University of
Waterloo Research Institute

PERFORMANCE MEASUREMENT IN COMPUTER NETWORKS

D.E. MORGAN

OFFICE OF RESEARCH ADMINISTRATION

UNIVERSITY OF WATERLOO

INCORPORATING THE

WATERLOO RESEARCH INSTITUTE

PROJECT NO. 3046-2

(2)

PERFORMANCE MEASUREMENT IN COMPUTER NETWORKS

SPONSORED BY

Department of Communications

Under

Department of Supply and Services
Contract Number OSU 4-0098

(1)

D.E. Morgan

March 31, 1975

1. Introduction

1.1 Computer Network Monitoring System

On 2 October 1974, at a panel discussion of the joint meeting of the ACM Special Interest Group on Measurement and Evaluation (SIGMETRIC) and Boole and Babbage Users' Group, leaders in the computer measurement field stated that since a number of organisations were forming computer networks, there was a need for a system of hardware and software to observe the activities of these networks. It soon became evident that we were the only people who had attempted to design and implement such a system, although a few other organisations, such as Comress and Tesdata, are hoping to produce suitable monitoring systems within one to two years.

Most measurements that have been made of computer networks have been made using software techniques (e.g., see <1,2>). Such software can and does interfere with network activities and produces significant inaccurracies in the measurements. Moreover, measurement traffic on the network can distort traffic statistics.

The system of special hardware and software we have been creating for the past three years is designed to observe the activities of a computer network while interfering minimally with them. This Computer Network Monitoring System (CNMS) is described in Appendices A, B, and C.

1.2 Computer Network Dependability Research

More and more organisations are realizing that an unreliable computer system is not cost-effective, regardless of how quickly it executes or how efficiently it handles resources. Enhanced dependability is one reason often given for building networks of computers. We are investigating the possibility of using the Computer Network Monitoring System as a tool for enhancing the dependability of a computer network. We are devising an automated maintenance system for computer networks which is an integrated system composed of a number of tools for achieving dependability.

Bell Laboratories has devised a variety of tools and techniques for enhancing the dependability of electronic switching systems. These are described in Bell Systems Technical Journals of 1964, 1969, 1970, and 1973. , and in <3,4>. A fairly extensive bibliography of computer system dependability studies is included in Appendix E. Appendices E and F survey the fields of computer system and network dependability.

2. Purpose

The objectives of this research are:

1. To provide an easily used, yet powerful computer network monitoring system. Experience indicates that neither hardware nor software alone are completely satisfac-

tory; thus, a combination is sought.

2. To learn how to provide a flexible, easily used network maintenance system that facilitates rapid detection, diagnosis, and recovery from network troubles, whether malfunctions or bottlenecks.

3. Summary of Methods

For the past four years, we have been designing, implementing, testing, and, within the past year, we have been using a prototype of a system of special hardware and software for monitoring a computer network (or computer system). Called a Computer Network Monitoring System (CNMS) and described in Appendices A, B, C, the system consists of:

(1.) A set of software-controlled hardware monitors (often called hybrid monitors), each of which is attached by its probes to a computer and associated data links of the network to be monitored. Each monitor can be controlled by a remotely-located computer via a telecommunications link.

(2.) Software to control the monitors and analyse the data.

(3.) Software to generate traffic for the object network (i.e., the network to be monitored), so that measurements of the network's activities can be made as it responds to known stimuli.

(4.) A minimal amount of measurement software in

each system of the object network.

Our philosophy is to use hardware to monitor that which is best observed by hardware, to use software for that for which it is best suited, and to use a combination where neither is best.

Telephone lines, normally different from those of the object network connect the monitors to the controlling computer. Each monitor consists of one or more of each of the specially-designed components listed in Table 3.1. They are joined by a single bus (called a MONIBUS) that is attached to the controlling computer.

Figure 3.1 illustrates the interconnection of these components to form the monitor. Figure 3.2 illustrates the interconnection of the monitors to observe the activities of a computer network. Note that the communications lines of the network are not used to transmit measurement data or monitor control information. Rather, the controlling computer uses switched voice-grade lines to set up the measurement experiment, then disconnects while the measurements are made. Periodically, the connection is re-established while data is collected and/or additional control information is transmitted. This technique reduces communications costs in measuring geographically distributed networks. For a more complete description of the CNMS, including the software structure, see Appendix C.

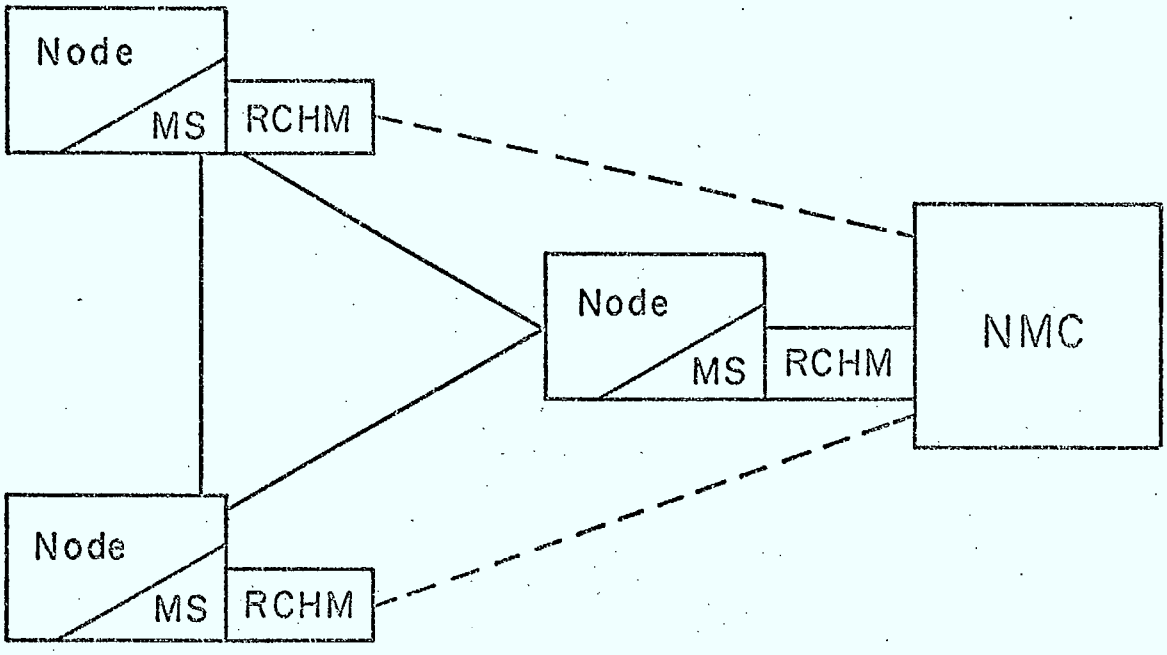


Fig. 3.1

MS - Measurement Software

RCHM - Remote Computer Controlled Hardware Monitor

RNMC - Regional Network Measurement Centre

NMC - Network Measurement Centre

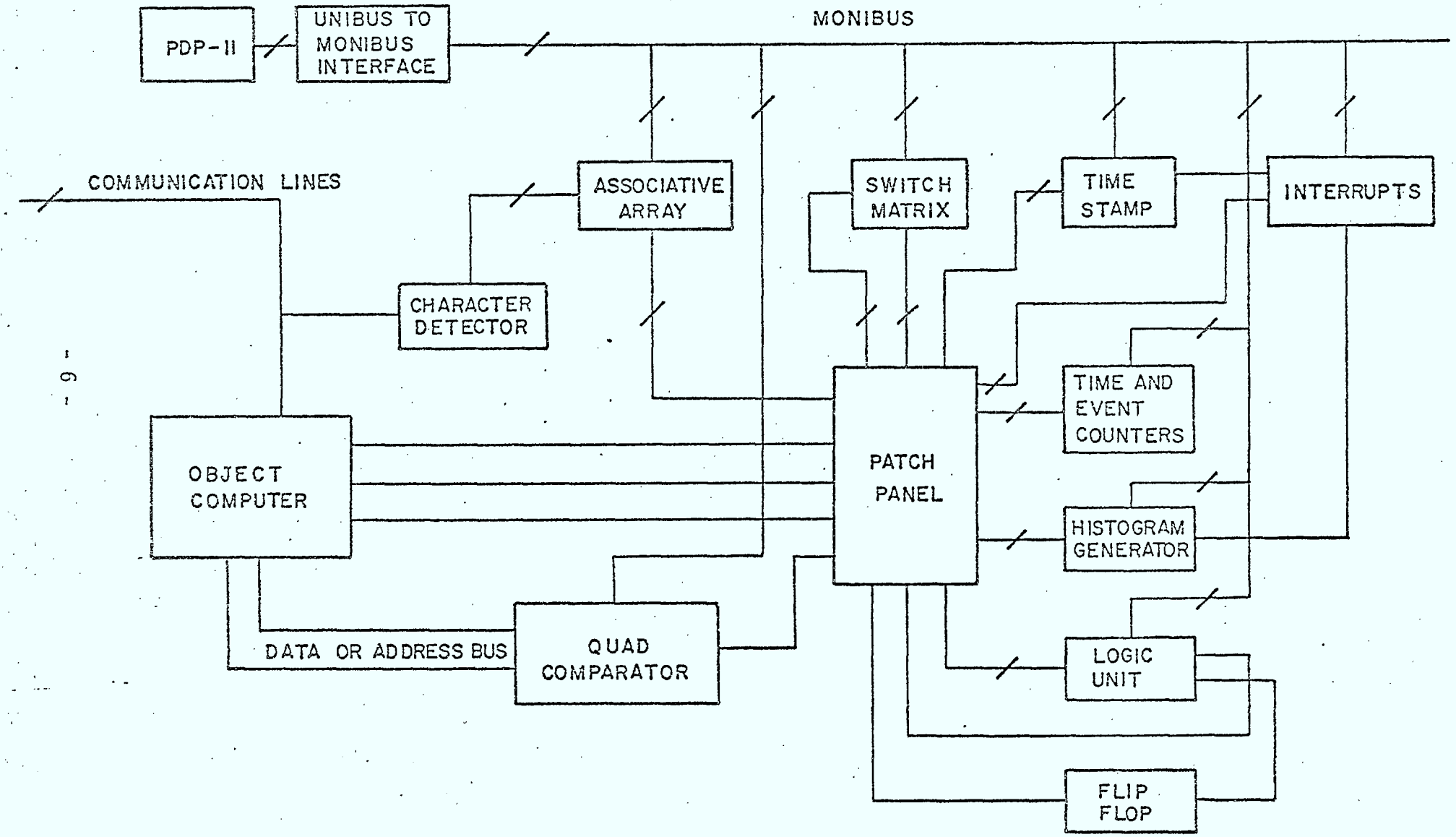


Fig. 3.2 GENERALIZED MONITOR

CNMS component	Hardware status			Status of associated software			In use?
	Design	Implementation	No. Built	Design	Implementation	Testing with hardware	
1. RCHM and components	Complete	In progress	2	DOS-11 version complete; RSX-11 version in progress	DOS-11 version nearly complete	DOS-11 version nearly complete	DOS-11 version in use
a. Timer and event counter	complete	complete	10	complete	complete	complete	yes
b. Combinational logic unit	complete	complete	6	complete	complete	complete	yes
c. Sequential logic unit	version 1 complete; version 2 in progress	version 1 being built	1	nearly complete	in progress	-	no
d. 16x4 Switch matrix	complete	complete	4	complete	complete	complete	yes
e. 8x8 Switch matrix	complete	complete	8	complete	complete	complete	yes
f. Quadri-comparator	complete	complete	3	complete	complete	complete	yes
g. Single comparator	complete	complete	1	complete	complete	under way	no
h. Interrupt generator	complete	complete	6	complete	complete	complete	yes
i. Interval timer	complete	complete	1	complete	complete	under way	no
j. Time stamp unit	complete	complete	1	complete	complete	nearly complete	no
k. Histogram generator	complete	complete	1	complete	complete	under way	no
l. Character detector	complete	complete	1	complete	complete	under way	no
m. Monitor Diagnostic Aid	complete	complete	1	complete	complete	nearly complete	partly

Table 4.1

CNMS component	Hardware status			Software status			In use?
	Design	Implementation	No. Built	Design	Implementation	Testing with hardware	
n. Network clock	In progress	-	0	-	-	-	no
o. Probes for PDP-11/20	complete	complete	2 sets	complete	complete	complete	yes
p. Probes for PDP-11/45	complete	started	0	in progress	-	-	no
q. Probes for DR-11A & DC-11 communic. links	complete	complete	2 sets	complete	complete	complete	yes
2. NMC software (DOS-11 version)	-	-	N/A	complete	nearly complete (see below)	in progress (see below)	yes
a. Experiment Manager (DOS-11 version)	N/A	N/A	N/A	complete	nearly complete	in progress	yes
b. Maintenance Manager (DOS-11 version)	N/A	N/A	N/A	complete	version 1 is complete; version 2 in progress	version 1 is complete	yes
c. Results Manager (DOS-11 version)	N/A	N/A	N/A	nearly complete	nearly complete	in progress	yes
d. Communications Manager	N/A	N/A	N/A	nearly complete	in progress	-	no
3. NMC software (RSX-11D version)	N/A	N/A	N/A	in progress	-	-	no
4. RCHM controller (PDP-11/10 & interface)	complete	in progress	1	in progress	-	-	no
5. RCHM controller (LSI-11 & interface)	in progress	-	0	in progress	-	-	no

Table 4.1 (continued)

CNMS component	Hardware status		No. Built	Software status			In use?
	Design	Implementation		Design	Implementation	Testing	
6. Measurement Language	N/A	N/A	N/A	versions 1 & 2 are complete; version 3 is a dream	version 1 is complete; version 2 is in progress	version 1 is complete	version 1 is in use
7. Load generator	N/A	N/A	N/A	version 1 is complete; version 2 is in progress	version 1 is complete	version 1 is complete	version 1 is in use
8. Measurement software	N/A	N/A	N/A	In progress	-	-	no
9. Analysis software	N/A	N/A	N/A	version 1 is complete; version 2 is in progress	version 1 is being implemented	version 1 is in progress	no

Table 4.1 (continued)

4. Status

One monitor and a good first generation of software have been implemented, tested, and are being used. A second monitor has been assembled and is being tested. We are nearly ready to begin using it for a series of experiments, which are scheduled to begin in May 1975.

Table 4.1 summarizes the status of software and hardware aspects of the CNMS project as of 31 March 1975. As the table indicates, we have used at least one version of each of the components of the Basic Monitor, and we are now emphasizing testing an important set of extensions.

During the rest of 1975 we plan to complete implementation and testing of a useful version of each component of the CNMS. Meanwhile we shall continue using tested portions of the CNMS to monitor computer systems and networks in our laboratory. So far we have used the prototype CNMS to monitor several aspects of a PDP-11/20 and some aspects of two small laboratory networks.

Appendix D illustrates some of the types of experiments that we have been performing using the CNMS.

We have been working with E. Gelenbe and his group at IRIA in France to verify some of his mathematical models for operating systems and computer networks. We have also been working with Louis Pouzin and his group at IRIA who are building the CIGALE and CYCLADES computer networks. He has asked us to study the feasibility of monitoring the CIGALE

network using our CNMS.

Appendices E and F are two research reports that reflect most of our work so far in our effort to discover tools and techniques to enhance the dependability of computer networks and operating systems. Because the network maintenance system we are designing uses the CNMS as one of its principal components, for the past year we have emphasized understanding the problems, tools, and techniques involved with achieving a dependable system or network.

Several reports have been produced during the past year, and we have given a number of invited presentations. These reports and presentations are listed at the end of this report. Appendices A through F are six of these reports.

Considerable interest has been expressed in this project by a number of organisations, including manufacturers of monitoring equipment such as Tesdata and Comress; computer manufacturers such as DEC, Honeywell, Hitachi; industries such as Weyerhauser, and U.S. Government agencies such as Lawrence Livermore Laboratories, USAF FEDSIM, and the National Bureau of Standards. The interest shown ranges from wanting to use the system to wanting to manufacture and market it, depending on the organisation. Tesdata, Consolidated Computer, Interactive Business Logic Ltd., and ESE Ltd., are actively considering manufacturing, marketing, and providing a commercial monitoring service based on the

system. Interactive Business Logic has submitted a proposal to Canada Development Corporation for funding to establish three monitoring centres and provide a commercial monitoring service remotely. Tesdata has proposed to integrate the monitor into their product line.

5. Summary of Conclusions

We have drawn a few conclusions based on our work on this project:

1. It is feasible to build and use such a computer network monitoring system. However, with the cost of each monitor ranging from about $10,000 for the Basic Monitor to about $100,000 for a fully extended version, the CNMS seems rather expensive, though not when compared with the cost of other monitors.

2. The system is not yet as easily used as it should be. The worst problem is the complex patch panel wiring required to set up a number of experiments so that one can switch from one experiment to another without changing the wiring. We have made some modifications to the hardware that eliminates the need for many of these patch panel wires.

3. The modular hardware and software architecture we have followed have made it extremely easy to change the system.

4. The maintenance software we have included in the

CNMS makes it easy to determine whether the CNMS is working and to quickly perform an experiment to get a rough idea of what the object system is doing.

5. The interest exhibited so far in our CNMS indicates that there is a need for such a system.

6. A simple, yet rather powerful hybrid monitor can be built from a set of high-speed content-addressable memory (CAM), a slow-speed CAM, a random-access memory (RAM), an array of high performance specialized processors, some switch matrices and a bus to interconnect these components and join them to the controlling computer.

6. Recommendations

Dr. C. D. Shepard, who has served as contract officer for the Department of Communications contract which has supported much of the cost of the CNMS research, thinks that the technology of the Basic Monitor is now ready to be transferred to industry to develop the system into a product or set of products. Therefore, we have been actively seeking Canadian organisation(s) who are interested in making use of this technology. As mentioned in Section 4, four firms are actively attempting to find ways of exploiting this technology.

We would be happy to demonstrate the system to representatives of the Canadian government. There appear to

be numerous potential applications of the CNMS in the Canadian government. As but one example, the Department of National Defence might find the system useful and informative to monitor the SAMSON network once it is implemented. Monitoring can be a useful tool for acceptance tests for a computer system or network.

Future Research

We thank the Department of Communications for their generous support of the project for the past three years. We are especially grateful for the encouragement and help that Dr. C.D. Shepard has provided as project officer for DOC, and for the support given us by Dr. D. F. Parkhill and R. Tanner.

Although this is the final report for this Department of Communications research contract, the research aspects of the CNMS and its use are far from complete. Furthermore, much work remains to be done in order to develop the system into a marketable, usable product. The research problems that immediately come to mind fall in four categories:

A. Research to complete the capabilities of the basic CNMS, e.g.:

1. The CNMS itself is a network of computers, and requires a distributed operating system to manage its resources. The design and performance of such a software system are challenging research

problems.

2. We have completed the design of one version of sequential logic unit for the monitor, but a content-addressable memory version seems to be a better, though possibly more expensive, way of building such a unit. The software to translate from a regular expression to the table to control the unit is also a research problem involving the analysis of algorithms and automata theory. We are working on these two research problems.

B. Extending the CNMS c B. Create techniques for using the tool:

1. There are no satisfactory answers as to how to characterise the workload of a system or network nor the behaviour of the network in response to the applied workload. Many measures are used, but there are no standard definitions nor interpretations for them. We are actively engaged in research in this area.

2. Once measures have been defined, techniques need to be created to provide these measures and represent them in a meaningful and useful way.

C. Use the CNMS to evaluate systems and networks:

1. A number of ideas for building computer networks are being tried experimentally in the laboratory of the Computer Communications Networks Group at the University of Waterloo. In order to determine the relative merits of these techniques, we need to measure and evaluate these experimental networks using the CNMS. Examples include Mininet, the network simulator network, the coding simulator network, the Newhall loop experiment, the hardware packet switch, not to mention the CNMS itself.

2. Louis Pouzin of IRIA has discussed with us the possibility of monitoring the CIGALE/CYCLADES computer network of France.

D. Apply the CNMS to other problems:

1. As discussed in this report, we are exploring the possibility of using a modified version of the CNMS to monitor the behaviour of a computer network to detect malfunctions by observing degradations in performance and program logic sequences.

2. An operating system for a computer system or network that is capable of adapting its scheduling policies to its observed workload is a possibility

that has been shown to have exciting potential. We are investigating the possibility of building such an adaptive operating system based on UNIX of Bell Telephone Laboratories plus a hybrid monitor based on that of the CNMS.

PUBLICATIONS LIST

1. "A Computer-Controlled Hardware Monitor: Hardware Aspects", Proc. A.I.M. Conference on Minicomputers and Data Communication, Liege, Belgium, Jan. 1975; with W. Banks.

2. "A Performance Measurement System for Computer Networks", Proc. of IFIP 74, Stockholm, Sweden, Aug. 1974; with W. Banks, W. Colvin and D. Sutton.

3. "Suitable Local Transformations in Computer Networks", Proc. of 24th Symposium of ASOVAC (Venezuelan Assoc. of Sciences), July 1974; with J. Araoz-Durand.

4. "A Computer Network Monitoring System", submitted for publication in IEEE Transactions on Computers; with W. Banks, D. Goodspeed and R. Kolanko.

5. "Software of the Network Monitoring System: Control of the Combinational Logic Unit," D. Goodspeed, W. Colvin and D.E. Morgan (External Report E-1).

