp pointer,
        (hlog,dlog,found) bit(1),
        addone entry(fixed,fixed),
        (i,row,col) fixed,
        bpt entry;

        row = 1; col = 0;
        do while (arcnt ~= logcnt);
           logcnt = logcnt + 1;
           messtmp = messptr; hlog = false; dlog = false;

           do while (messtmp ~= null);

              found = false;
              if ((messtmp->type = 'T') & (messtmp->id =rxar(logcnt,2)))
              | ((messtmp->type = 'S') & (messtmp->id = rxar(logcnt,3)))
              | ((messtmp->type = 'D') & (messtmp->id = rxar(logcnt,4)))
              then found = '1'b;
              if (found)&(((messtmp->action = 'H')&(~hlog))|
                ((messtmp->action ='D')&(~dlog))) then do;
                 call addone(row,col);
                 messdump(row,col) = STX;
                 call addone(row,col);
                 if messtmp->action = 'H' then messdump(row,col) = 'H';
                 else if messtmp->action = 'D' then messdump(row,col) = 'D';

                    if messtmp->action = 'H' then do;
                  hlog = true;
                  do  i = 2 to 4;
                     call addone(row,col);
                     messdump(row,col) = rxar(logcnt,i);
                  end;
```

```
                end;
                if messtmp->action = 'D' then do;
                  i = 2;
                  dlog = true;
                  do while (rxar(logcnt,i) ~= ETX);
                     call addone(row,col);
                     messdump(row,col) = rxar(logcnt,i);
                     i = i + 1;
                  end;
                end;
               tick = 1;         /* just temporarily */
               if (messtmp->action = 'D') |  (messtmp->action = 'H') then do;
                  call addone(row,col);
                  messdump(row,col) = character(tick);
                  call addone(row,col);
                  messdump(row,col) = ETX;
               end;
               if (messtmp->action = 'B') then call bpt;
            end; /* if */
            messtmp = messtmp->nextmess;
         end;         /* while */
      /* could blank a field in rxar to show that message is dealt with */
         end;         /* while */
      finrow = row; fincol = col;
end logmess;
```

LMSDATA.PLI

```
lmsdata : procedure;
/* prints out info stored in the message table. Tests watmess */
%include 'mextern.pli';
dcl pt pointer;

        pt = messptr;
        do while(pt ~= null);
           line = '^M^Jtype'||pt->type||'id'||pt->id||'action'||pt->action;
           call puts(line);
           pt = pt->nextmess;
        end;
        call puts('^M^JPush any key to continue^M^J');
        call gets(line);
end lmsdata;
```

GETDISC.PLI

```
getdisc : procedure(temp);
/* will eventually read information from disc */
%include 'mextern.pli';
dcl temp char(80) varying;

        temp = '.';
        call puts('^M^Jgetting info from disc ');
end;
end getdisc;
```

START.PLI

```
start : procedure;
/* will eventually be loop processed until clock tick occurs */
%include 'b:mextern.pli';

        line = '
entered start';
        call puts(line);
end start;
```

BPT.PLI

```
bpt : procedure;
/* activated when a breakpoint occurs. Will eventually display a message on */
/* the screen asking user what to do next */
%include 'mextern.pli';

        call puts('^M^Jdoing bpt');
end bpt;
```

DOWNLOAD.PLI

```
download : procedure;
/* will be program which downloads the programs through the monitor */
/* into the appropriate processors */
%include 'b:mextern.pli';

        line = '^M^J entered download';
        call puts(line);
end download;
```

MESS.CMD

```
link b:message,b:addone,b:watmess,b:logmess,b:listmess,b:getdisc,b:bpt,b:inter
b:message
```

T.CMD

```
link b:monitor,b:download,b:getdisc,b:templ,b:watvar,b:watmess,b:start,&
b:monitor
```

SETBRK.PLI

```
setbrk : procedure;
/* sets up time or the message table with information from the user */
/* about when next breakpoint should occur */
%include 'b:mextern.pli';

scree2 : procedure(choice);
dcl choice fixed;
        choice = 1;
        call puts(clear);
        call puts('^M^JSet breakpoints based on ');
        call puts('^M^J        1 Time');
        call puts('^M^J        2 Message');
        call getfix('^M^JEnter choice',choice,1,2);
end scree2;

scree3 : procedure(choice);
dcl choice fixed;

        call puts('^M^JHow many ticks before the breakpoint? ');
        call getfix('^M^JEnter number of ticks',choice,1,1000);
end scree3;
```

```
        call scree2(choice);
        if choice = 1 then do;
           call scree3(choice);
           bptime = tick + choice;
        end;
        else  call watmess(disc);
end setbrk;
```

TEMPL.PLI