6. "Switching Matrices for Programmable Time-Division Multiplexing," E. Manning, W.M. Gentleman, C.E. Køhn and D.E. Morgan (External Report E-6).

7. "A Performance Measurement for Computer Networks," D.E. Morgan, W. Banks, W. Colvin and D. Sutton (External Report E-18).

8. "A Computer Network Monitoring System," D.E. Morgan, W. Banks, D. Goodspeed and D. Sutton (External Report E-21).

9. "The Monitoring of Computer Systems and Networks: A Summary and Proposal," D.A. Sutton and D.E. Morgan (External Report E-22).

10. "An Automated Maintenance System for Computer Networks," D.E. Morgan and G.F. Clement (Internal Report I-2).

APPENDIX A

A PERFORMANCE MEASUREMENT SYSTEM FOR COMPUTER NETWORKS

INFORMATION PROCESSING 74 – NORTH-HOLLAND PUBLISHING COMPANY (1974)

A PERFORMANCE MEASUREMENT SYSTEM FOR COMPUTER NETWORKS*

D. E. MORGAN, W. BANKS, W. COLVIN and D. SUTTON

University of Waterloo
Waterloo, Ontario, Canada

A system of special hardware and software for monitoring the activities of a network is described. It consists of (1) a hardware monitor controlled by a locally or remotely located computer; (2) monitor control and data analysis software; (3) a network traffic generator; (4) measurement software in each computer measured. Each computer to be measured is attached to a monitor. Telephone lines, different from those of the network, connect the monitors to the controlling computer.

Each monitor consists of a bus and selected event detectors, time measuring components, data recording devices, and communications and control components.

A high-level measurement language is being developed to facilitate controlling the measurements and analyzing the data.

1 INTRODUCTION

A network of computers consists of two or more computers linked together, while a computer network can be either a network of computers, or a set of terminals connected to one or more computers. Most networks of computers consist primarily of nodes, hosts, transmission links, and terminals. A node (in this context) usually refers to a computer used principally to switch data. A computer whose primary role is not switching data in the network to which it is attached, is called a host. In some networks, a sharp distinction is made between nodes and hosts, while in others no distinction exists. Terminals are devices which serve as the interface between man and the computer. The transmission links, of course, join this collection of hardware together to form a network.

The problem considered in this paper is how to monitor a computer network. Four fundamental reasons for monitoring a network are:

(i) To see how well it performs;
(ii) To discover why it performs as it does and to learn how and where to change network hardware and/or software to improve its performance;
(iii) To detect trouble and aid in diagnosing its cause so that appropriate corrective or recovery actions can be taken;
(iv) To charge users of the network's services for the network resources used.

R. W. Hamming of Bell Labs is credited with saying that the goal of measurement is insight, not numbers. Depending on the type of network and the reasons for monitoring it, several different aspects of it can be monitored. Table 1.1 lists many of these aspects of possible interest. For some aspects, the desired insight can best be gained by analyzing distribution functions; for other aspects, studying a set of numbers is sufficient.

Often the data for several nodes of the network must be analyzed as a whole in order to have the perspective necessary to gain the required insight. In such cases the measurement activities should be distributed across the network, yet controlled and coordinated from a measurement centre rather than occur independently at each node. The resulting data could be transmitted to the measurement centre either via the network or through physically (or logically) separate facilities. Unlike a computer system, the

*Research supported by Department of Communications of Canada, research contract no. SP2-36100-3-0406; Defence Research Board of Canada, grant no. 9931-37; National Research Council of Canada, grant no. A8116 and by Nordata Ltd. The research was performed in the Computer Communications Networks Group's laboratory at the University of Waterloo.

Table 1.1

(i) Time measurements
- a. Time required to set up a logical or physical path through the network or through a node;
- b. Time required to disconnect a logical or physical path through network or node;
- c. Time required to transmit a message (or packet) through network, node, selected components of the switch, or transmission link (often called message delay);
- d. Time required for certain components of the network to detect, correct, and/or recover from trouble in the network, e.g. data transmission error, link, host or node out of service;
- e. Time required to detect and/or take appropriate action for network overload;
- f. Time required to respond to a request for service;
- g. Time between arrivals of messages (or packets);
- h. Time required to disassemble (or reassemble) a message into (from) a set of packets;
- i. Amount of time software and hardware resources are utilized;
- j. Amount of time logical or physical path is utilized.

(ii) Space and Time measurements
- a. Auxiliary storage space used in network or selected node(s);
- b. Main storage space used in network or selected node(s);

(iii) Event counts
- a. Number of messages (or packets) handled by node, network, link or host;
- b. Number of bits transmitted and received by node, network, link, or host;
- c. Number of requests for service.

(iv) Length measurements
- a. Number of items selected in queue(s);
- b. Number of bits or characters in each message or packet;
- c. Amount of data to be stored in main storage;
- d. Amount of data to be stored in auxiliary storage.

components of a computer network are often widely separated, sometimes by thousands of miles. Thus, monitoring a network poses communications problems not encountered when monitoring a computer system. A challenging problem is to coordinate monitoring activities across the network and then collect the resulting data for analysis.

1.1 Requirements of a Network Monitoring System

Experience monitoring computer systems and study of pertinent literature indicate that an ideal network monitoring system should possess the following characteristics:

(i) Be easy to use, yet flexible and expandable;
(ii) Be as system independent as practical;
(iii) Interfere minimally with the performance and integrity of the measured system;
(iv) Interfere minimally with computer-computer and terminal-computer communications;
(v) Be dependable and easily diagnosed;
(vi) Offer a choice of resolution, so that the unit of measure fits what is measured;
(vii) Allow gathering of measurement data at a distance from the monitor control and analysis functions, with minimal human intervention;
(viii) Span the network;
(ix) Be low in cost while not compromising other goals.

The problem, then, is to create a network monitoring system having as many of these characteristics as practical while satisfying the four reasons for monitoring a network.

Experience indicates that monitoring software without hardware aid in each node often perturbs the network unsatisfactorily. Hardware monitors without software aid are too inflexible for most network applications. Thus, there is a need for a set of software-controlled hardware monitors, each of which can be attached to a node of the network to be monitored. To achieve network-wide perspective and insight, the activities of these monitors should be centrally controlled and coordinated. Because of the wide geographic spread of many networks, such a network monitoring system should include the ability to have control information and monitored data transmitted via telecommunications links. Furthermore, some items currently can only be monitored economically by software within each node, so a means of controlling and receiving data from node measurement software must also be provided.

2. CURRENT STATE OF THE ART

To the best of our knowledge no such network monitoring system has been designed. Although a great deal of effort and study has been devoted to the creation of hardware and/or software to monitor a computer system (see for example,[1-11]), so far very little work has been devoted to creating techniques and tools for monitoring a computer network. [12]

Computer network monitoring today is accomplished by placing software in each node and using the transmission and switching facilities of the network to send control information to the nodes and monitored data to a Network Measurement Centre. This is the technique used by Prof. Kleinrock and his staff at UCLA to monitor the ARPA network. [12]

Although a few hardware monitors have been designed to have the ability to have their monitoring activities changed under control of software [5,10,13,14, 15], most monitors are patch-board programmable. There seems to be no hardware monitor that includes the ability to be controlled from a remotely-located computer, nor to have its activities coordinated with the activities of several other monitors. Several organizations have used one computer to monitor the activities of another.[9,14,14] In such systems, probes attached to the monitored (or object) computer are connected to registers of the monitoring computer with no pre-processing to reduce the data. To reduce communications costs, some pre-processing is essential.

Our studies indicate that the following monitors would appear to require complete redesign in order to be used to monitor a network of computers:

Program Monitor (1962,IBM) [1,11]
POEM (1963,IBM) [11]
Execution Plotter (1965, IBM) [11]
CPM,CPM II, and CPM III (1968, Applied Computer Technology) [6,15,16,17]
CPA (1968, Computer and Program Analysis, Inc.) [15,16,18]
Event Monitor (1970, Boole and Babbage) [6]
Dolby (1970, Dolby) [15]
CMSM (1970, Copac, Inc.) [6]

The following monitors would require major modifications of the design, usually including the ability to be controlled by a processor plus a synchronizable clock, better resolution, more comparators, and/or more data storage:

Hardware Monitor (1961, IBM) [11,19]
Channel Analyzer (1962, IBM) [11]
PEC (1964, IBM) [11]
SAMI (1967, IBM) [11]
TS/SPAR (1967, IBM) [20]
BCU (1968,IBM) [11]
System Monitor (1968, IBM) [3]
SUM (1968 Computer Synectics, Inc.)[6,11,15,16 18]
UNIVAC Instrument (1968, UNIVAC) [4,21]
GDM (1968, Project MAC, MIT) [14,15]
Hardmon (1970, University of Waterloo) [22,23]
XRAY (1970, Applied Systems) [6,15]
A Counting Monitor (1970, University of Erlangen)
DYNAPROBE 7700, 7900 (1970, Comress) [13,18,24, 25,26,27]
TESDATA 1010, 1155 (1972, TESDATA) [28,29]

Minor modifications, usually the addition of a synchronizable clock and/or the ability to be controlled by a processor, could make the following capable of monitoring a network;

DYNAPROBE 8000 (1970, COMRESS)
Neurotron (1970, Stanford University) [2,30]
ADAM (1972, Hughes of Xerox) [3]
TESDATA 1185, 1200 (1972, TESDATA) [28,29]
Instrumentation for C.mmp (1972, Fuller of Carnegie-Mellon University) [32]

Relatively trivial changes, i.e., new software, the ability of control the monitor's activities remotely, and the ability to synchronize clocks between monitors, are necessary for these monitors:

SNUPER 2 (1967, Estrin) [5]
SLUR (1968, Murphy) [10]

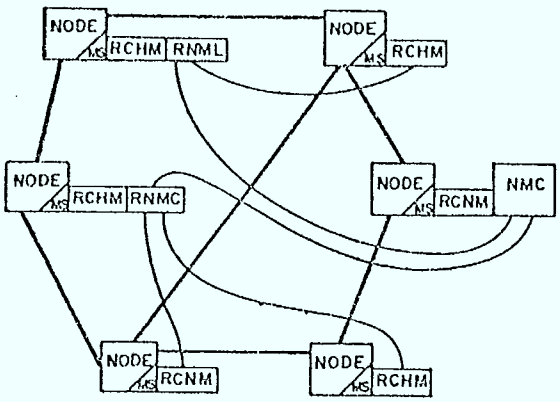
3. A COMPUTER NETWORK MONITORING SYSTEM

Section 1 lists several requirements that a designer of a network monitoring system should consider and attempt to satisfy. With these as goals, a network monitoring system has been designed and is being developed.

The monitoring system consists of:

(i) A set of computer-controlled hardware monitors, each of which is attached to a computer to be monitored;
(ii) Monitor control and data analysis software;
(iii) A network traffic generator;
(iv) Measurement software in each computer measured.

Figure 3.1 shows how the components of the system can be configured to monitor a network. Note that a telephone line, different from those of the network, may connect each monitor to the computer controlling it, or the monitor may be attached directly to the controlling computer. These telephone connections need not remain established throughout a monitoring session. Computer control is required only to set



MS - Measurement Software
RCHM - Remote Computer Controlled Hardware Monitor
RNMC - Regional Network Measurement Centre
NMC - Network Measurement Centre

Fig. 3.1 Application of Monitoring Hardware to a Network

up the experiments, to read accumulated data periodically during some experiments and to terminate the session. Each remotely-controlled monitor has a chip processor to handle real-time control details, and a mini-disk to hold the accumulated data.

A prototype system has been implemented, tested, and has been used to monitor a small(three node)network on the University of Waterloo campus. Plans are to measure a network consisting of two nodes separated by 400 miles in early 1975.

3.2 The RCHM

As Figure 3.2 illustrates, the Remote-Computer-controlled Hardware Monitor(RCHM) is composed of a number of specialized modules interconnected by a bus (the MONIBUS). The modules included in a monitor depend on the activities to be monitored. Each module is assigned a set of MONIBUS addresses which are used by the controlling computer to send control information and to receive monitored results. Thus far, the modules include:

(i) Event detectors
  a. Masked-word range comparators
  b. Character detectors (for bit-serial lines)
  c. Combinational logic unit
  d. Sequential logic units

(ii) Time measuring modules
  a. Time stamp units (to record time of occurence, identity, and other selected attributes of an event)
  b. Timer and event counters
  c. Network clock (synchronized with a standard source)
  d. Interval timer (for sampled measurements)

(iii) Data reducers and recorders
  a. Histogram generators
  b. Moment generator (yields the first four moments of a distribution)
  c. Timer and event counters
  d. Flip-flop bank
  e. Temporary memory

(iv) Communications and control equipment
  a. MONIBUS-to-communications-link interface and controller
  b. MONIBUS-to-UNIBUS (PDP-11) interface
  c. Interrupt generator
  d. Programmable switch matrices

A set of high impedance probes connect points of interest in the monitored computer to the monitor. The probes terminate on a patch panel containing signal conditioning circuitry.

The highly modular monitor architecture makes it quite easy to add new special-purpose data gathering or data reducing components as needed.

3.3 Software

The highly modular monitor hardware architecture and the desire to allow the system to evolve dictate a similar architecture for the software.

The current version of the monitoring system software has been designed and implemented to be the kernel of the eventual software. A rather detailed knowledge of the monitoring system is required to use this version. A high-level measurement language and a translator are being devised to make the system easier to use.

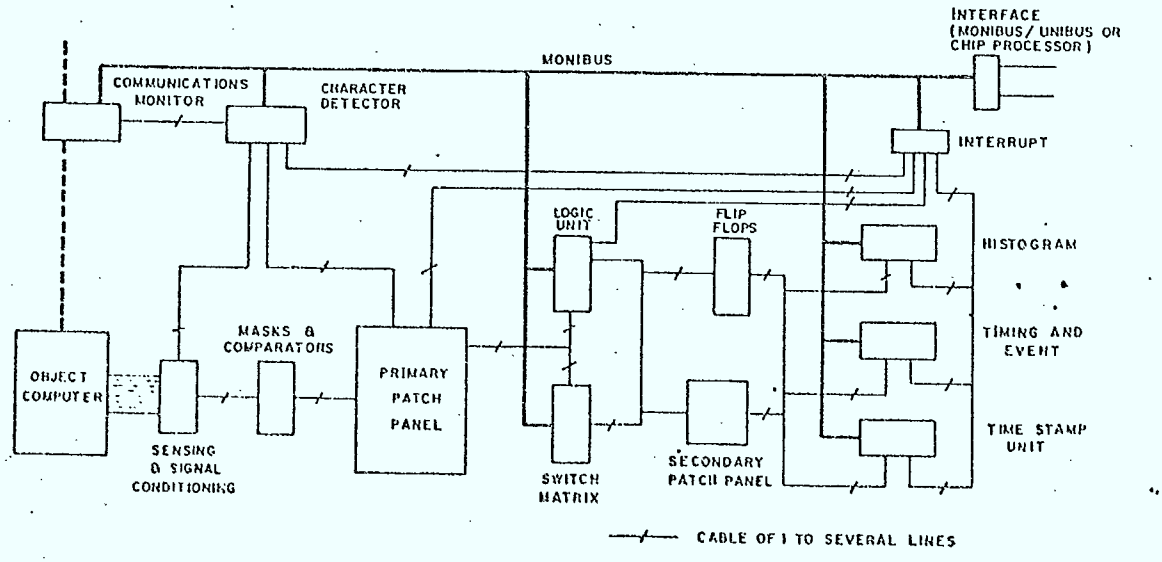


Fig. 3.2 Possible Interconnections of RCHM for each node

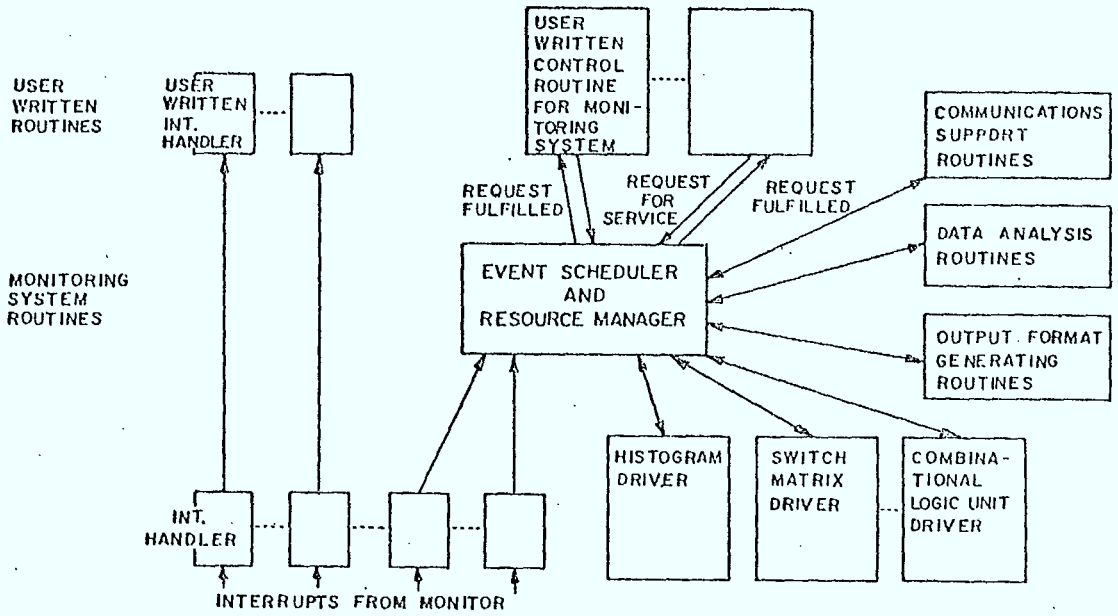


Fig.3.3 Monitor Control Software Structure

The heart of the software is a small real-time operating system. Depicted in Figure 3.3, it contains a set of RCHM module drivers, interrupt handling routines, primitives to aid authors of experiment control and data analysis routines in scheduling their routines, a small supervisor to allocate the resources (processor,memory,and communication), and support interaction with experiment control routines plus communications control routines. A limited set of standard routines for data analysis and output formatting, and an embryonic version of the measurement language translator complete the present version of the system software.

As experience is gained using the system, the measurement language is being defined. The measurement language is to be extensible so that the system will be easy to use and will allow the following scenario: First, a user instructs the system in how to measure, say, a certain type of message delay between selected nodes; subsequent users wanting this kind of delay will need to specify only its name and the nodes involved, so that the translator can generate the proper instructions for the system.

A load generator has been implemented to provide a user with the ability to specify what traffic should be in the network while monitoring, if known traffic is desired. The current version simulates the typing action of up to sixteen users at terminals with speeds up to 300 baud. The load generator transmits prepared scripts from disk to the appropriate line. It can simulate thinking and typing distributions. A load generator to produce higher speed traffic, simulating host traffic, will soon be available.

Measurement software in each node is obviously quite system independent. However, sets of standard measurement software primitives are being designed to measure parameters characteristic of many systems. We are striving to minimize the amount of such software required, as well as the amount of work required to write, install, and debug it. A standard means of communicating between this software and the RCHM is being designed.

4. APPLICATIONS

In order to monitor a network using the system, several steps must be taken:

(i) Determine where probes must be placed in each computer to perform the required monitoring activities;
(ii) Install the probes in the computers and set up the monitors;
(iii) Test the monitors and the probes;
(iv) Provide the load generator with the necessary scripts to drive the network (if desired);
(v) Write programs to control the network monitoring system as it monitors the network;
(vi) Debug these programs;
(vii) Initiate the experiment on the network monitoring system;
(viii) Analyze and interpret the results.

Today's computer architecture unfortunately demands that the first three steps be performed by a computer hardware expert. The fourth step for our system requires only the skills of one who knows how to use the network and the load generator's text editing package. Steps 5-8 require knowledge of use of our monitoring system and the characteristics of the monitored network.

Besides its intended use of monitoring a computer network, the system has been used to monitor the activities of a computer system. The monitor system, with minor software and hardware modifications, can be used to monitor a set of electronic switching offices for telephones. Furthermore, it appears that a slightly modified version of the monitor can be used as an important component of a computer network diagnostic system.

5. CONCLUSIONS

A few conclusions can be drawn from our experience thus far:
(i) The hardware monitor works and is not prohibitively expensive to build or use.
(ii) The modular components plus the bus architecture make it easy to add or subtract components as needed. Thus, the cost of a monitor depends on the complexity of the experiments to be performed.
(iii) Writing control programs for the monitor,even without the measurement language, is not difficult, because each component is addressed as a set of memory locations on the controlling PDP-11.
(iv) A system hardware expert would not be required to install the probes if computer manufacturers provided an accessible panel of probe points on their products. It appears that at least one manu-

facturer, Honeywell, has recognized this need, and plans to provide such a feature on their equipment.

ACKNOWLEDGEMENTS

The hardware skills of J. Runge and K. Jedermann, the software talents of F. Mellor, D. Goodspeed, R. Shen and J. Palframan have contributed immensely to the project.

REFERENCES

[1] C. T. Apple, The Program Monitor - A Device for Program Performance Measurement, Proc. of the ACM 20th National Conference, August 1965, pp. 66-75.

[2] R. A. Aschenbrenner, et al, Neurotron Monitor System, AFIPS, v. 39 (FJCC) 1971, pp.31-37.

[3] A. J. Bonner, Using System Monitor Output to Improve Performance, IBM Systems Journal, v.8 #4, 1969, pp.290-298.

[4] D. T. Bordsen, Univac 1108 Hardware Instrumentation System, ACM SIGOPS Workshop on System Performance Evaluation, Harvard U., April 1971 pp. 1-29

[5] G. Estrin, et al, SNUPER Computer, AFIPS, v.30 (SJCC) 1967, pp.645-656.

[6] L. E. Hardt, G. J. Kipovitch, Choosing a System Stethoscope, Computer Decisions, Nov. 1971, pp. 20-23.

[7] T. Y. Johnston, Hardware vs. Software Monitors Proc. of SHARE, XXXIV, v.1, March 1970, pp. 523-547.

[8] K. W. Kolence, Software Physics and Computer Performance Measurements, Proc. of the 1972 ACM Annual Conference, pp. 1024-1040.

[9] H. Lucas, Performance Evaluation and Monitoring, Computer Surveys, v.3, #3, Sept. 1971, pp. 79-91.

[10] R. W. Murphy, The System Logic and Usage Recorder, AFIPS, v.35 (FJCC) 1969, pp. 219-229.

[11] C. D. Warner, Hardware Techniques, ACM SIGCOSM Newsletter #5, Aug. 1970, pp. 5-11.

[12] G. D. Cole, Computer Network Measurements - Techniques and Experiments, UCLA, Oct. 1971, NTIS #AD-739-344.

[13] Comress: Dynaprobe 7900: System Specifications, Comress Report No. CR4-0031.

[14] J. M. Grochow, Real Time Graphic Display of Time-Sharing System Operating Characteristics, AFIPS, v.35 (FJCC) 1969, pp. 379-386.

[15] J. H. Salzer, J. W. Gintell, The Instrumentation of Multics; Second Symposium on Operating System Principles, Oct. 1969, pp. 167-174.

[16] K. W. Kolence, System Improvement by System Measurement, Data Base, Winter, 1969, pp. 6-11.

[17] E. F. Miller, An Experiment in Hardware Monitoring, General Research Corporation, RM-1517, July 1971.

[18] R. G. Canning, Savings from Performance Monitoring, EDP Analyzer, Sept. 1972.

[19] R. L. Patrick, Measuring Performance, Datamation, v.10, #7, July 1964, pp.24-27.

[20] P.D. Schulman, Hardware Measurement Device for IBM/360 Time Shared Evaluation, Proc. of the 22nd ACM National Conference, Aug. 1967, pp. 163-199.

[21] D. J. Roeck, W. C. Emerson, A Hardware Instrumentation Approach to Evaluation of Large Scale Systems, Proc. of the 24th ACM National Conference, 1969, pp. 351-367.

[22] L. Boyle, HARDMON: The University of Waterloo Hardware Monitor, University of Waterloo, Applied Analysis and Computer Science Department, Masters Essay, May 1973.

[23] C. E. Kühn, Organization for System Evaluation, INFOR v.9, #2, July 1971.

[24] Comress; Dynaprobe 7900: System Specifications Comress Report No. CR4-0032-1.

[25] Comress; Dynaprobe 7818: System Specifications Comress Report No. CR4-0036-1.

[26] Comress; Dynaprobe 7720: System Specifications Comress Report No. CR4-0023-2.

[27] Comress; Dynaprobe 7700: System Specifications Comress Report No. CR4-0022.

[28] System 1000-General Information 1100 Series, 1973, Tesdata Systems Corp., 7900 Westpark Drive, McLean, Virginia 22191.

[29] System 1000-Computer Performance Management System, 1972, Tesdata Systems Corp., 5539 Wisconsin Avenue, Chevy Chase, Maryland.

[30] L. Amiot, N. K. Natarajan, R. A. Aschenbrenner, Evaluation of a Remote Batch Processing System, Second Symposium on Operating System Principles Oct. 1969,pp. 24-29.

[31] J. Hughes, D. Cronshaw, On Using a Hardware Monitor as an Intelligent Peripheral, Xerox Corporation, Jan. 1973.

[32] S. H. Fuller, R. J. Swan, W. A. Wulf, The Instrumentation of C.mmp, A Multi-(Mini) Processor, IEEE Intl. Computer Society Conference, Feb. 1973, pp. 173-176.

[33] E. L. Burke, A Computer Architecture for System Performance Monitoring, 1st Annual SIGNE Symposium on Measurement and Evaluation, Feb. 1973, pp. 161-169.

APPENDIX B

A COMPUTER CONTROLLED HARDWARE MONITOR: HARDWARE ASPECTS

1. INTRODUCTION

The advent of complex computer technology has made it necessary to develop new flexible methods of evaluation. It is now required to evaluate systems because technology is making many hardware-software alternatives a reality.

Many organizations are connecting computers into complex networks. The rapid development of solid state technology has meant sudden changes in concepts of reality. A means of evaluating these new concepts was needed by the Computer Communication Networks Group at the University of Waterloo. A major effort has been made in the devolpment of a general purpose software-controlled hardware monitor as part of the creation of a hardware and software system to monitor computer networks.

The purposes and design objectives of such a monitor are well documented in other publications [1,2,3] and are outside the scope of this paper. In an earlier paper, the general aspects of our monitoring tools were discussed [4]. It is our intent to restrict discussion to the hardware design aspects of the monitor.

2. MONITOR SPECIFICATIONS

Electronic technology available when the project started in 1971 and measurement requirements suggested that it was reasonable to achieve a monitor resolution of 100 nanoseconds. This figure expresses the minium time between

events to ensure detection. It is also the degree to which we are able to identify the occurance of any single event.

Measurement technology as well as measurements were both goals. A hardware design that could be readily modifled without major re-design effort was needed. A bus structured hardware monitor was developed to facilitate changes.

In order to monitor a network successfully using hardware monitors, parts several physically separate locations. This implies two essential features: first, the monitor must be controlled remotely and second, many of the monitor functions should be performed under software control. Results have to be available under software control.

3. SYSTEM DESIGN

The evolution of the hardware monitor tools has resulted in the generalized block diagram shown in Fig. 1. Four essential types of devices have been developed in association with the hardware monitor.

Communication and control hardware was developed to provide both local and remote control. Initial control hardware consists of UNIBUS to MONIBUS interfacing. Reverse signalling is accomplished through an interrupt system. Control hardware used in sampling and timing applications also comes under this category.

Event detectors include simple signal con-

ditioners, character detectors on serial lines, combinational logic units and sequential logic units. These devices are used to enable the hardware monitor to have access to the computer system under study. It is the responsibility of devices in this class to provide the rest of the monitor hardware with signals in a standard format.

Actual measurements are performed in two modes, absolute and reduced. Absolute measurements are those which are recordings of raw data and do not represent reduced information. Examples are time measurements and event occurances. Reduced measurements are those in which the reported value represents some composite value. Examples of these include histograms and moments of distributions.

All devices in the monitor have been implemented using TTL logic. This was done because it was readily available and provided sufficient speed to meet our requirements.

4. MONIBUS IMPLEMENTATION

The monitor is structured around an asynchronous bus known as MONIBUS which has a simple efficient protocol. The bus contains all the address, data, control, and timing requirements to effect data transfers. The protocol allows for interrupt handling. The MONIBUS is used as a communication path between the controlling computer and a device on the MONIBUS. No provision is made for inter-device com-

munication on the MONIBUS. The signals used on the MONIBUS are shown in Table 1.

Certain assumptions are made in the implementation of the MONIBUS in our hardware monitor. All devices on the MONIBUS are assumed to be slave devices. This means that devices operate in a controlled mode under the control of the MONIBUS to UNIBUS interface. It means that the MONIBUS devices cannnot control any devices on the host machine. As a further simplification, no positive acknowledgement of the existence of a device is given. This reduces the time required to access devices on the MONIBUS, although this departs from practice commonly used on other bus-structured machines. The common problem associated with no positive acknowledgements is that of phantom devices, but this has not proved troublesome . The need for positive acknowledgements should be questioned as the penalties of longer access time and more complex hardware are the result. When attached to a PDP-11, the interface monitor makes devices appear in address space of the PDP-11.

5. MONITOR DEVICES

All devices are designed to function in conjunction with the protocol of the MONIBUS. As new monitor devices are found to be needed, they can be added with little disturbance to those already present.

In the same way, software has been developed to be modular, allowing it to be integrated into the system as hardware is added or removed.

6. COMBINATIONAL LOGIC UNIT

The Combinational Logic Unit is used to realize any function of eight Boolean variables. It is a programmable device consisting of sixteen words of scratch-pad memory and logic which can select each bit individually (Fig. 2). Four bits of the function select the appropriate word. The remaining four bits are used to select the individual bit of the word. Each bit represents the presence or absence of a prime implicant of the Boolean function.

7. TIME AND EVENT COUNTERS

The time and event counters provide information on two items which are of interest: the number of times a logical event occurs, and the total duration of the event. These are used to provide much of the general information on systems under study. As a basic module in the system, they have so far been used in nearly every measurement performed by the monitor. The maximum resolution of the counters is established as 100 nanoseconds.

Two thirty-two bit counters form the bases of the time and event counters (Fig. 3). The word length could be expanded readily if the need ever arose. The status register

determines whether each counter is used as a timer or as an event timer.

8. SWITCH MATRIX

The program-controlled switch matrix is used to facilitate rapid changes in experiments either by changing measurement types, or acting in response to variations as a result of measurements made in experiments in progress. New measurement technology will employ dynamic measurements as a method of reducing the amount of redundant data collected.

In our monitor two types of program-contolled switch matrices are employed. The first has sixteen inputs and four outputs. The second has eight inputs and eight outputs. In both cases an output can select any of the inputs (Fig. 4). The switch matrix can be changed at the instruction rate of the controlling machine.

9. INTERRUPT GENERATORS

In any program-controlled device there is often a need for intelligent hardware to raise alarm conditions. An interrupt system satisfies this need. This is the only means available on the MONIBUS to request reaction to events. A priority system is also included to handle simultaneous interrupts by several devices (Fig. 5).

Two lines on the MONIBUS are associated with the interrupts. The interrupt request line listens to all inter-

rupt cards. It is the responsibility of the MONIBUS to UNIBUS interface to acknowledge interrupts when other transfers are not in progress. This is achieved by asserting the Interrupt Acknowledge line. The interrupt card requesting service receives this acknowledgment and identifies itself on the data lines. At the same time the requesting interrupt card prevents any other interrupt card futher along the bus from receiving the interrupt acknowledge signal.

10. SERIAL LINE TAP

At the present time the majority of computers in networks communicate over bit-serial lines. In most cases the actual communication lines are readily available to the user of a hardware monitor, whereas the data in the interfaces is not usually accessable.

Two devices have been developed to process data from bit-serial lines. The first is a device which taps on a serial line and decodes the data. The second is an associative array which is used to identify particular characters.

The associative array can be loaded under program control. It identifies characters key to a particular measurement problem. An output is provided by the character detector which allows characters to be counted or particular sequences of characters to be recognized.

11. TIME STAMP

Periodic or predictable events can be measured with relative ease. A problem found by all who debug software or hardware is the apparent random failures. It was with this in mind that a device was developed to aid in monitoring rare and random events.

The Time-Stamp device consists of sixteen monitor lines which wait for logical transistion to activate them. When a line becomes active, its line number is recorded, along with the current real time and a snapshot of eight environment lines (Fig. 6). Thus one is able to obtain information about rare and random events without redundancy. The full potential of this device as a maintainance and debugging aid has not yet been realized.

12. HISTOGRAM

The simplest method of placing data information in perspective is through the use of a histogram. In the hardware monitor two types of histogram generators have been developed. The fast histogram generator has sixteen channels with 100 nanosecond resolution and 16 or 32 bit accuracy. The slow histogram generator has sixteen channels with five microsecond resolution and 16 bit accuracy.

The fast histogram generator is implemented with high speed counters which can be read under program control. Overflow detection is provided and can be fed to an inter-

rupt card.

The slow speed histogram is implemented using a sixteen word scatch-pad memory. It can be read under program control and is provided with overflow detection.

13. INTERVAL TIMER

It is common in many measurement systems to take periodic measurement samples. A program controlled interval timer was developed for this purpose (Fig. 8). It is a device consisting of two registers x and y. Its output line remains true for x time units followed by false for y time units. The period is x+y time units and the sample window is x time units. The value of each time unit is variable with an upper limit of 100 nanoseconds. X and y registers each have sixteen bit resolution.

14. QUAD COMPARATOR

In most computer systems much of the internal information appears on a parallel data bus. Two types of data buses are found in most computers containing information useful to hardware monitor measurements. The first is the address bus which contains information on the flow of the program as well as identifying the areas used to store data used in calculations. The second type of bus is the data bus used to move data within the computer. This bus demonstrates the effectiveness of peripherials as well as

providing a means of tracking key processing events.

We developed the quad comparator to observe bus structured data.(figure 7). The quad comparator has nine registers accessible from the MONIBUS. One register is used to mask out those bits which are not needed. The remaining eight registers are used to establish the range of the masked data. Because they are under program control, the data can be identified to any degree of accuracy. In the implementation our Quad Comparator can accept a sixteen bit bus. This bus width can be extended by stacking more than one Quad Comparator. For example in order to examine a thirty-two bit bus, two Quad Comparators are needed.

The Quad Comparator provides eleven outputs to give information about the range of values to be checked. The eight-program controlled registers give eight break-points. Seven of the outputs provide information indicating that the input value is in the range outlined by adjactent registers. The remaining four outputs deal with the two-end registers. Two of the outputs are used to indicate equality with the end values. The remaining two specify when the bus value is greater than one the end range of values or less than the other end value.

15. CONCLUSIONS

The bus structured nature together with the modular nature of the hardware and software components seem to have made our monitor easy to develop and use. These

features are the best safe-guard against obsolescence. New measurement tools, when they are developed, will be added with a minimum of effort.

To date, a single hardware monitor is operational under control of the first version of the Network Monitoring Centre software. These are physically co-located as the hardware and software have not been completed to permit control via telephone lines. The Network Monitoring Centre is controlled by a user language based on FORTRAN, which requires fairly detailed knowedge of the subject by the user. The hardware and software is under construction in order to permit remote operation of the hardware monitor, a second hardware monitor is under construction, and a second-generation software system for the Network Monitoring Centre is under development for allowing simpleuser interaction and automatic multitasking. The monitor has been used to measure single-CPU subjects, and simple two-node network. We plan to have monitored a three-node network by early 1975.

BIBLIOGRAPHY

[1] J. Hughes, D. Cranshaw, On Using a Hardware Monitor or an Intelligent Peripheral, Xerox Corporation Report, January 1973.

[2] C. D. Warner, "Hardware Techniques", ACM SIGCOSIM Newsletter #5, Aug. 1970, pp. 5-11.

[3] H. Lucas, "Performance Evaluation and Monitoring", Computer Surveys, v. 3, #3, September 1971, pp. 79-91.

[4] D. E. Morgan, W. Banks, W. Colvin, D. Sutton, "A Performance Measurement System for Computer Networks", Proceeding of 1974 IFIP Congress, pp.29-33.