```
templ : procedure;
/* information on how messages are to be stored is entered from disc */
/* then stored in a table */
%include 'b:mextern.pli';

        call getdisc(line);
        prev = null;
        do while(line ~= '.');
           allocate messtemp set (newtemp);
           newtemp->messtype = decimal(substr(line,1,index(line,'/')-1));
           line = substr(line,index(line,'/')+1);
           if (newtemp->messtype=0) & (prev ~= null) then
              newtemp->messtype = prev->messtype;
           newtemp->length = decimal(substr(line,1,index(line,'/')-1));
           if (newtemp->length = 0) & (prev ~= null) then
              newtemp->length = prev->length;
           newtemp->nextemp = null;
            prev = newtemp;
           if temptr = null then temptr = newtemp;
           else do;
              newtemp = temptr;
              do while ((prev->messtype > newtemp->messtype) &
                (newtemp->nextemp ~= null));
                tail = newtemp;
                newtemp = newtemp->nextemp;
              end;
```

```
            if newtemp = temptr then do;
              prev->nextemp = newtemp;
              temptr = prev;
            end;
            else if (newtemp->nextemp = null) &
              (prev->messtype > newtemp-> messtype) then
              newtemp->nextemp = prev;
            else do;
              tail->nextemp = prev;
              prev->nextemp = newtemp;
            end;
          end; /* else */
          call getdisc(line);
        end; /* while */
end templ;
```

VAR Disc

VARIAB.PLI

```
variab : procedure options(main);
/* main program for the variable test suite. It calls watvar, getvar, */
/* recval and listvars */
%include 'mextern.pli';
dcl          watvar entry(bit(1)),
         (getvar,recval,listvars) entry;

         on error begin;
            call puts('^M^Jerror has occured,push RETURN to continue');
            call gets(line);
            goto theend;
         end;

         disc = false;
         call watvar(disc);
         call getvar;         /* changed so it sends message to the screen */
         call recval;         /* changed so it receives values from the screen */
         call listvars;
theend:
end variab;
```

WATVAR.PLI

```
watvar : procedure(disc);
/* asks user what variables and how many values to log. The data */
/* structure is then constructed for storing these variables */

%include 'b:mextern.pli';
dcl          found bit(1),
        (lastn,n,i) fixed,
        (previnfo,newinfo,temp) pointer;

scree5 : procedure(line);
dcl line char(80) varying;

        call puts(clear);
        call puts('^M^JFor each variable you want to view give');
        call puts('^M^Jname,processor number,address,number of bytes');
        call puts('^M^Jnumber of values logged');
        call puts('^M^JFor the final line simply enter ''.''^M^J');
end scree5;

        on error begin;
           line = 'out of range';
           call puts(line);
           goto ntvar;
        end;

        if ~disc then do;
           call scree5;
           call gets(line);
        end;
        else call getdisc(line);
        prev = null; lastn = 0;
        do while (line~= '.');
           allocate varnode set(newvar);
           newvar->name = substr(line,1,index(line,'/')-1);
           line = substr(line,index(line,'/')+1);
           if ((newvar->name = ' ') & (prev ~= null)) then
              newvar->name = prev->name;
```

```
newvar->prnum = decimal(substr(line,1,index(line,'/')-1));
line = substr(line,index(line,'/')+1);
if (newvar->prnum=0) & (prev ~= null) then
   newvar->prnum = prev->prnum;
newvar->address = decimal(substr(line,1,index(line,'/')-1));
line = substr(line,index(line,'/')+1);
if (newvar->address = 0) & (prev ~= null) then
   newvar->address = prev->address;
newvar->numbytes = decimal(substr(line,1,index(line,'/')-1));
line = substr(line,index(line,'/')+1);
if (newvar->numbytes = 0) & (prev ~= null) then
   newvar->numbytes = prev->numbytes;
n = decimal(substr(line,1,index(line,'/')-1));
if n = 0 then n = lastn; lastn = n;
newvar->nextvar = null;
prev = newvar;
if varptr = null then varptr = newvar;
else do;
   found = '0'b;
   newvar = varptr;
```