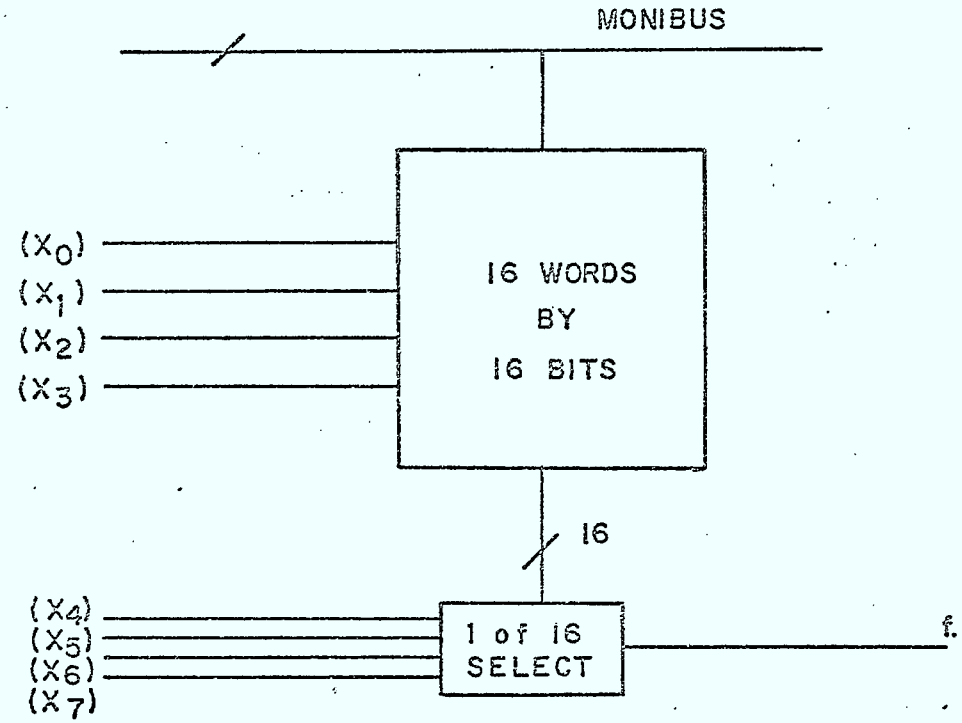


Fig.2 COMBINATIONAL LOGIC UNIT

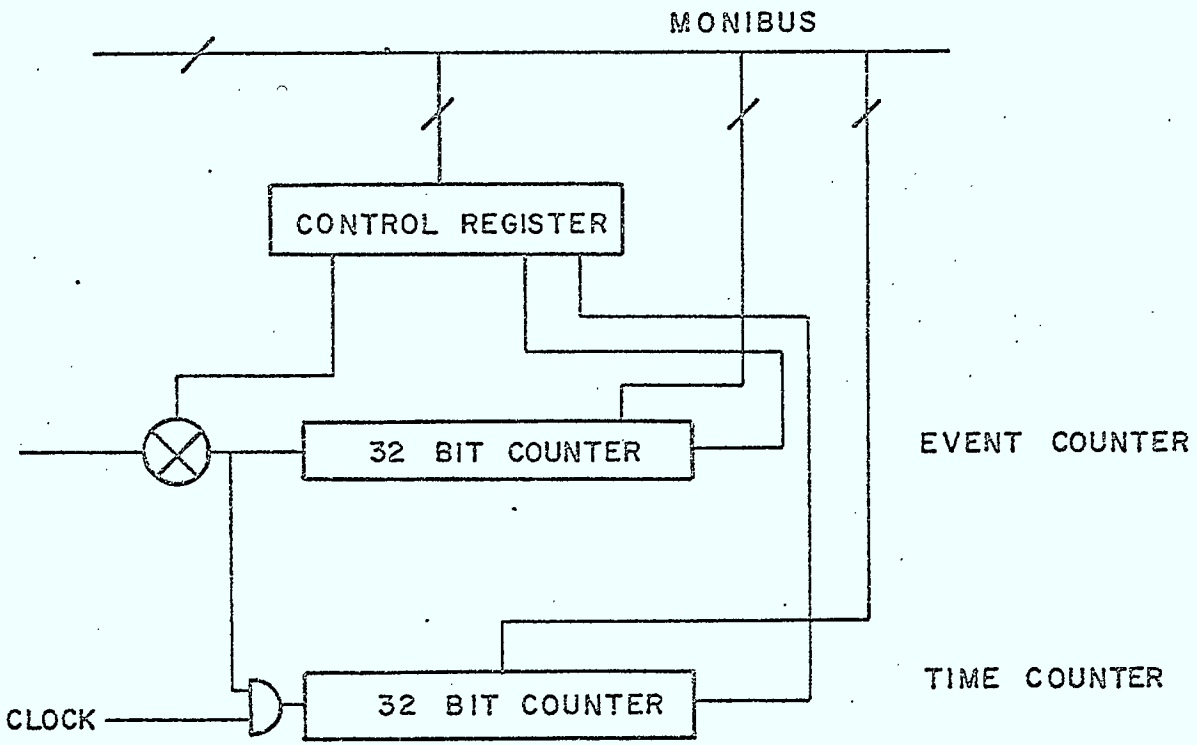


Fig.3 TIME AND EVENT COUNTERS

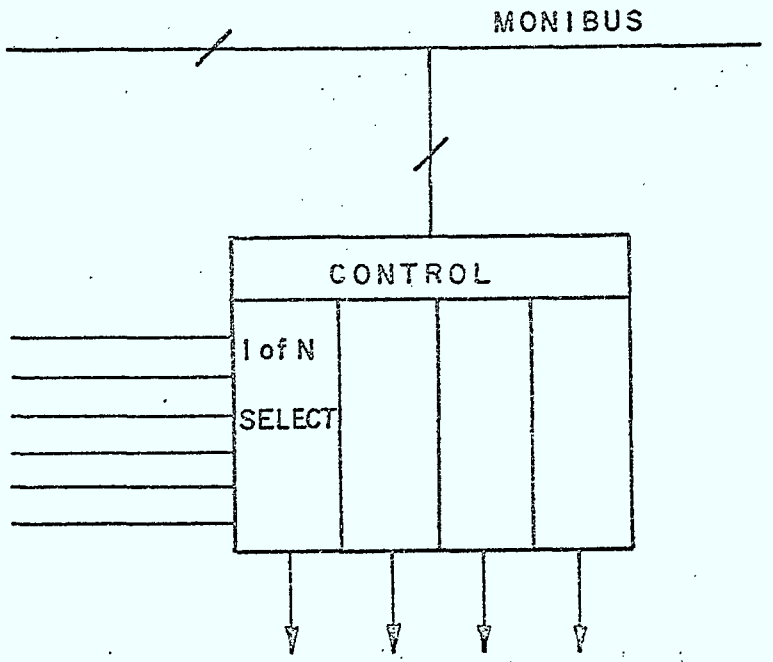


Fig.4 SWITCH MATRIX

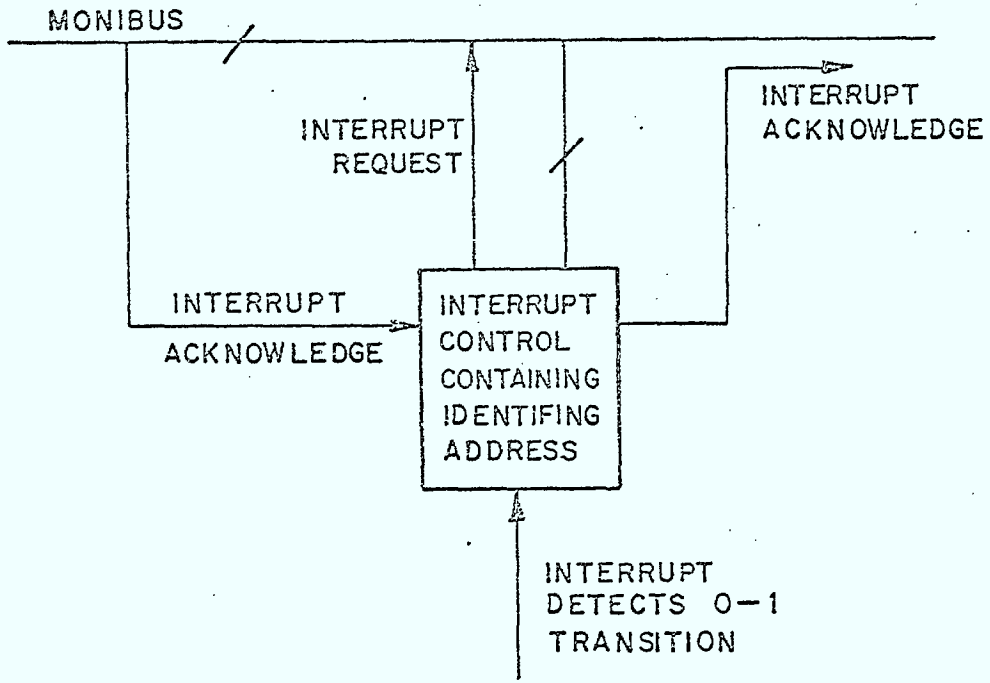


Fig.5 INTERRUPT GENERATOR

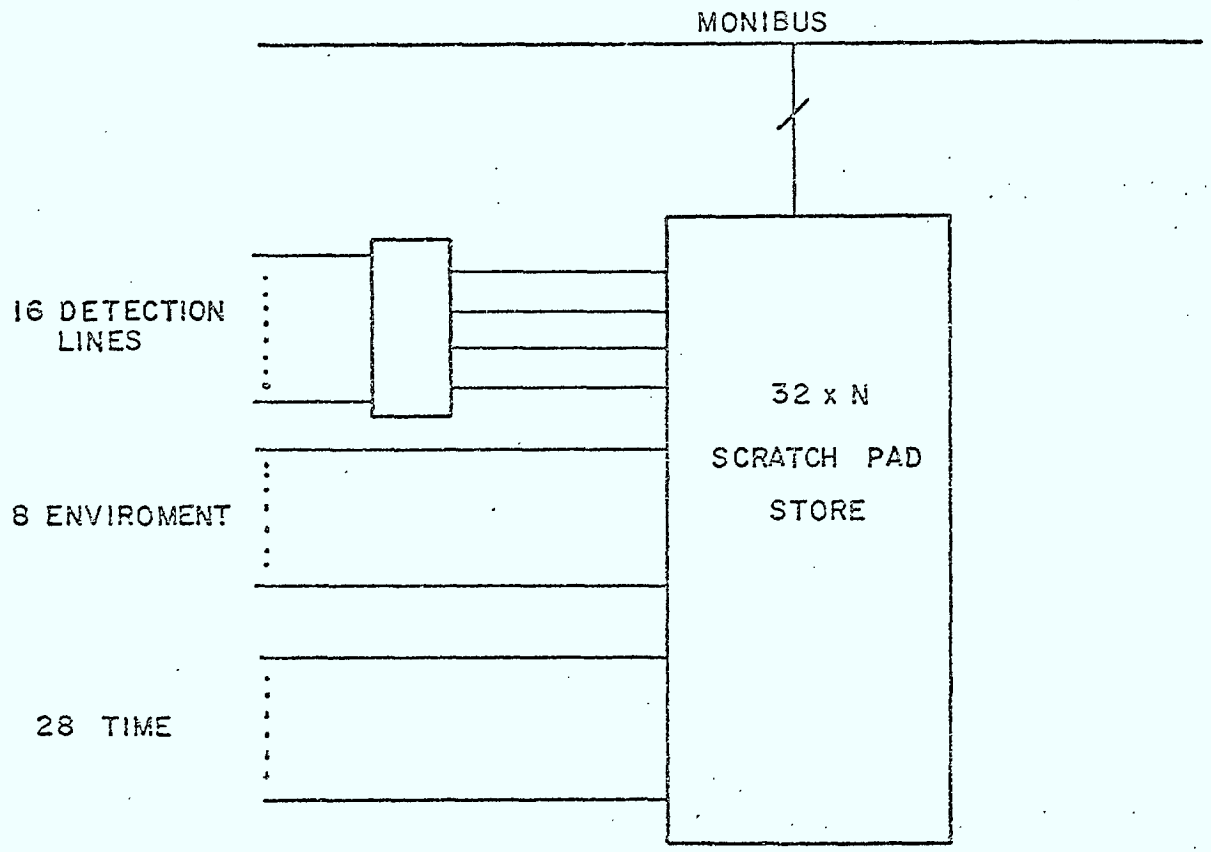


Fig. 6 TIME STAMP

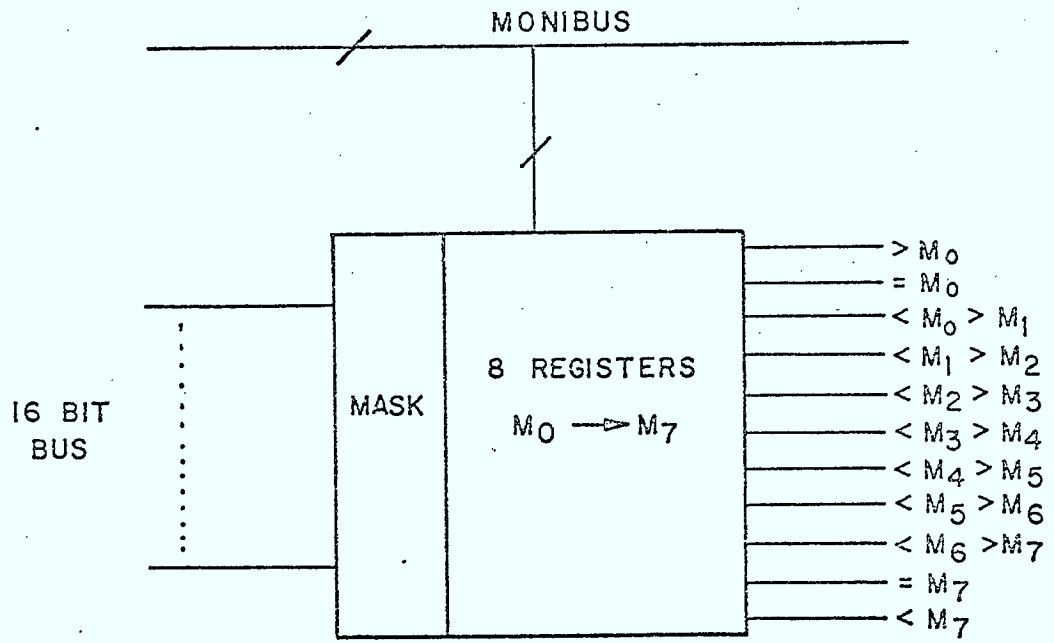


Fig. 7 QUAD COMPARATOR

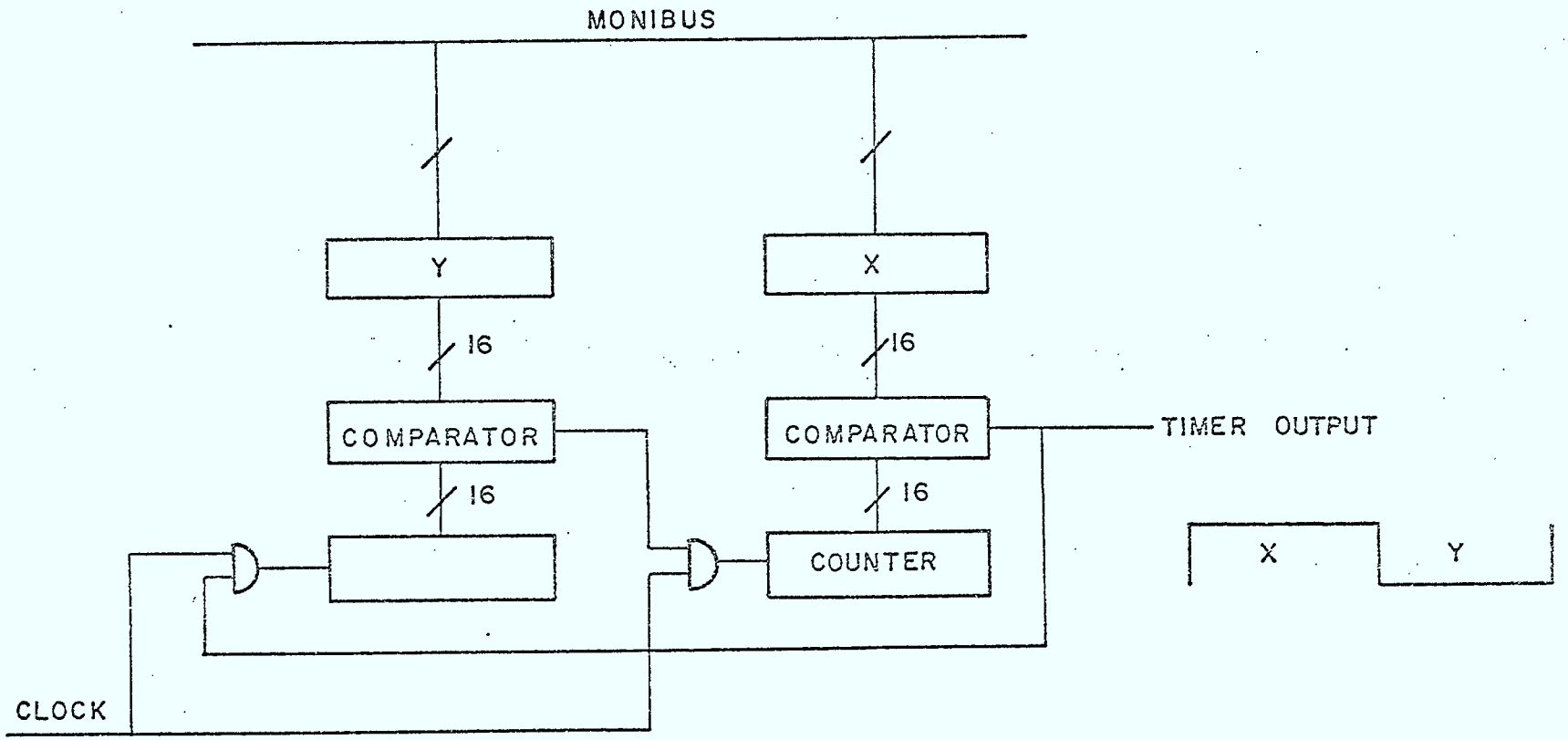


Fig. 8 INTERVAL TIMER

SUMMARY:

The remote computer-controlled hardware monitor or RCHM is a major component of the Computer Network Monitoring System which has been developed at the University of Waterloo during the last three years. This paper describes the hardware concepts and the design of the RCHM.

The Computer Communications Networks Group at the University of Waterloo has recognized the need to monitor networks as a means of evaluating their performance. This knowledge will help to validate analytical models, help to optimize and debug software, and lead to the development of a flexible means by which charging algorithms and diagnostics can be implemented.

In a typical computer network there are several computer sites linked by data lines of varying capacities. The sites may be separated by only a few feet or by several hundred miles. It is the task of the monitoring system to observe and correlate activity at several locations simultaneously. It is important to minimize interference to the system being monitored. The RCHM hardware is centred around a bus-structured wired-logic processor. The bus structure allows variations to be introduced readily into an individual monitor. It also allows new monitoring devices to be introduced at a later date without significantly disturbing those already present.

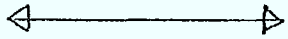
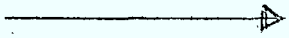
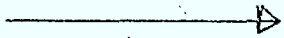
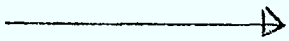
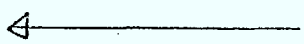
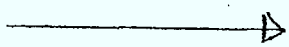
SIGNAL NAME	SIGNAL DIRECTION
ADDRESS 0:7	
DATA 0:15	
READ/WRITE	
GO	
INITIALIZE	
INTERRUPT REQUEST	
INTERRUPT ACKNOWLEDGE	

TABLE 1 MONIBUS SIGNALS

APPENDIX C

A COMPUTER NETWORK MONITORING SYSTEM

A COMPUTER NETWORK MONITORING SYSTEM

by

D. E. MORGAN*, W. BANKS*,

D. GOODSPEED*, R. KOLANKO*

* Computer Communications Networks Group
and Department of Applied Analysis and Computer Science
University of Waterloo, Waterloo, Ontario, Canada

Research supported by Department of Communications of Canada, research contract no. SP2-36100-3-0406; Defence Research Board of Canada, grant no. 9931-37; National Research Council of Canada, grant no. A8116 and by Mordata Ltd. The research was performed in the Computer Communications Networks Group's laboratory at the University of Waterloo

ABSTRACT

In order to help satisfy an apparent need for tools for monitoring the activities of a computer network (See Mamrak <1>), a system of special hardware and software, called a Computer Network Monitoring System (CNMS), is being implemented in the University of Waterloo Computer Communications Networks Group (CCNG). The paper discusses the motivation and derivation of the CNMS, then provides functional descriptions of most of the major hardware and software components, illustrates use of the CNMS, and lists experiments and applications. In a previous paper <2>, the motivation and architecture of the system were sketched.

The CNMS consists of: (1.) A set of hybrid monitors, each of which is controlled by a locally or remotely located computer; (2.) Monitor control and data analysis software; (3.) A Network Traffic Generator; (4.) Measurement software in each computer monitored. Each computer to be monitored is attached to a monitor. Telephone lines, possibly different from those of the network, connect the monitors to the controlling computer.

1. INTRODUCTION

A computer system, consisting of hardware and the software to control it, is often so complex that it is difficult to understand what is being done, how efficiently it is being done, and what problems exist. Moreover, computer systems are often connected to other computers as well as terminals to form even more complex computer networks.

Cole and others have defined a network of computers to consist of two or more computers linked together, while a computer network has been defined to be either a network of computers or a set of terminals connected to one or more computers<3>. Most networks of computers consist primarily of nodes, hosts, transmission links, and terminals. A node (in this context) usually refers to a computer used primarily to switch data. A computer whose primary role is not switching data in the network to which it is attached, is called a host. In some networks, a sharp distinction is made between nodes and hosts, while in others no distinction exists. Terminals are devices which serve as the interface between man and the computer. The transmission links, of course, join this collection of hardware together to form a network.

Determination of what a computer system or network is doing is essential to effective management of it. This involves monitoring (observing) its behaviour as it executes a set of programs and responds to its environment. The

behaviour of a system or network acting on a set of programs and data is characterized by the sequence of values of certain parameters of the system and by the sequence of events that occur as the system executes. These are manifestations of the sequence of states traversed by the system as programs of the workload are executed.

Although a variety of hardware and software monitoring tools and techniques have been developed to aid in observing the behaviour of computer systems (See, for example, <4-14>), little attention has been paid to developing tools for monitoring computer networks. Kleinrock and Cole <3> have successfully used elegant software techniques to monitor the performance of the ARPA network. Abrams et al at the National Bureau of Standards of the U.S.A., have developed tools for observing data flowing along a communications line between a computer and a terminal<15>. Fuller and others are instrumenting the C.mmp multi- mini processor system.

The purpose of this paper is to discuss the motivation, architecture, components, and use of a system of hardware and software designed to monitor the behaviour of a computer network or system. Morgan, Banks and others have previously sketched the purpose and architecture of the Computer Network Monitoring System(CNMS) <2,16>. This CNMS is being created in the Computer Communications Networks Group at the University of Waterloo.

The paper is organised in six sections. Section 2 motivates the need for a monitoring system rather than a set of unrelated, uncoordinated software and hardware tools. By considering the problems involved in monitoring a computer network, the section motivates characteristics and major components of an ideal computer network monitoring system. Section 3 presents the architecture and describes the components of the CNMS being created at Waterloo. Use of this CNMS is explained and illustrated in Section 4. Section 5 lists some experiments being performed using the CNMS, and mentions some potential applications of the system. Section 6 summarises this research, evaluates the CNMS in terms of nine characteristics presented in Section 2, presents our conclusions so far, and outlines some future work.

2. NETWORK MONITORING SYSTEM MOTIVATION

There are four fundamental reasons for monitoring a computer system or network:

A. To observe its performance, thereby determining whether work is flowing satisfactorily through it;

B. To detect malfunctions;

C. To diagnose the causes of any problems observed;

D. To measure the resources used so that appropriate charges can be made.

Usually the people who wish to monitor the activities of a computer system are neither hardware, software, nor statistics experts. Rather, they are either managers responsible for the system, software maintenance people who lack detailed knowledge of hardware, hardware maintenance personnel who lack detailed knowledge of software, or researchers (students or professionals) seeking statistics for their work. It is indeed rare that the person seeking information is a computer software, hardware, and statistics expert all in one. Thus, system monitoring tools should be easy to learn to use as well as easy to use, and detailed knowledge of hardware, software, and/or statistics should not be necessary to observe the system and get useful information about it. Furthermore, the tools and techniques should be incorporated in a single monitoring system to avoid having to learn to use several different tools and

techniques.

Computer networks are often distributed geographically, so monitoring the behaviour of a computer network involves distributing the monitoring activities across the network. In order to correlate observations from scattered sites, these monitoring activities must be centrally controlled and coordinated, and the results must be centrally analyzed. Thus, monitoring a computer network involves communications as well as monitoring. Network monitoring tools and techniques must be designed with this in mind.

Although software has been used with some success to monitor the performance of computer systems and networks, experience indicates that monitoring software without hardware often perturbs the system or network unsatisfactorily. Hardware monitors without software aid are too inflexible for most network applications. Certain parameters, events and their attributes can best be determined by software, while others can best be determined by hardware. Thus, some combination of hardware and software monitoring tools appears better than either hardware or software tools alone. Moreover, if a computer network is to be monitored, either the computers in the network could include special hardware to aid them in self-monitoring while producing minimal interference with the network's functions, or a set of software-controlled hardware monitors (often called 'hybrid monitors'), one attached to each computer to be monitored,

could be employed to complement necessary monitoring software in each computer. Each of these software-controlled monitors should be capable of having its activities controlled and its monitored results sent through telecommunications links. Control of these monitors and analysis of the data could be performed by a computer system at a Network Monitoring Centre (NMC), such as the ARPA Network's NMC, which is used to coordinate software measurement activities at each IMP and to collect and analyse the data.

Because transmission of large volumes of data is expensive and usually not necessary, a network monitoring system needs facilities for reducing data before transmission to the NMC. Cole <3> showed that good approximations to many kinds of distribution functions could be obtained from log histograms. Much of the measurement data obtained by the ARPA network measurement software is transmitted in the form of log histograms to reduce the amount of data transmitted. Extending Cole's reasoning, only the first four moments of a distribution are required to model its behaviour for most purposes. Rather than produce these moments at the NMC from data transmitted from remotely-located monitors, equipment,hardware could be provided in each remotely-located monitor to produce these moments, then transmit only the moments to the NMC, thereby reducing the data that must be transmitted. The main disadvantage is the cost of such data

reduction equipment.

The activities and parameters of a system or network can be monitored continuously, periodically on a sampling basis, or only when events of interest occur. An event usually consists of a logical and/or sequential combination of other events. Fundamentally, an event is the occurrence of a specific pattern or sequence of patterns in particular portion(s) of the system, network, environment, or workload. Tools for monitoring events should include facilities:

A. To detect specified event(s);

B. To register the (real or relative) time of occurrence of the event;

C. To time the duration of the event (i.e., the set of states comprising the event, or bounded by two particular events) and/or its consequences;

D. To obtain and record selected attributes of the system, workload, and/or environment when the event occurred;

e. To count the number of occurrences of the event;

F. To initiate some action as a consequence of the event, e.g., diagnosing the cause of a problem defined by the occurrence of the event, checking for any damage, or initiating repair and/or recovery activities.

A well-known problem in physics and astronomy is to

determine the order in which nearly simultaneous events occur in widely separated systems. The same problem occurs when monitoring a network of computers. One way to minimise such problems is to synchronise all the clocks as accurately as possible with a single, very accurate, reliable source of time-of-day readings such as that provided by the National Bureau of Standards of the U.S., or the National Research Council of Canada.

In order to determine the effects that changes have on the performance of the network, some way of controlling the workload applied to the network is desirable. Thus, a monitoring system would be more useful if it included facilities to apply loads with specified characteristics to the object system or network.

From the above discussion, it follows that an ideal CNMS should include the hardware and/or software tools necessary:

A. To observe, measure, record, and evaluate the behaviour of each of the components of a computer network (including its workload and environment), such tools being:

1. A set of computer monitoring systems, each having hardware and/or software capable of detecting particular events, measuring their attributes, recording and reducing the data, and transmitting the data for analysis elsewhere;

2. Terminal and/or telecommunications link monitors, each having hardware and/or software to observe activities and information flow.

B. To define, control, and coordinate monitoring activities throughout the network.

C. To provide a single, accurate, reliable source of time (e.g., time of day readings and precise intervals) for the entire CNMS.

D. To provide network traffic with known characteristics, should the CNMS user need this capability for monitoring purposes.

To the best of our knowledge, no such computer network monitoring system has yet been developed. However, a number of hardware monitors and software monitors, plus a few hybrid monitors (e.g., <17,18,19>) have been developed for computer systems, and software techniques and tools have been used by Kleinrock, Cole, and others to monitor computer networks <3>.

Experience in monitoring computer systems, and study of pertinent literature indicate that an ideal computer network monitoring system (CNMS) should possess the following characteristics:

A. Be easy to use, yet flexible and expandable;

B. Be as system independent as practical;

C. Be dependable and easily diagnosed;

D. Allow gathering of measurement data at a

distance from the monitor control and analysis functions, with minimal human intervention required;

E. Span the network;

F. Interfere minimally with the performance and integrity of the measured systems;

G. Interfere minimally with computer-computer and terminal-computer communications;

H. Have no ill effects on the security or integrity of any of the systems;

I. Offer a choice of resolution, so that the unit of measure fits what is measured;

J. Be low in cost while not compromising the other goals.

It is, of course, impossible to achieve the ideal CNMS. Nevertheless, this paper describes an attempt to produce a system that hopefully will accomplish many of these goals to a reasonable degree. In Section 6, the CNMS described in Section 3 is evaluated in terms of these characteristics of an ideal CNMS.

3. DESCRIPTION OF A COMPUTER NETWORK MONITORING SYSTEM

3.1 SYSTEM ARCHITECTURE

Using the characteristics listed in Section 2 as goals, a CNMS has been designed and is being developed. A prototype system has been implemented and is being tested by using it to monitor two types of mini-computers and two small laboratory networks. This CNMS consists of the following major components, which correspond with the list of tools needed that is given in Section 2:

A. A set of hybrid monitors (called Remote Controlled Hybrid Monitors, and abbreviated RCHM), each being controlled by a computer which can be miles away (i.e., at the Network Monitoring Centre), and containing components to enable it to monitor a computer system and a set of communications links to terminals or other computers;

B. Software to define, control, and coordinate the activities of a set of hardware monitors, and to obtain and analyse data from them;

C. Monitoring software in each observed computer, and tools to enable the activities of the monitoring software to be controlled and coordinated from the Network Monitoring Centre;

D. Facilities in each hardware monitor to gain access to a single standard clock for time-of-day readings as well as precise intervals;

E. A network traffic (load) generator capable of simulating the activities of several users (human or non-human) interacting with the network.

Figure 3.1 shows how the components of this CNMS can be configured to monitor a network. Note that a telephone line, which may be physically or logically separate from the data links of the network, can connect each monitor to the computer controlling it, or the monitor can be attached directly to the controlling computer. These telephone connections need not remain established throughout a monitoring session. Computer control is required only to set up the experiments, to read accumulated data periodically during some experiments, and to terminate the monitoring session. Each remotely-controlled monitor has a micro-processor to handle real-time details, and a mini-disk to hold accumulated data for the NMC.

If the NMC cannot control all the RCHMs in the network, Regional Network Monitoring Centres (RNMCs) are introduced to form the hierarchy illustrated in Figure 3.2. Note that RCHMs can be controlled indirectly (i.e., via telecommunications links) or directly by either a RNMC or a NMC.

As Figure 3.3 illustrates, the RCHM is composed of a number of specialized modules interconnected by a bus, called the MONIBUS. The modules included in a monitor depend on the activities to be monitored. Each module is assigned a set of MONIBUS addresses which are used by the

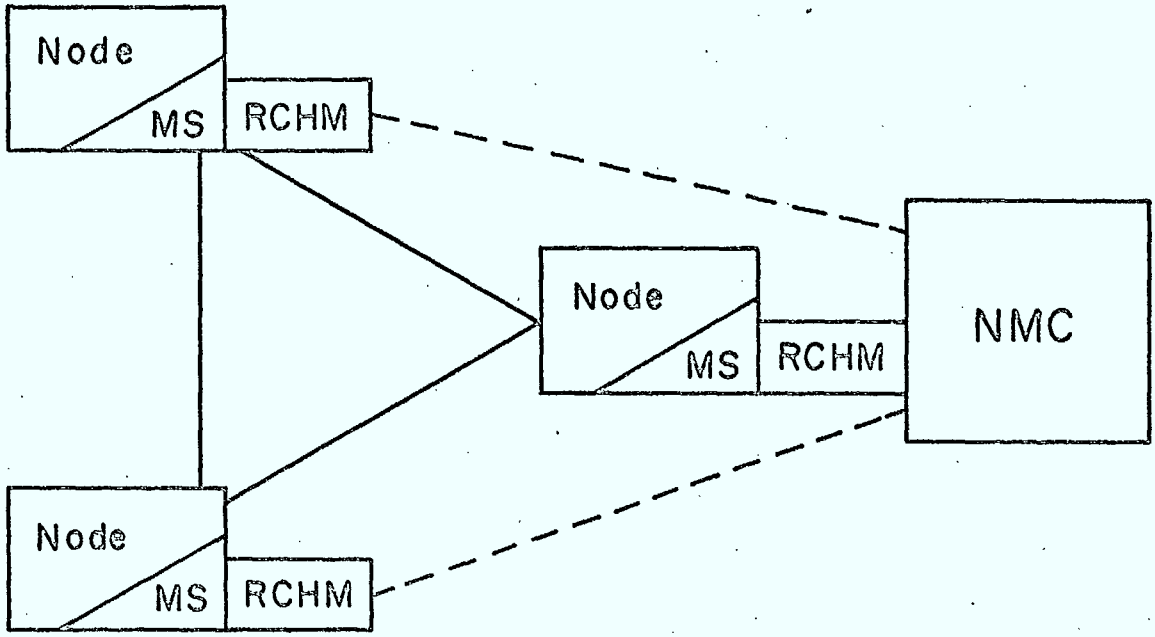


Fig. 3.1

MS - Measurement Software

RCHM - Remote Computer Controlled Hardware Monitor

RNMC - Regional Network Measurement Centre

NMC - Network Measurement Centre

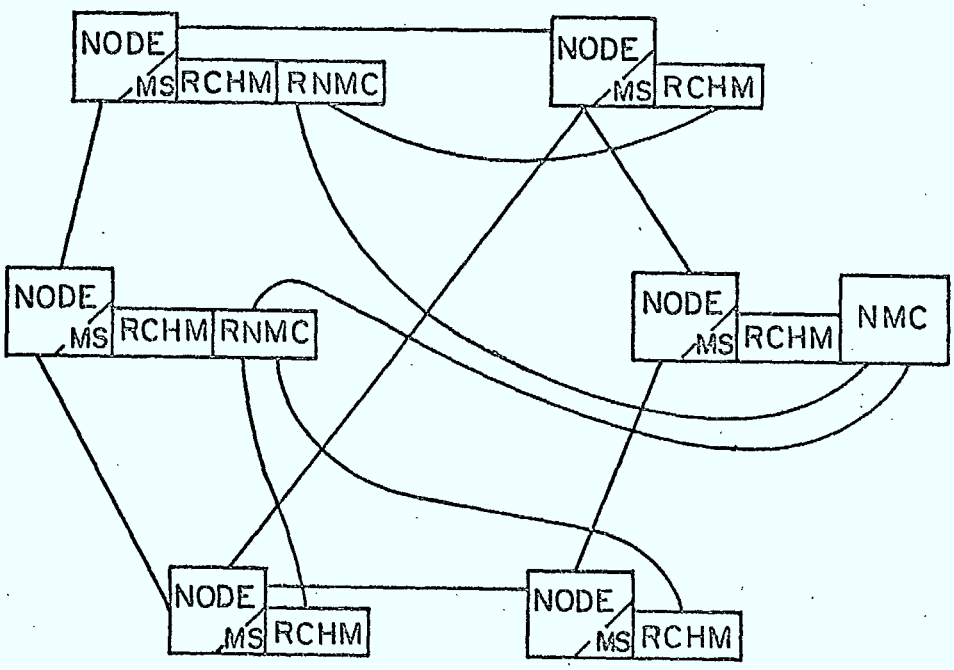

NODE
MS
RCHM
RNMC
NODE
MS
RCHM
NODE
MS
RCHM
RNMC
NODE
MS
RCHM
NMC
NODE
MS
RCHM
NODE
MS
RCHM

Figure 3.2

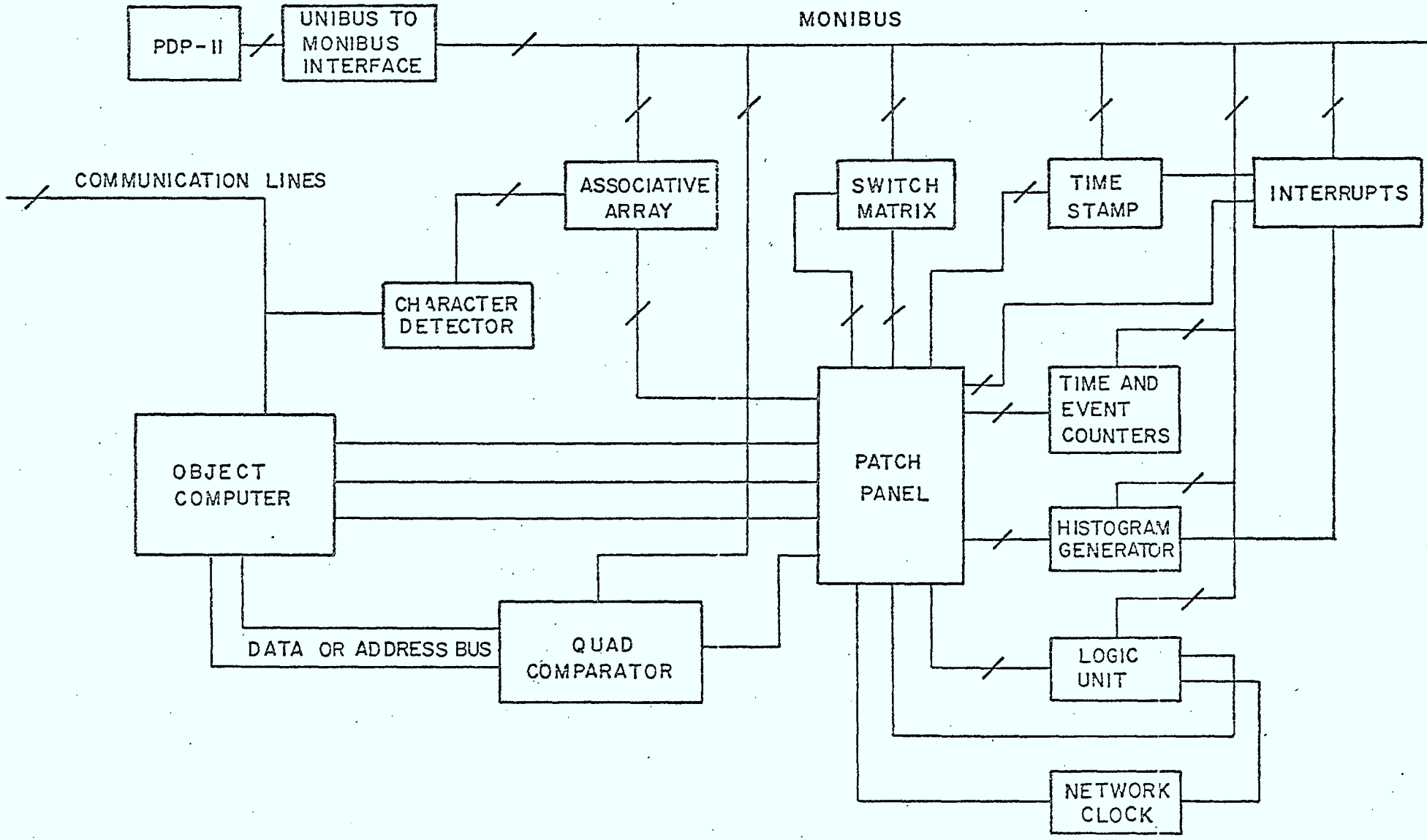

PDP-11
UNIBUS TO MONIBUS INTERFACE
MONIBUS
COMMUNICATION LINES
ASSOCIATIVE ARRAY
SWITCH MATRIX
TIME STAMP
INTERRUPTS
CHARACTER DETECTOR
TIME AND EVENT COUNTERS
OBJECT COMPUTER
PATCH PANEL
HISTOGRAM GENERATOR
DATA OR ADDRESS BUS
QUAD COMPARATOR
LOGIC UNIT
NETWORK CLOCK

Figure 3.3

controlling computer to send control information and to receive monitored results. The modules included so far are listed here and described in Section 3.2:

A. Event detectors

1. Masked-word range comparators, used to detect an event defined in terms of data or address ranges;

2. Combinational logic units, used to detect an event defined in terms of Boolean functions of other events;

3. Sequential logic units, used to detect an event defined as a sequence of other events;

4. Character detectors for bit-serial lines.

B. Time measuring modules

1. Time stamp units, used to record time of occurrence, identity, and other selected attributes of an event;

2. Event timers;

3. Interval timers, for sampled measurements;

4. Network clock, which is synchronized with a standard reference clock.

C. Counters, data reducers, and data recorders

1. Histogram generators;

2. Moment generator, used to yield the first four moments of a given distribution;

3. Event counters;

4. Flip-flop banks;

5. Content-addressable memory (CAM);

6. Random-access memory (RAM).

D. Communications and control equipment

1. Programmable switch matrices

2. Interrupt generators;

3. RCHM controller and communications link interface;

4. MONIBUS-to-UNIBUS (PDP-11) interface.

E. Signal conditioning circuitry and patch panels.

A set of high impedance probes connects points of interest in the object computer to the monitor. The probes terminate on a patch panel containing signal conditioning circuitry.

The highly modular monitor architecture makes it quite easy to add new special-purpose data gathering or data reducing components as needed. This modular hardware architecture and the desire to allow the monitoring system to evolve dictate a similar architecture for the software. Figures 3.4 and 3.5 depict the architecture of the software at this stage of its evolution.

The heart of this software is a small, real-time, message-switched operating system, containing a set of RCHM module drivers, interrupt handling routines, primitives to aid authors of experiment control and data analysis routines in writing and scheduling execution of their routines, a

small supervisor to allocate resources (processor, memory, and communications) plus the communications control routines. A limited set of standard routines for data analysis and output formatting, an embryonic version of a translator for a monitoring language, and software to allow the user to interact with the monitoring system, complete the present version of the system software. These software components are described in Section 3.2.

As experience is gained using the system, a monitoring language is being defined. The current version of the language is an extension of Fortran; however, the possibility of basing the language on a BCPL-like language (e.g., B <20>) is being actively pursued.

A load generator has been implemented to provide a user with the ability to specify what traffic should be in the network while monitoring, should known traffic be desired <21>. The current version simulates the typing action of up to sixteen users at terminals with speeds up to 300 baud. The load generator transmits prepared scripts from disk to the appropriate line(s), and can simulate thinking and typing time distributions. A version of the load generator to produce higher speed traffic, simulating host activities, is being created. Eventually, the load generators will also be controllable from the NMC using the monitoring language.

Monitoring software in each node (or host) is obviously quite system dependent; however, sets of standard

monitoring software primitives are being designed to observe parameters characteristic of many systems. We are striving to minimize the amount of such software required, as well as the amount of work required to write, install, and debug it. Standard means of communicating between this software and the RCHM are being designed.

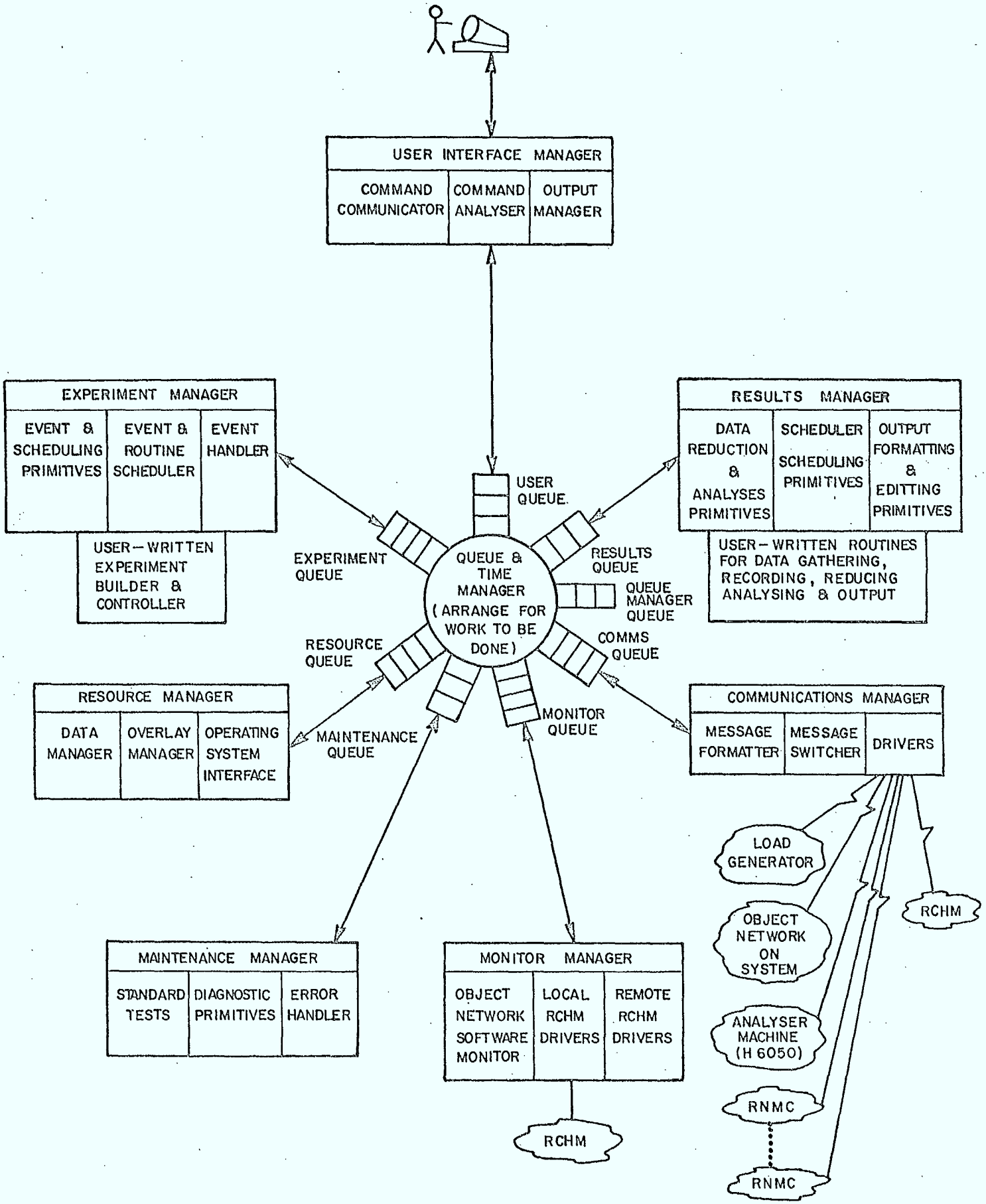

USER INTERFACE MANAGER
COMMAND COMMUNICATOR
COMMAND ANALYSER
OUTPUT MANAGER
EXPERIMENT MANAGER
EVENT & SCHEDULING PRIMITIVES
EVENT & ROUTINE SCHEDULER
EVENT HANDLER
USER-WRITTEN EXPERIMENT BUILDER & CONTROLLER
RESULTS MANAGER
DATA REDUCTION & ANALYSES PRIMITIVES
SCHEDULER SCHEDULING PRIMITIVES
OUTPUT FORMATTING & EDITTING PRIMITIVES
USER-WRITTEN ROUTINES FOR DATA GATHERING, RECORDING, REDUCING ANALYSING & OUTPUT
USER QUEUE
EXPERIMENT QUEUE
RESULTS QUEUE
QUEUE MANAGER QUEUE
QUEUE & TIME MANAGER (ARRANGE FOR WORK TO BE DONE)
RESOURCE QUEUE
COMMS QUEUE
MONITOR QUEUE
MAINTENANCE QUEUE
RESOURCE MANAGER
DATA MANAGER
OVERLAY MANAGER
OPERATING SYSTEM INTERFACE
COMMUNICATIONS MANAGER
MESSAGE FORMATTER
MESSAGE SWITCHER
DRIVERS
LOAD GENERATOR
OBJECT NETWORK ON SYSTEM
ANALYSER MACHINE (H 6050)
RNMC
RNMC
RCHM
MAINTENANCE MANAGER
STANDARD TESTS
DIAGNOSTIC PRIMITIVES
ERROR HANDLER
MONITOR MANAGER
OBJECT NETWORK SOFTWARE MONITOR
LOCAL RCHM DRIVERS
REMOTE RCHM DRIVERS
RCHM