```
    do while ((prev->prnum >= newvar->prnum) & (~found)
     & (newvar->nextvar ~= null));
       if (prev->prnum = newvar->prnum) &
        (prev->address<=newvar->address) then found = '1'b;
       if ~found then do;
             tail = newvar;
          newvar = newvar->nextvar;
       end;
    end;
    if (newvar = varptr) & (prev->prnum < newvar->prnum) then do;
       prev->nextvar = newvar;        /* insert at start */
       varptr = prev;
    end;
    else if (newvar->nextvar = null) then  /* insert at end */
      newvar->nextvar = prev;
    else do;                /* insert within list */
      tail->nextvar = prev;
      prev->nextvar = newvar;
    end;
end;
if (n>0) then do;
    allocate infonode set(temp);
    prev->infoptr = temp;
    previnfo = prev->infoptr;
    previnfo->info = 0;
    i = 1;
    do while (i<n);
       allocate infonode set(newinfo);
       previnfo->nextinfo = newinfo;
       previnfo = newinfo;
       previnfo->info = 0;
       i=i+1;
    end;
    previnfo->nextinfo = prev->infoptr;        /* make ring */
    prev->current = prev->infoptr;        /* indicates where to put
                                          next value */
```

```
            ntvar :
            if disc then call getdisc(line);
            else call gets(line);
          end;
        end;        /*while */
end watvar;
```

GETVAR.PLI

```
getvar : procedure;
/* sends message out to processors for each of the variables in the list */
/* asking them to send the values of a particular variable */
%include 'mextern.pli';
dcl message char(80) varying,
        temp fixed;
        newvar = varptr;
        do while (newvar ~= null);        /* put message in buffer */
            message = STX||vartype||newvar->prnum||monitornum||newvar->address;
          message = message||ETX;
/*          temp = length(message);
          call send(message,temp);
*/
message ='^M^J'||message;
call puts(message);
          newvar = newvar->nextvar;
        end;
end getvar;
```

RECVAL.PLI

```
recval : procedure;
/* used only for test purposes. */
/* receives the value from the message and places it in a pointer table */
%include 'mextern.pli';
dcl found bit(1),
        message char(80) varying,
        (ntinfo,temp,valtmp) pointer;
message =' ';
call puts('^M^JType in values^M^J');
```

```
do while (message ~= '.');
call gets(message);
if (message ~= '.') then do;
        found = false;
        temp = varptr;
        do while ((~found) & (temp ~= null));
           found = ((temp->prnum=decimal(substr(message,4,1)))&
              (temp->address=decimal(substr(message,5,1))));
           if ~found then temp = temp->nextvar;
        end;
        if found then do;
           valtmp = temp->current;
           valtmp->info = decimal(substr(message,6,1));
           temp->current = valtmp->nextinfo;
        end;
end;
end;
end;        /* recval */
```

LISTVARS.PLI

```
listvars:procedure;
/* lists all the variables which the user has requested to be logged */
%include 'mextern.pli';
dcl (proc,add,where) fixed,
        printvar entry(pointer,fixed);

        scree9 : procedure(choice,proc,add,where);
        dcl (choice,proc,add,where) fixed;

           call puts(clear);
           call puts('^M^JWhich variables do you want listed?');
           call puts('^M^J1        by processor');
           call puts('^M^J2        address within processor');
           call puts('^M^J3         all');
           call getfix('^M^JEnter choice',choice,1,3);
           line = ' ';
           proc = 0;add = 0;
```

```
            if choice = 1 then do;
               call puts('^M^JEnter processor number ');
               call gets(line); proc = line;
            end;
            else if choice = 2 then do;
               call puts('^M^JEnter processor number/address');
               call gets(line);
               proc = decimal(substr(line,1,index(line,'/')-1));
               line = substr(line,index(line,'/')+1);
               add = decimal(substr(line,1,index(line,'/')-1));
            end;
            call getfix('^M^JList at 1 screen, or 2 Printer',where,1,2);
         end scree9;

         call scree9(choice,proc,add,where);
         newvar = varptr;
         do while (newvar ~= null);
            if choice = 3 then call printvar(newvar,where);
            else if (choice = 1) & (newvar->prnum=proc) then
               call printvar(newvar,where);
            else if (choice=2) & (newvar->prnum=proc) & (newvar->address=add)
               then call printvar(newvar,where);
            newvar = newvar->nextvar;
         end;
end listvars;
```

PRINTVAR.PLI

```
printvar : procedure(newvar,where);
/* called from listvar. Prints information about variable then lists values */
%include 'mextern.pli';
dcl where fixed,
         print entry(char(80) varying),
         (previn,newinfo) pointer;

         output : procedure(line);
         dcl line char(80) varying;

            if where = 2 then call print(line); else call puts(line);
         end output;
```

```
        line = '^M^J'||newvar->name||newvar->prnum||newvar->address;
        call output(line);
        newinfo = newvar->current->nextinfo;
        previn = null;          /* introduced so final value can be printed */
        do while (previn ~= newvar->current);
           if (newinfo->info >= 0) & (newinfo->info <= 9) then do;
              line = newinfo->info;
              previn = newinfo;
              call output(line); newinfo = newinfo->nextinfo;
           end;
        end;
end printvar;
```

LVAR.PLI