Figure 3.4

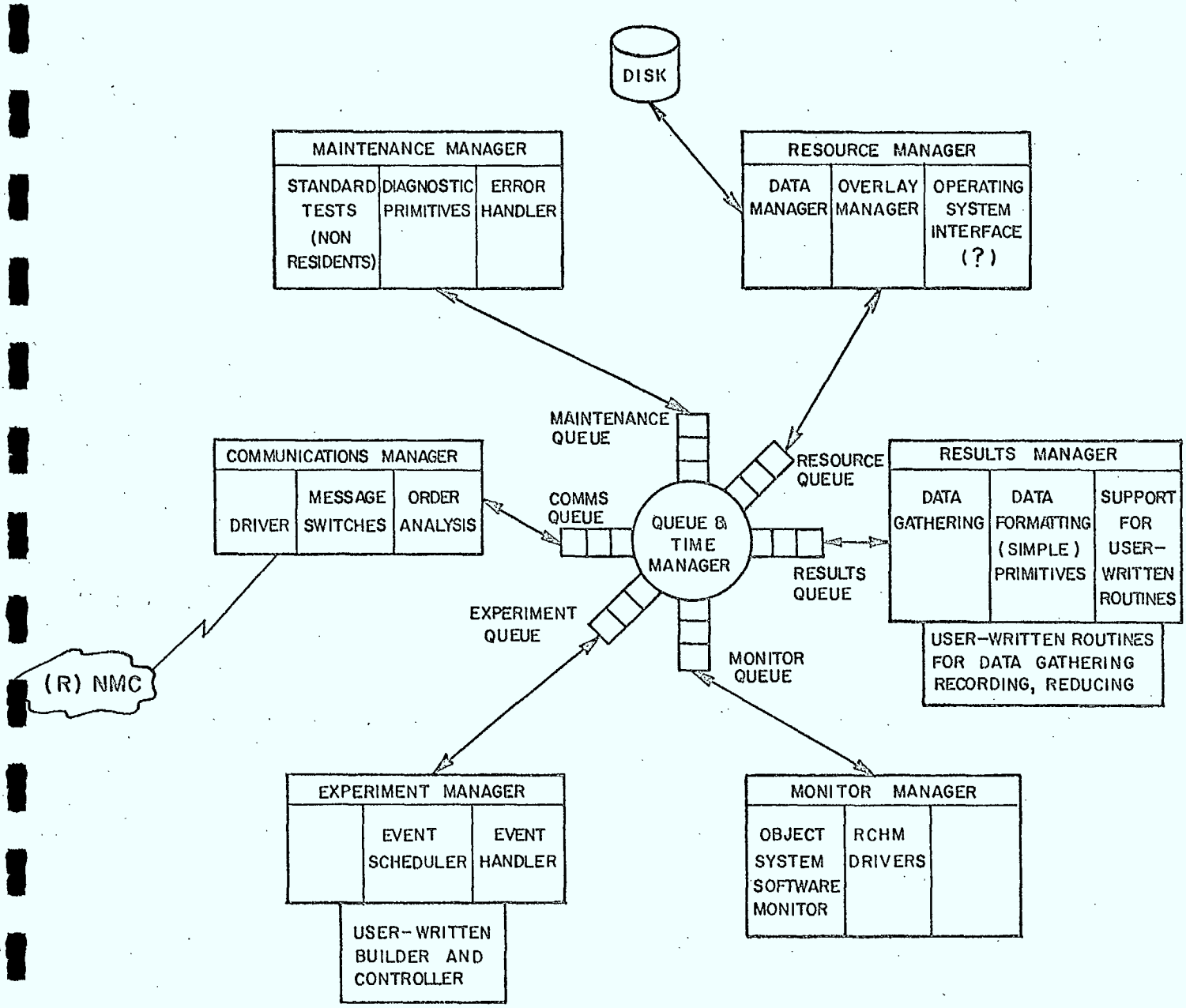

RCHM CONTROLLER
DISK
MAINTENANCE MANAGER
STANDARD TESTS (NON RESIDENTS)
DIAGNOSTIC PRIMITIVES
ERROR HANDLER
RESOURCE MANAGER
DATA MANAGER
OVERLAY MANAGER
OPERATING SYSTEM INTERFACE (?)
MAINTENANCE QUEUE
RESOURCE QUEUE
COMMUNICATIONS MANAGER
DRIVER
MESSAGE SWITCHES
ORDER ANALYSIS
COMMS QUEUE
QUEUE & TIME MANAGER
RESULTS QUEUE
RESULTS MANAGER
DATA GATHERING
DATA FORMATTING (SIMPLE) PRIMITIVES
SUPPORT FOR USER-WRITTEN ROUTINES
USER-WRITTEN ROUTINES FOR DATA GATHERING RECORDING, REDUCING
EXPERIMENT QUEUE
MONITOR QUEUE
(R) NMC
EXPERIMENT MANAGER
EVENT SCHEDULER
EVENT HANDLER
USER-WRITTEN BUILDER AND CONTROLLER
MONITOR MANAGER
OBJECT SYSTEM SOFTWARE MONITOR
RCHM DRIVERS

Figure 3.5

3.2 COMPONENT DESCRIPTIONS

3.2.1 RCHM HARDWARE COMPONENTS

Corresponding to the three main types of computer system events, there are three types of event detectors in the RCHM:

"Value" events <--> Masked-word Range Comparator

Sequential events <--> Sequential logic unit

Combinational events <--> Combinational logic unit

The MASKED-WORD RANGE COMPARATOR determines whether a bit string, logically 'ANDed' with the specified mask, then regarded as a binary value, falls within two program-specified limits. There are five output lines to indicate whether the given string is above, below, or within the range, or whether it is equal to the upper or lower limit. Currently each comparator tests strings of at most sixteen bits, but the comparators have been designed so that four can be easily combined to test a 64-bit string. Four comparators are usually interconnected in a different way to form what we call a QUADRI-COMPARATOR which has four ranges and 16 (of the 17 possible) outputs. The outputs are: below the lowest value, equal to one of the range boundaries, within a range, between two ranges, or greater than or equal to the highest value.

The SEQUENTIAL LOGIC UNIT determines whether a sequence of events represented on its input lines is following a specified pattern. The pattern is defined by a regular ex-

pression which was specified by the experimenter, manipulated by the controlling computer, and stored in the sequential logic unit. The current design features eight inputs, eight outputs, and a maximum of 32 states; however, the unit could, like the other types of units, be of arbitrary size, determined by cost and need. Races are avoided by buffering the inputs and by using a synchronous design.

The COMBINATIONAL LOGIC UNIT determines whether the eight events represented on its input lines satisfy the Boolean expression specified by the experimenter's program. The functional representation is translated to a Karnaugh map and stored as such in the unit.

The CHARACTER DETECTOR receives characters serially by bit from the telecommunications link to which it is attached, thereby detecting telecommunications link events such as the start of a message. The detector tests to see if each character (5, 7, or 8 bit codes) matches one of the sixteen patterns specified by the controlling program. If so, the output line corresponding to the pattern matched is set to 'high'. This unit employs a simple associative memory to achieve the desired speed. The character detector and the sequential logic unit are used together to handle communications protocols -- e.g., to detect the header of a message or packet.

In a monitoring system there is a need to accurately

determine real time, to measure durations, and to obtain pulses of specified widths at specified regular intervals. Facilities for each are provided in the RCHM.

The NETWORK CLOCK serves as the source of accurate real time as well as very accurately spaced pulses. The current clock supplies pulses every 100 nanoseconds and selected multiples.

When a specified event occurs, the TIME STAMP UNIT records an identifier for the event, time of occurrence of the event, and sixteen indicators of the state of the object system when the event occurred. Currently, the output for an event contains 48 bits of information, the minimum time between recordable events is 200 nanoseconds, and the clock resolution is 100 nanoseconds.

The TIMER AND EVENT COUNTER counts the number of occurrences of the specified event, and measures the total amount of time the event occurred during the period of observation. Currently, each register has 32 bits, the timing resolution is 100 nanoseconds, and the maximum count rate is 10 MHz. The controlling program selects one of four clock rates. Under program control, each timer may be used as either a timer or an event counter. One can arrange for the controlling computer to be notified when a counter overflows by connecting the counter's overflow output to the input of an Interrupt Generator (See below).

The INTERVAL TIMER produces a 'high' output every X+Y

microseconds, and the output lasts for Y microseconds. X and Y are set under program control. The Interval Timer is used to indicate when to sample the system, should sampling rather than event-driven monitoring be desirable for a particular set of experiments.

In order to keep track of the fact that a set of events of interest has occurred or to aid in measuring the time between events, a set of flip flops is provided on the patch panels together with numerous standard logic gates.

To record monitored data until the controlling computer has an opportunity to access it, EVENT MEMORY is provided in the form of some semi-conductor memory and a mini-disk.

In most measurement experiments, the object is to obtain the distribution function for the quantity being measured. In order to facilitate obtaining this distribution function, HISTOGRAM GENERATORS and a MOMENT GENERATOR are included in the design of the RCHM. A Histogram Generator consists of a mask, a bank of comparators (or a CAM), and a corresponding bank of counters. When the input data falls within a range, the corresponding counter is incremented by one. The mask and ranges are set under program control.

The MOMENT GENERATOR, working with the output of the histogram generator, produces the first four moments of the distribution.

The INTERRUPT GENERATOR signals the nearest controlling

computer (whether RCHM controller or NMC or RNMC) whenever one of its input lines indicates that an event has occurred which requires computer intervention. Such events include overflow of counters or timers, data overrun in the time stamp unit, or events selected by the experimenter.

The PROGRAM-CONTROLLED SWITCH MATRICES connect a software specified input to one or more specified outputs. Using the switch matrices, several experiments can be set up by making the necessary patch panel connections in advance, then, under program control, setting up the switch matrices to perform one experiment at a time. To change from one experiment to another, one merely calls a routine to disconnect certain links, then calls another routine to make the required connections through the switch matrices. Using these matrices, probes are linked to event detecting units, which are connected to data gathering units, and these are connected to data recording and data reducing units. Two sizes of switch matrices have been used thus far: 8 x 8 and 16 x 4.

These programmable switch matrices make the CNMS feasible. Originally, it was hoped to use switch matrices exclusively, eliminating patch panels, but the size, speed and cost of the switch matrices required dictated the compromise of using patch panels plus switch matrices to make the necessary connections. When compared with the exclusive use of patch panels to achieve interconnection, the com-

promise reduces the time required to change from one experiment to the next and reduces the amount of human intervention required, but not to the level we had hoped. A less expensive switch matrix on an MSI chip is planned to further reduce the use of a patch panel.

3.2.2 NMC, RNMC, and RCHM SOFTWARE

As Figures 3.4 and 3.5 illustrate, the system software for the micro-processor that controls the RCHM and the system software for the NMC and RNMC have basically the same structure. However, the RCHM software is simpler and much smaller than that for the (R)NMC. The principal components of the software correspond with the types of functions to be performed. The message-switched Operating System structure is well-suited to the problem of controlling parallel tasks in multiple machines.

The USER INTERFACE MANAGER receives, analyses, and interprets commands typed by the user as the user sets up an experiment, interacts with it, and obtains and analyses the results. The command interpreter calls upon the other components of the system to serve the user.

The EXPERIMENT MANAGER schedules and supports the execution of experiment control programs written by users in the monitoring language.

The MONITOR MANAGER drives (or arranges for the RCHM controller to drive) the components of the RCHM to perform the monitoring activities requested by the user.

The RESOURCE MANAGER allocates main and auxiliary memory, and records and retrieves monitored data, experiment control programs and system programs.

The COMMUNICATIONS MANAGER in the NMC handles communications with the RNMCs it controls, with the load

generator, with the object network monitoring software, with the data analysis system, and with any RCHMs it controls directly. The Communications Manager in the RNMC provides communications with its controlling NMC and possibly the object network monitoring software. The Communications Manager in the RCHM controller is only concerned with its controlling NMC or RNMC.

The RESULTS MANAGER schedules and supports user-written or system-supplied routines to record, reduce, and analyse monitored data. Experiments requiring a great deal of data analysis send the data to a larger system that is more suitable for such analysis.* At the University of Waterloo, the Honeywell 6050 is used for this purpose. When the analysis is complete, the results are formatted by routines of the Results Manager selected by the user.

The MAINTENANCE MANAGER provides diagnostic routines and standard test packages for hardware and software of the CNMS. Using these routines, a knowledgeable user can interact directly with the components of the RCHMs and can perform reasonably complex experiments without having to write and compile experiment control programs. The Maintenance Manager also contains all routines necessary to handle CNMS hardware or software errors.

The QUEUE MANAGER provides the primary means of communicating between these software components. A service to

---

* Alternately, the NMC could be used at some sacrifice in efficiency and power.

be performed is requested by building a queue entry and asking the Queue Manager to put it in the appropriate place in the proper queue. A service which is to be done at a particular time is placed in the Queue Manager's queue until it is time to move it to the appropriate action queue.

A good first generation of most of these software components is being used to perform moderately complex experiments. A more sophisticated and complete version of the software is now being written.

4. USING THE CNMS

4.1 GENERAL PROCEDURE

In order to monitor a network using the CNMS, several functions must be performed:

A. Determine what is to be monitored and how to monitor it;

B. Determine what hardware probes (if any) are to be used, install them, and test them using an oscilloscope and the diagnostic software of the CNMS;

C. Determine what software monitoring tools (if any) are to be used in the object network, install them, and test them using the diagnostic software of the CNMS;

D. Decide whether known, controlled traffic is desired for the experiment and, if so, provide the load generator with the necessary scripts, distributions and line descriptors;

E. Define the software necessary to define and control the experiment and analyse the resulting data;

F. Set up the patch panel as required and debug the resulting combination of hardware and software using the diagnostic software of the CNMS;

G. Using the command language, initiate the experiment from a terminal attached to the NMC;

H. Interact with the experiment;

I. Obtain and interpret the results.

Today's computer architecture unfortunately demands that step B be performed by a computer hardware expert. Step C must be performed by a computer software expert, but we are trying to develop software monitoring primitives that will simplify this step. Step D requires knowledge of a use of a load generator and a text editing system. The text editor is needed to create the scripts of transactions required by the load generator. To do step E requires knowing how to write experiment control programs using the monitoring language and how to use the support routines provided to help control experiments and analyse results. Step F hopefully only requires knowledge of CNMS and the system being monitored, but could require expert help if problems exist with the hardware or with the object network software. Steps G, H, and I only require knowledge of CNMS and the characteristics of the object network.

Thus, monitoring a computer network is still a rather demanding task. However, once steps A through F have been performed for a set of experiments, the remaining steps can be performed without detailed knowledge of how the monitoring is being accomplished.

4.2 AN EXAMPLE

The following example was chosen from a set of experi-

ments that have been performed to measure, evaluate, and improve the CNMS and its components <22>. The example illustrates use of the RCHM and indicates a useful way of representing data.

As discussed in Section 3, an important component of the CNMS is the load generator. In order to study its behaviour, we assembled a two-node computer network. Each node executes the load generator to produce traffic for the other node through a variety of data links. The rate at which the load generator applies transaction traffic to each of its output lines is controlled by the user's choice of distribution function (e.g., exponential, uniform, hyperexponential) and by his choice of parameters for the distribution. A variety of line speeds can be produced in our networks laboratory for the data links joining the two nodes of this captive network. Thus, we can subject the load generator to a wide variety of tests while observing its behaviour.

We have found two histograms to be particularly useful for understanding and modelling the behaviour of computer systems or networks: System state vs. time in each state, and system state transition vs. number of such transitions. Both types of histograms are being produced as part of the measurement and evaluation of this network of load generators, but only the first will be presented here. (A paper describing the measurement and evaluation of the load

generator network is being prepared.)

The histogram shown in Figure 4.1 was produced by connecting the components of the RCHM to the load generator network as shown in Figure 4.2, and by writing and executing the experiment control program shown in Figure 4.3. The subroutines called by the experiment control program are primarily RCHM drivers. The variable "RCHM" indicates which RCHM is being used. (When this experiment was being performed, only one RCHM was assembled and working, but now there are two.) The variable "UNIT" indicates which of the several components of the same type in an RCHM is being addressed.

MEASUREMENT OF FILE TRANSFER FROM MACHINE #1 TO MACHINE #2

*** EXPERIMENT #4 RESULTS ***

THE FOLLOWING TABLE INDICATES THE TIME SPENT IN EACH OF THE 8 POSSIBLE STATES INVOLVING 2 CPU'S AND A COMMUNICATIONS LINK BETWEEN THEM.

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.97723467000000000D 07
TOTAL COMPUTE TIME: 0.48486980000000000D 07
STATE VECTOR IS
<LINE BUSY,CPU #2 BUSY,CPU #1 BUSY>
CPU #1 = MATH PDP11/20
CPU #2 = ENGINEERING PDP11/20

STATE	STATE VECTOR	TIME IN STATE	TIME/ REAL TIME	TIME/ COMPUTE TIME
0	000	0.45763595000D 07	0.4683D 02 %	0.9438D 02 %
1	001	0.73560010000D 06	0.7527D 01 %	0.1517D 02 %
2	010	0.10521093000D 07	0.1077D 02 %	0.2170D 02 %
3	011	0.23034536000D 07	0.2357D 02 %	0.4751D 02 %
4	100	0.31808860000D 06	0.3255D 01 %	0.6560D 01 %
5	101	0.58159200000D 05	0.5951D 00 %	0.1199D 01 %
6	110	0.31752550000D 06	0.3249D 01 %	0.6549D 01 %
7	111	0.38223860000D 06	0.3911D 01 %	0.7883D 01 %

	TOTAL TIME	TIME/ REAL TIME	TIME/ COMPUTE TIME
CPU #1 BUSY:	0.34794515000D 07	0.3561D 02 %	0.7176D 02 %
CPU #2 BUSY:	0.40553270000D 07	0.4150D 02 %	0.8364D 02 %
LINE BUSY:	0.10760119000D 07	0.1101D 02 %	0.2219D 02 %

Fig. 4.1

STATE VECTOR MEASUREMENTS

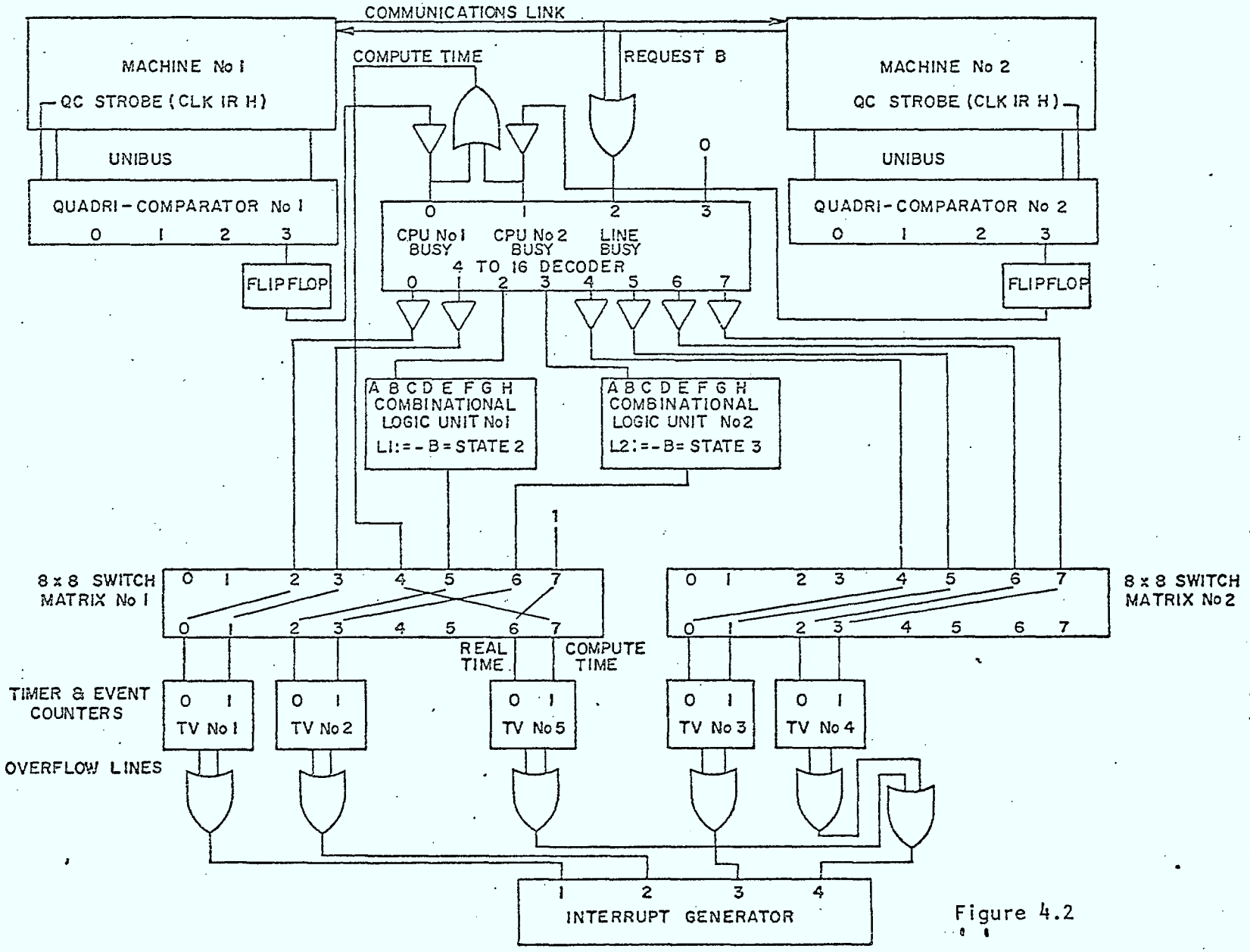


Figure 4.2

```
C
C                  FILE NAME:  DEXP4.EXP
C
C  PURPOSE OF EXPERIMENT #4
C
C
C        -TO DETERMINE THE TIME SPENT IN EACH OF
C         8 POSSIBLE STATES AS A RESULT OF USING THE
C         STATE VECTOR [LINE BUSY, CPU #2 BUSY, CPU #1 BUSY]
C         WHERE THE LINE IS A COMMUNICATIONS LINK BETWEEN
C         THE TWO CPU'S.
C
C        -TO FILL IN THE FOLLOWING TABLE:
C         STATE     TIME IN       TIME/          TIME/
C                    STATE       REAL TIME     COMPUTE TIME
C
C         WHERE REAL TIME=>TOTAL EXPERIMENT TIME, AND
C         COMPUTE TIME => CPU # 1 OR CPU #2 BUSY.
C
C  THE SYSTEM IS DEFINED TO BE BUSY, OR DOING USEFUL
C  COMPUTING, IF ONE OR BOTH OF THE MACHINES IS
C  EXECUTING OUTSIDE OF THE PROGRAM WAIT LOOP.
C
C
C
        SUBROUTINE DEXP4
C
        EXTERNAL TVOVFL
        IMPLICIT INTEGER(A-Z)
        INTEGER TVOVFO(5), TVOVFL(5)
        COMMON /CONFIG/NODE, UNIT
        COMMON /TVCOM/TVOVFO(5),TVOVFL(5)
C
C  QUADRA-COMPARATORS MUST BE SET UP, USING
C  THE TEST PROGRAM.  FOR COMPUTE TIME, SET RANGE #3 TO
C  THE ADDRESSES OF THE WAIT LOOP.
C
        WRITE(6,12)
12      FORMAT(' USE TEST PROGRAM TO SPECIFY QC RANGE #3 = WAIT LOOP',/
        1 '  ON BOTH MACHINES',/' ')
C
C  COMPUTE TIME => CPU#1 BUSY  OR  CPU #2 BUSY
C        => (OUTSIDE CPU #1 WAIT LOOP)  OR
C          (OUTSIDE CPU #2 WAIT LOOP)
C
C
C
C
C SET UP LOGIC UNITS.
C
C  LOGIC UNIT #1 = -B = STATE 2 = 1SW815
C
        L1:=-B
C
C  LOGIC UNIT #2 = -B = STATE 3 = 1SW816
C
        L2:=-B
```

Figure 4.3

```
C
C  SET UP 8X8 SWITCH MATRICES
C
C  STATES
        S0=2
        S1=3
        S2=5
        S3=6
        S4=4
        S5=5
        S6=6
        S7=7
C
C  TIMER & EVENT COUNTERS
C
        TV1B0=0
        TV1B1=1
        TV2B0=2
        TV2B1=3
        TV3B0=0
        TV3B1=1
        TV4B0=2
        TV4B1=3
        TV5B0=6
        TV5B1=7
        RTIME=7
        CTIME=4
C
        UNIT=1
        CALL SW8DIS
        CALL SW8CON(S0, TV1B0, S1, TV1B1, S2, TV2B0, S3, TV2B1,
        1 RTIME, TV5B0, CTIME, TV5B1)
C
        UNIT=2
        CALL SW8DIS
        CALL SW8CON(S4, TV3B0, S5, RV3B1, S6, TV4B0, S7, TV4B1)
C
C  SET UP TIMER & EVENT COUNTERS
C
        DO 35 UNIT=1, 5
35      CALL TVSET(1,1,1,3,1,1)
C
C
C  SET UP INTERRUPT GENERATOR AND CLEAR OVERFLOW COUNTS.
C
        DO 40 UNIT=1,4
        TVOVF0(UNIT)=0
        TVOVFL(UNIT)=0
40      CALL IGSET(TVOVFL, UNIT)
C
C  TV #5 OVERFLOW LINES ARE 'OR'ED WITH THOSE OF TV #4, SINCE
C  ONLY HAVE FOUR INTERRUPT LINES.  A SPECIAL CHECK IS MADE IN
C  'TVOVFL'.
        TVOVF0(5)=0
        TVOVFL(5)=0
C
        RETURN
        END
```

5. APPLICATIONS OF THE CNMS

Besides its intended use of monitoring a computer network, the system has been used successfully to monitor the activities of a computer system. Several experiments have been and are being performed, including:

A. Using the monitor to locate frequently used code and system bottlenecks in DOS-11 and Fortran as programs are compiled and executed;

B. Determining the loading on the PDP-11 UNIBUS;

C. Determining the frequency of execution of each of eight classes of instructions for different kinds of programs in order to compare our results with those obtained by Schreiber and Klar at the University of Erlangen <23>;

D. Validating a mathematical model of two transaction-oriented data base management systems interacting with each other and sharing data;

E. Locating inefficiencies in a computer network simulator implemented to work in parallel on three interconnected PDP-11 computers.

Reports describing the results of these experiments are being prepared. Many other experiments are planned, e.g., measuring the swapping activities of various PDP-11 operating systems, observing Honeywell's GCOS executing on a 6050, watching VM/370, monitoring the performance of an experimental loop network joining our laboratory with

laboratories in Toronto and Ottawa, as well as monitoring our campus network.

Eventually we think that some form of a CNMS will be a vital component in an automated maintenance system for computer networks, helping to detect and diagnose malfunctions and bottlenecks in hardware and software semi-automatically. Such a system is being designed, and implementation of a prototype is planned during 1975-6. Furthermore, we anticipate that a form of CNMS will eventually be an important component in an adaptively controlled computer network. We are working toward these goals as well as toward "simply" monitoring the performance of computer networks and systems.

Components of the CNMS are being employed in a novel experiment to test the feasibility of providing hardware assistance to an information retrieval system. The results of this experiment should be available in about 18 months.

Finally, it appears that the CNMS, ,with minor software and hardware modifications, can be used to monitor a set of electronic switching offices for telephones <24>.

6. SUMMARY, CONCLUSIONS, AND FURTHER WORK

In Section 2 it was concluded that an integrated monitoring system is preferable to a set of unrelated, uncoordinated monitoring tools, because few people who need to monitor a system or network are hardware, software, and statistics experts whose primary interest in life is to monitor the system or network. In Section 3 the Computer Network Monitoring System being created at the University of Waterloo was described. Section 4 presented a simple example to illustrate use of the system, and Section 5 mentioned several experiments that are being performed using the CNMS.

Sandra Mamrak, in her forthcoming article entitled "Performance Evaluation in Computer Networks: A Survey"<1>, states: "Actual system measurements, analyzed using statistical techniques and used to improve queueing and simulation models, have been relatively neglected. (This neglect may be due in part to the unavailability of tools for making desired observations of dynamic systems and of statistically significant test environments.)" It is hoped that the CNMS described in this paper will be a good first step toward satisfying this need.

As promised in Section 1, we have evaluated our CNMS based on our experience in using it. Table 6.1 is our evaluation of the system as it stands at this writing and our prediction of an evaluation as the system should be at

the beginning of 1976. The evaluation is based on the nine characteristics of an ideal CNMS listed in Section 2. The scores range from -5 to +5, with -5 meaning "Terrible, couldn't be worse", +5 meaning "Excellent, couldn't be better", and 0 being the borderline between being acceptable and unacceptable.

As the scoring indicates, the main problems with the CNMS are cost and the continuing need for a patch panel. We anticipate that both problems can be solved in time, especially considering the rate at which the cost of logic is dropping and recent developments in solid state switching for data communications.

The prototype RCHMs use TTL logic, which limits our resolution to 10 MHz; however, some of the newer components contain Schottky logic in order to monitor the microprogrammed PDP-11/45.

A few conclusions can be drawn from our experience thus far:

A. The hardware monitor (RCHM) works and is not prohibitively expensive to build (i.e., $10,000 to $100,000, depending on which modules are included and in what quantity).

B. The modular components plus the bus architecture make it easy to insert or remove components as desired. Above the cost of a basic monitor, the cost of the monitor increases as the complexity of

the experiments to be performed increases.

C. It is not difficult to write control programs for the monitor, even without the monitoring language, because each component is addressed as a set of memory locations on the controlling PDP-11. The primitives of the current, embryonic monitoring language are simply Fortran subroutine calls. The routines themselves are written in either Fortran or Assembler for the PDP-11.

D. The diagnostic hardware and software that we included in the CNMS continually proves its value. Many monitoring tools do not include such error detection and diagnostic tools. We have found that it is definitely worth our time to quickly run through a set of routine hardware and software test programs before running an experiment.

E. A system hardware expert would not be required to install the probes if computer manufacturers provided an accessible panel of probe points on their products.

The security and privacy questions that arise from considering the widespread use of CNMSs are thought provoking, to say the least. and could be the subject of a long discourse. A simple way to prevent unauthorised snooping is to keep the phone numbers of the RCHMs of the CNMS a well-guarded secret.

Creating a network monitoring system is an ambitious project. Although much has been accomplished since the project began in mid 1971, a great deal of work remains to be done, some of which is listed here:

A. Develop a theory of computer monitoring. Possible topics include extending the work of Morgan and Sutton <35> in formally defining events in terms of system states, providing a formal basis for deciding when the performance of a system is acceptable, and creating a theoretical basis on which to build monitoring systems.

B. Find solutions to the problems of determining exactly when an event occurred and deciding the order in which two nearly simultaneous events occurred in separate computers of the network.

C. Develop an easily used, extensible language for defining and controlling monitoring experiments as well as analysing the data. Our Fortran-based language is only a poor first step toward this goal.

D. Determine what parameters characterise the performance and workload of a computer system or network. Some form of so called Kiviat graph might be useful to represent the performance of the network in terms of these parameters <36>.

E. Create a self-monitoring computer system, and

then a self-monitoring computer network. A self-monitoring computer system is one that includes special hardware (e.g., micro-programmed special instructions) to aid the system in observing its own activities. Similarly, a self-monitoring computer network would contain special hardware and firmware to help the network observe its own activities.

F. Create an adaptive computer system that monitors its workload and its performance while continually adjusting its resource multiplexing parameters accordingly. A mathematical model of such a system has been analysed by Gelenbe et al <25>.

And much, much more.

ACKNOWLEDGEMENTS

The hardware skills of J. Runge and K. Jedermann, and the software talents of F. Mellor, K. Pammett, J. Palpraman, D. Sutton, W. Colvin, and R. Shen have contributed immensely to the project.

BIBLIOGRAPHY

(1) S. Mamrak, Performance Evaluation in Computer Networks, to be published in Computing Surveys.

(2) D. Morgan, W. Banks, W. Colvin and D. Sutton, A Performance Measurement System for Computer Networks, Proceedings of IFIP Congress 74, pages 29-33.

(3) G. D. Cole, Computer Network Measurements - Techniques and Experiments. UCLA, Oct. 1971, NTIS #AD-739-344.

(4) C. T. Apple, The Program Monitor - A device for Program Performance Measurement, Proc. of the ACM 20th National Conference, August 1965, pp. 66-75.

(5) R. A. Aschenbrenner, et al, Neurotron Monitor System, AFIPS, v. 39 (FJCC) 1971, pp. 31-37.

(6) A. J. Bonner, Using System Monitor Output to Improve Performance, IBM Systems Journal, v. 8 #4, 1969, pp. 290-298.

(7) D. T. Bordsen, Univac 1108 Hardware Instrumentation System, ACM SIGOPS Workshop on System Performance Evaluation, Harvard U., April 1971, pp. 1-29.

(8) G. Estrin, et al, SNUPER Computer, AFIPS, v. 30 (SJCC) 1967, pp. 645-656.

(9) L. E. Hardt, G. J. Kipovitch, Choosing a System Stethoscope, Computer Decisions, Nov. 1971, pp. 20-23.

(10) T. Y. Johnston, Hardware vs. Software Monitors Proc. of SHARE, XXXIV, v. 1, March 1970, pp. 523-547.

(11) K. W. Kolence, Software Physics and Computer Performance Measurements, Proc. of the 1972 ACM Annual Conference, pp. 1024-1040.

(12) H. Lucas, Performance Evaluation and Monitoring, Computing Surveys, v. 3 #2, Sept. 1971, pp. 79-91.

(13) R. W. Murphy, The System Logic and Usage Recorder, AFIPS, v. 35 (FJCC) 1969, pp. 219-229.

(14) C. D. Warner, Hardware Techniques, ACM SIGCOSM Newsletter #5, August 1970, pp. 5-11.

(15) M. D. Abrams, Consumer-Oriented Measurements of Computer Network Performance, Proceedings of National

Telecommunications Conference, IEEE Communications Society, December 1974, San Diego, Calif.

(16) W. Banks, D. Morgan, A Computer Controlled Hardware Monitor: Hardware Aspects, Proceedings of International Meeting on Minicomputers and Data Communications, Liege, Belgium, January 1975.

(17) J. Hughes, D. Cronshaw, On Using a Hardware Monitor as an Intelligent Peripheral, Xerox Corporation, October 1973.

(18) Y. Kalef and M.Melman, Performance Analysis of the Golem B Computer System, Technical Report from the Department of Applied Mathematics, The Weizmann Institute of Science, Israel.

(19) P. R. Sebastian, Hybrid Events Monitoring Instrument, Proceedings of SIGMETRICS 74, October 1974.

(20) B. W. Kernighan, A Tutorial Introduction to the Language B; Computer Science Technical Report, Bell Laboratories, Murray Hill, N. J.

(21) F. Mellor, A General Purpose Load Generator, University of Waterloo, April 1974.

(22) D. Morgan, D. Goodspeed, R. Kolanko, Demonstration of a Programmable Hardware Monitor, CCNG External Report, University of Waterloo, October 1974.

(23) R. Klar and H. Schreiber, unpublished report, University of Erlangen, Institute of Computer Science.

(24) R.E. Machol, Acquiring Data for Network Planning and Control, Bell Laboratories Record, v. 52, no. 9, October 1974, pp. 279-285.

(25) M. Badel, E. Gelenbe, J. Lenfant, J. Leroudier, D. Potier, Adaptive Optimization of the Performance of a Virtual Memory Computer, Proceedings of SIGMETRICS 74, October 1974.

(26) Comress: Dynaprobe 7900: System Specifications, Comress Report No. CR4-0031.

(27) J. M. Grochow, Real Time Graphic Display of Time-Sharing System Operating Characteristics, AFIPS, v. 35 (FJCC) 1969, pp. 379-386.

(28) J. H. Salzer, J. W. Gintell, The Instrumentation of Multics; Second Symposium on Operating System Principles, Oct. 1969, pp. 167-174.

(29) K. W. Kolence, System Improvement by System Measurement, Data Base, Winter, 1969, pp. 6-11.

(30) E. F. Miller, An Experiment in Hardware Monitoring, General Reseanch Corporation, RM-1517, July 1971.

(31) R. G. Canning, Savings from Performance Monitoring, EDP Analyzer, Sept. 1972.

(32) R. L. Patrick, Measuring Performance, Datamation, v. 10, #7, July 1964, pp. 24-27.

(33) S. H. Fuller. R. J. Swan, W. A. Wulf, The Instrumentation of C.mmp, A Multi-(Mini) Processor, IEEE Intl. Computer Society Conference, February 1973, pp. 173-176.

(34) E. L. Burke, A Computer Architecture for System Performance Monitoring, First Annual SIGME Symposium on Measurement and Evaluation, Feb. 1973, pp. 161-169.

(35) D.A. Sutton and D. E. Morgan, The Monitoring of Computer Systems and Networks: A Summary and a Proposal, CCNG External Report, April 1974, University of Waterloo.

(36) J.D. Noe and N. W. Runstein, Develop Your Computer Performance Pattern, Proceedings of SIGMETRICS 74, October 1974.

APPENDIX D

DEMONSTRATION OF A PROGRAMMABLE HARDWARE MONITOR

Demonstration of a Programmable Hardware Monitor

Dr. David E. Morgan
Dale P. Goodspeed
Richard Kolanko

Computer Communications Networks Group
University of Waterloo
Waterloo, Ontario, Canada
October, 1974.

The programmable hardware monitor developed by the University of Waterloo's Computer Communications Networks Group(CCNG) is intended to be part of a computer network monitoring system. Detailed information about this system is contained in the references, and it is assumed that the reader is familiar with some of this material. All of the monitoring for this demonstration is done locally, within the CCNG Laboratory. Software currently being developed will allow for remote monitoring to be done at the various nodes of a computer network, as described in the references.

Figure 1 shows the configuration for monitoring. The object computer system consists of two PDP11/20 computers, with a communication link between them.

Figure 2 is the wiring diagram used for performing measurements on a single computer, while Figure 3 is the configuration used when measurements are performed on both computers.

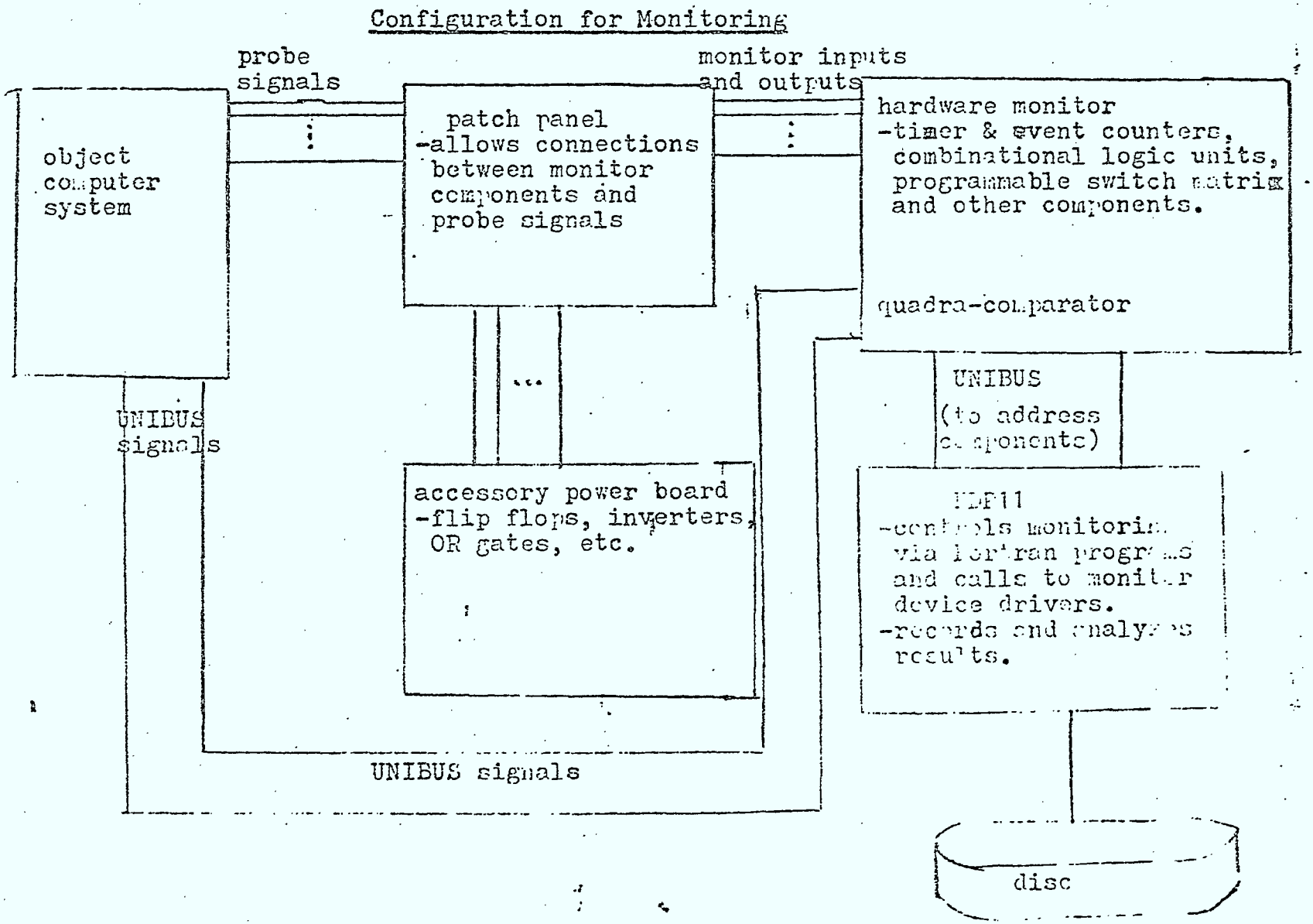

Configuration for Monitoring
probe signals
monitor inputs and outputs
object computer system
patch panel
-allows connections between monitor components and probe signals
hardware monitor
-timer & event counters, combinational logic units, programmable switch matrix and other components.
quadra-comparator
UNIBUS (to address components)
UNIBUS signals
accessory power board
-flip flops, inverters, OR gates, etc.
PDP11
-controls monitoring via Fortran programs and calls to monitor device drivers.
-records and analyzes results.
UNIBUS signals
disc