```
lvar : procedure;
/* lists the variables to be logged. Tests watvar */
%include 'mextern.pli';
dcl pt pointer;

        pt = varptr;
        do while (pt ~= null);
           line = '^M^Jname '||pt->name||' prnum '||pt->prnum;
           call puts(line);
           line = ' address '||pt->address||' numbytes '||pt->numbytes;
           call puts(line);
           pt = pt->nextvar;
        end;
        call puts('Push any key to continue');
        call gets(line);
end lvar;
```

VAR.CMD

```
link b:variab,b:getvar,b:recval,b:listvars,b:printvar,b:watvar,b:getfix,&
b:variab
```

TESTREC.PLI

```
testrec : procedure options(main);
/* tests that receive procedure works */
%include 'mextern.pli';
dcl (putcyc,arpos) fixed,
   ch char;

   on error begin;
      call puts('^M^Jerror ');
      call gets(line);
      goto thend;
   end;

   call puts('^M^JType in messages, one/line, each ending with ''.''');
   call puts('^M^JThe final line should contain only ''.''^M^J');
   call gets(line);
   putcyc = 0; rxchcnt = 0;
   do while (line ~= '.');
      if (putcyc < 200) then putcyc = putcyc + 1; else putcyc = 1;
      rxcyc(putcyc) = STX;
      call add(rxchcnt);
      ch = substr(line,1,1);
      do while (ch ~= '.');
         line = substr(line,2);
         if (putcyc < 200) then putcyc = putcyc + 1; else putcyc = 1;
         rxcyc(putcyc) = ch;
         ch = substr(line,1,1);
         call add(rxchcnt);
      end;
      if (putcyc < 200) then putcyc = putcyc + 1; else putcyc = 1;
      rxcyc(putcyc) = ETX;
      call add(rxchcnt);
      call gets(line);
   end;
   call receive;
```

```
   call puts('^M^JMessages in rxar are ');
   logcnt = 0;
   do while (logcnt < arcnt);
      call puts('^M^J');
      logcnt = logcnt + 1;
      arpos = 2;
      do while (rxar(logcnt,arpos) ~= ETX);
         call puts(rxar(logcnt,arpos));
         arpos = arpos + 1;
      end;
   end;
thend:
end testrec;
```

TESTSEND.PLI

```
testsend : procedure options(main);
/* tests that send procedure works */
%include 'mextern.pli';
dcl loopcnt fixed;
        sxchcnt = 0;
        call puts('^M^JEnter lines, finishing with ''.'''^M^J');
        call gets(line);
        do while ((line ~= '.') & (length(line)>0));
           call send(line,length(line));
           call gets(line);
        end;
        loopcnt = 0;
        do while (loopcnt < sxpos);
           call puts('^M^J');
           do while (sxcyc(loopcnt) ~= ETX);
              call sub(sxchcnt);
              call puts(sxcyc(loopcnt));
              if (loopcnt < 200) then loopcnt = loopcnt + 1; else loopcnt =1;
           end;
        end;
        line = '^M^Jsxchcnt is '||sxchcnt;
        call puts(line);
end testsend;
```

TEST.PLI

```
test : procedure options(main);
/* tests to see if the assembler routines add and sub are working */
%include 'mextern.pli';
dcl (testcnt,i) fixed;

        testcnt = 0;
        do i = 1 to 4;
           call add(testcnt);
        end;
        line = '^M^Jvalue of testcnt is '||testcnt;
        call puts(line);

        do i = 1 to 3;
           call sub(testcnt);
        end;
        line = '^M^Jvalue of testcnt is '||testcnt;
        call puts(line);
end test;
```

```
receive : procedure;
/* receive message from circular buffer in the comms handler */
%include 'mextern.pli';
dcl        arpos fixed;

        do while (rxchcnt ~= 0);
           arcnt = arcnt + 1;
           arpos = 1;
           do while (rxcyc(cycpos) = ETX);
              if (cycpos < 200) then cycpos = cycpos + 1; else cycpos = 1;
              call sub(rxchcnt);
           end;

           do while (rxcyc(cycpos) ~= ETX);
              rxar(arcnt,arpos) = rxcyc(cycpos);
              if (cycpos < 200) then cycpos = cycpos + 1;
              else cycpos = 1;
              if arpos < 200 then arpos = arpos + 1; else arpos = 1;
              call sub(rxchcnt);
           end;
           rxar(arcnt,arpos) = ETX;
        end;
end receive;
```