Figure 1A

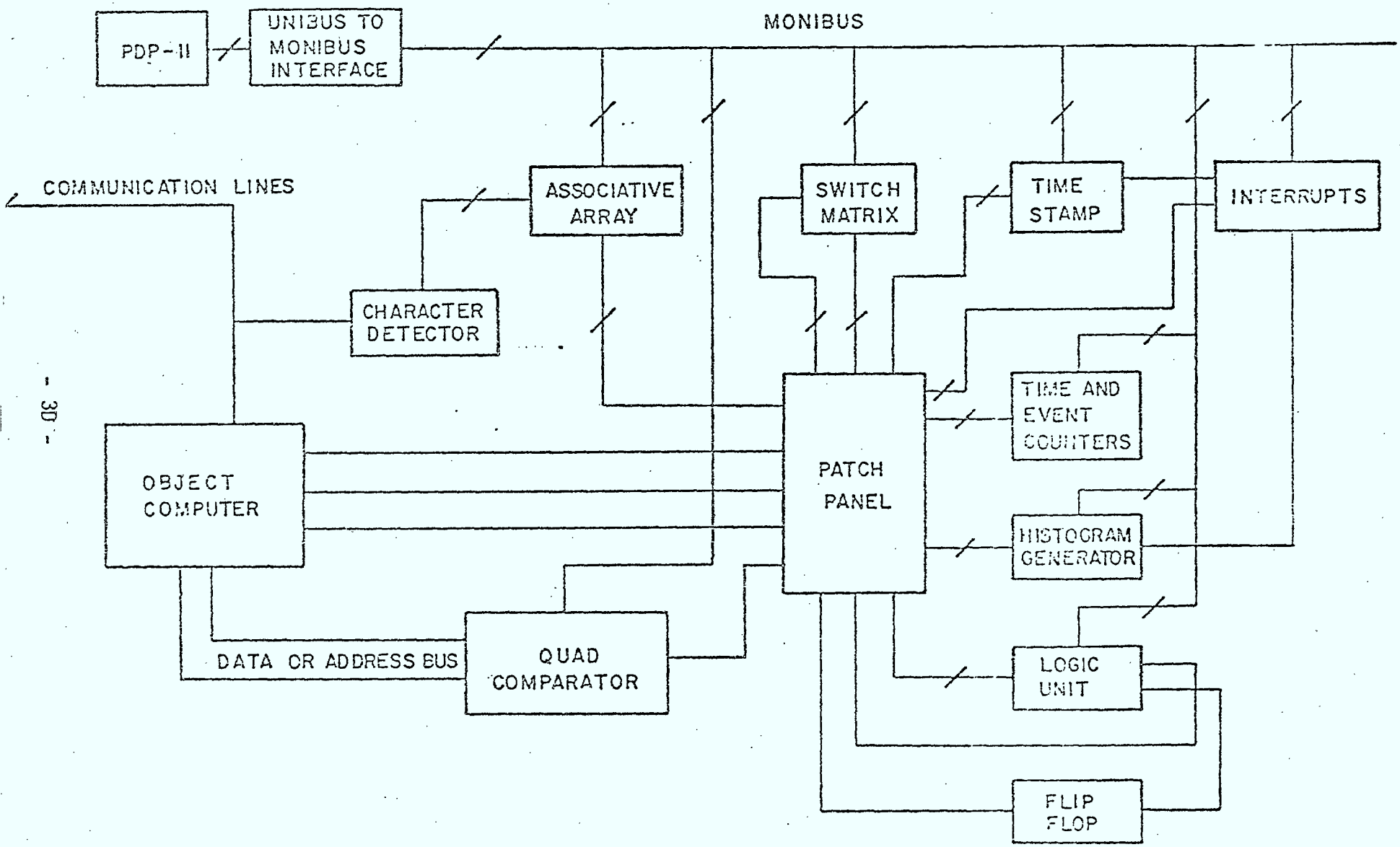


Fig. 16 GENERALIZED MONITOR

MEASUREMENT OF PROCESSOR MAJOR STATES AND INSTRUCTION TIME

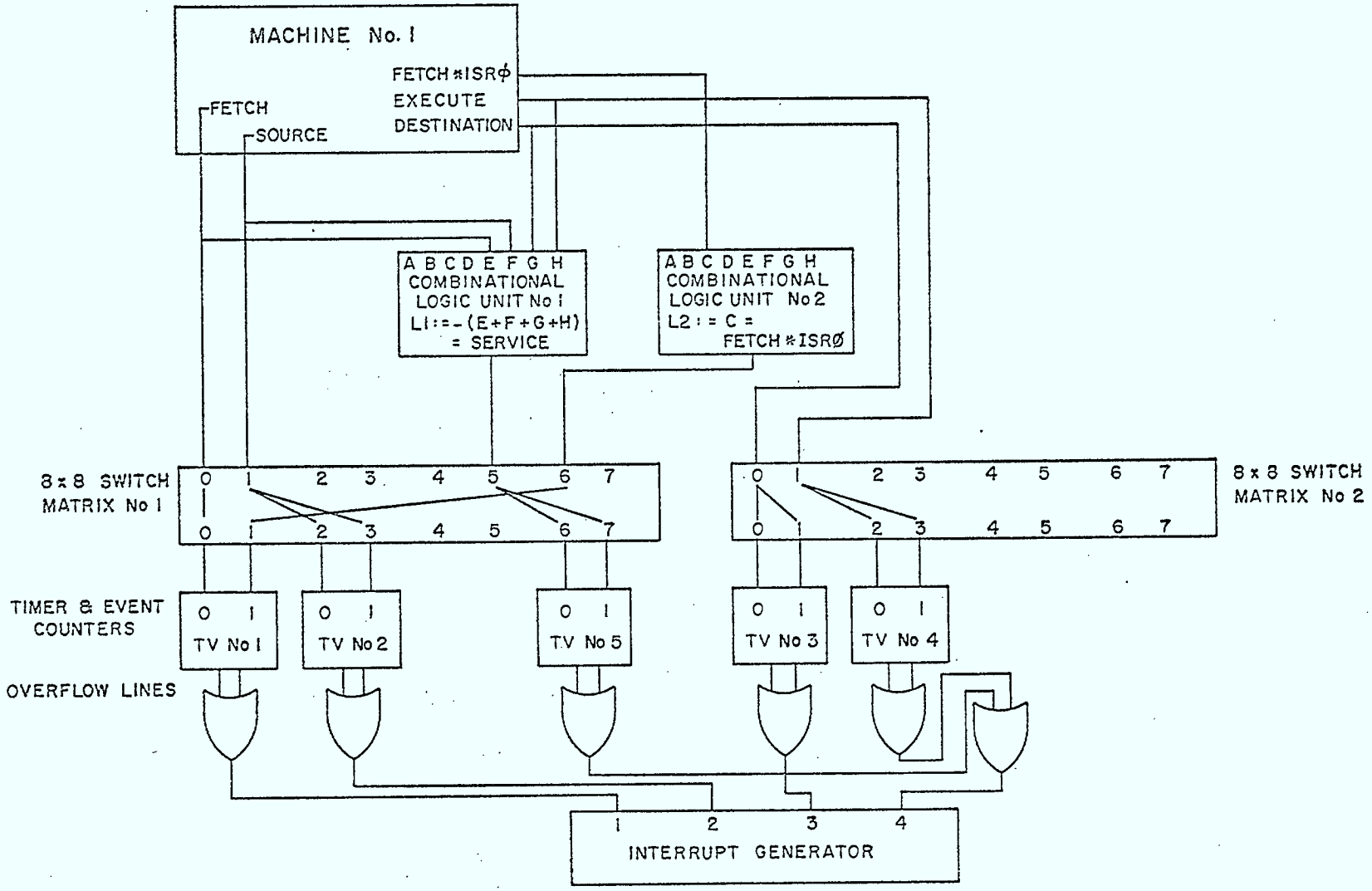


FIGURE 2A

MEASUREMENT OF UNIBUS ACTIVITY

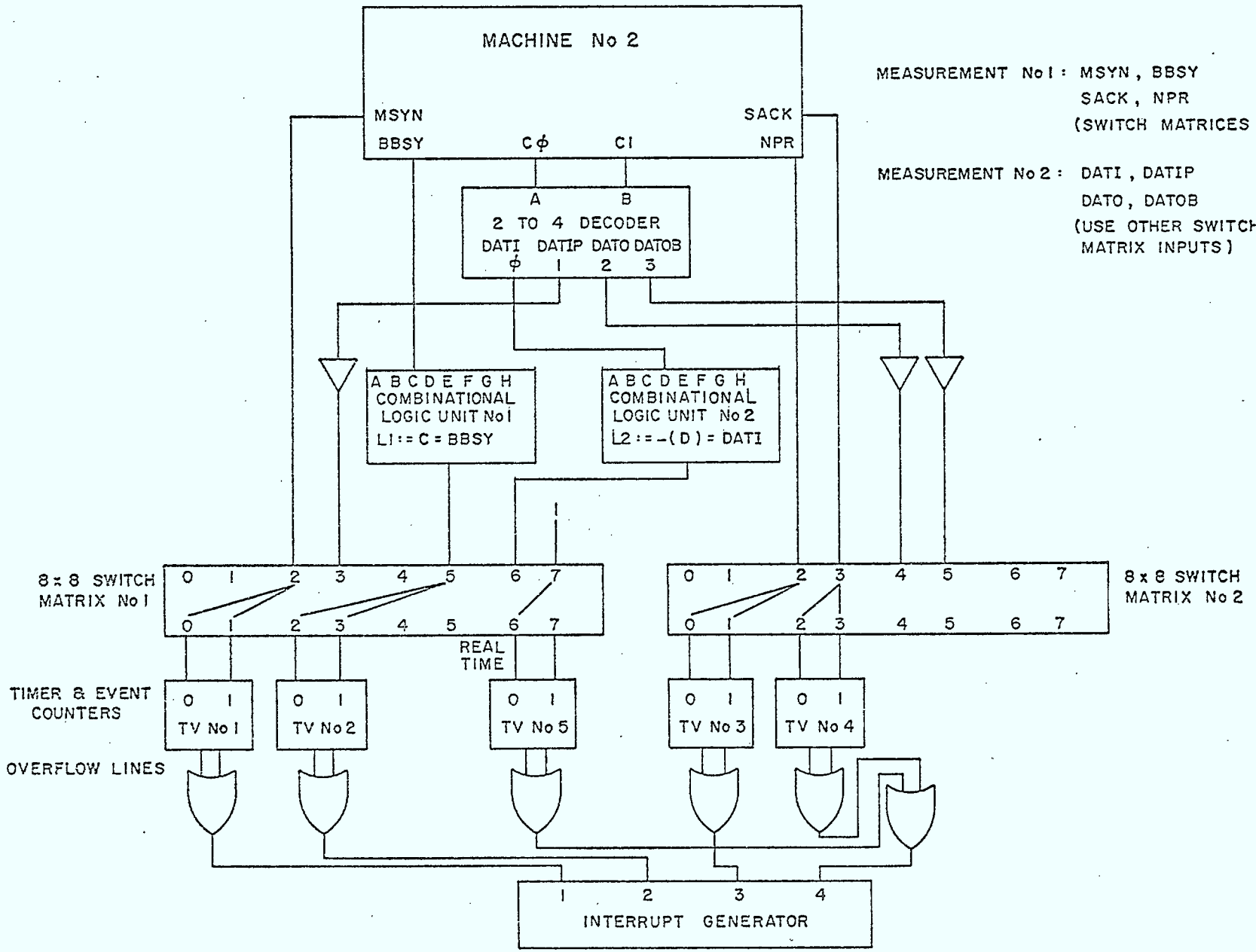


FIGURE 2B

MEMORY ACTIVITY MEASUREMENTS

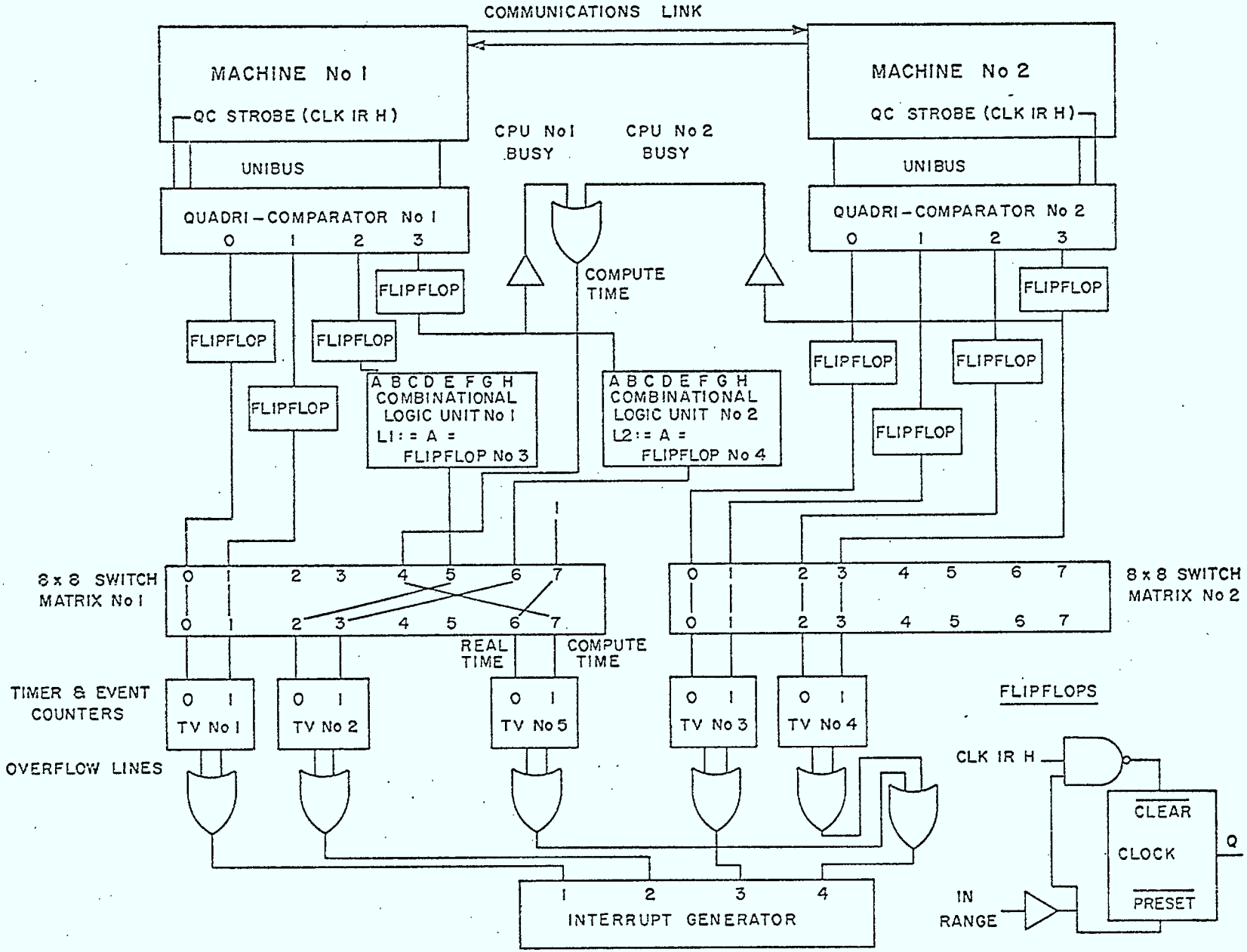


FIGURE 3

The User Interface

The user interface of the monitor control program allows measurement experiments to be performed with a minimum of difficulty. The keyboard commands available to the user are:

EXn: This sets up the monitor components for experiment #n. The user's Fortran subroutine, which contains calls to the drivers, is executed. This establishes connections in the programmable switch matrices, and selects the speed to be used by the timers.

GO: This allows the actual monitoring to begin, by setting the 'go' bits of the timer and event counters. The monitoring may continue for a specified time interval, or until the 'ST' command is given.

ST: This stops the experiment.

AN: This command permits the measurement data just collected to be analyzed. The user's Fortran analysis subroutine is executed, and results are sent to the keyboard, or may be saved on disc. The user may analyze results while the experiment is still running, or may wait until it has stopped.

CL: This command is used only occasionally. It causes the timer and event counter data buffers to be cleared, and resets any interrupts. Normally, these functions are done under program control.

TE: Using this command, the user may access all of the hardware diagnostic programs. Aside from verifying that the hardware is operating correctly, the diagnostic commands may also be used to set up entire experiments. This provides an alternative to writing the Fortran programs needed with the 'EX' command.

FI: Control is returned to the PDP11 operating system(DOS) when the user is finished monitoring.

The Demonstration Experiments

The demonstration consists of five measurement experiments. Three of these are done on only one of the machines (two on Machine #1 and one on Machine #2). The last two experiments perform simultaneous measurements on both computers, while the machines talk to each other across the communication link. In between the two sets of experiments, a few(16) wiring changes must be made on the patch panel. The interrupt generator inputs and some of the inputs to the programmable switch matrices are changed. These simple changes must be made because of the constraints imposed by the current number of components installed in the monitor. With an additional switch matrix and interrupt generator installed, the changes would not be necessary, and all five experiments could be performed without ever having to modify the patch panel wiring. All connections could then be made via programmed changes to a switch matrix connection. When necessary, a combinational logic unit could be used to select an appropriate input to a switch matrix. The preceding two techniques are already being used to maximum advantage for the demonstration experiments.

The wiring for the patch panel and the accessory board might appear to be quite complex, especially for a 'programmable' hardware monitor. To understand the reason for this, the observer must realize that the wiring is arranged so that five different experiments can be performed with only a minimum of wiring changes needed. The experiments require using fourteen different probes attached to two computers, and also the UNIBUS address lines of each machine. On the accessory board, approximately half of the wiring is used to invert signals. The flip flops and decoder chips currently in use operate on a 'low' true signal, while the monitor components use 'high' true logic. Accessory modules which operate on a 'high' true signal are currently being built, so that much of the accessory board wiring will no longer be necessary.

The experiments to be performed are:

1) measurement of CPU major states and average instruction time on Machine #1.

2) measurement of UNIBUS activity on Machine #2.

3) measurement of Machine #1 while a simple communication takes place between it and Machine #2.

4) simultaneous measurements of Machine #1 and Machine #2. Both machines are driven by a load generator, which

causes the machines to talk to each other. The state vector produced is:

<line busy, CPU#2 busy, CPU#1 busy>

5) simultaneous measurements of Machine #1 and Machine #2. Memory activity is examined on both machines, and a table is produced showing the percentage of time spent in the various ranges specified.

The experiments are discussed in more detail in the following sections. Included in the discussions are listings of the programs used to obtain the measurements, and copies of the some of the results. More detailed comments about some of the results are given in reference #2, with explanations given for those cases where the results might appear invalid. Typically, the explanation involves having a detailed knowledge of the PDP11 architecture. Rather than go into those details here, emphasis is instead put upon the motivation behind each experiment. For all of the experiments, one of the subgoals is to gain experience which can be applied to the monitoring of computer networks.

Experiment #1

The CPU major states (fetch, source, destination, execute, service) of Machine #1 are monitored. Results obtained include:

1) average time in state.
2) percentage time in state.
3) percentage of instructions that enter state.
4) average instruction time.

The results can be used to verify the PDP11/20 specifications provided by Digital Equipment Corporation. If the source and destination states are entered too frequently, it is possible that the program efficiency could be improved by better use of the register-register mode of addressing. Entries to service state provide a partial measure of disc activity.

MEASUREMENT OF MACRO COMPILER ON MACHINE #1

*** EXPERIMENT #1 RESULTS ***

TIME UNITS = MICROSECONDS

TOTAL TIME SPENT IN ALL MAJOR STATES = 0.26606146200 08

	FETCH	SOURCE	DESTINATION	EXECUTE	SERVICE
TOTAL TIME IN STATE:	0.1160D 08	0.4415D 07	0.4635D 07	0.5804D 07	0.1562D 06
# OF TIMES ENTERED:	0.7188D 07	0.2537D 07	0.2896D 07	0.6397D 07	0.4369D 06
AVERAGE TIME IN STATE:	0.1613D 01	0.1740D 01	0.1601D 01	0.9073D 00	0.3576D 00
% TIME IN STATE:	0.4358D 02	0.1659D 02	0.1742D 02	0.2181D 02	0.5872D 00
% OF INSTRUCTIONS THAT ENTER STATE:	0.1000D 03	0.3530D 02	0.4029D 02	0.8900D 02	0.6078D 01

AVERAGE INSTRUCTION TIME =
(TOTAL TIME IN ALL STATES) / (# OF FETCHES) = 0.3702D 01

```
C
C                FILE NAME: DEXP1.EXP
C
      SUBROUTINE DEXP1
C
C PURPOSE OF EXPERIMENT #1
C
C       -TO OBTAIN THE TIME SPENT IN THE PROCESSOR MAJOR STATES
C       -USE TV #1 TO RECORD THE TIME SPENT IN FETCH MAJOR STATE
C       -USE TV #2 TO RECORD THE TIME SPENT IN SOURCE MAJOR STATE.
C       -USE TV #3 TO RECORD THE TIME SPENT IN DESTINATION MAJOR STATE.
C       -USE TV #4 TO RECORD THE TIME SPENT IN EXECUTE MAJOR STATE.
C       -USE TV #5 TO RECORD THE TIME SPENT IN SERVICE MAJOR
C        STATE.
C
C
C
      EXTERNAL TVOVFL
      IMPLICIT INTEGER(A-Z)
      INTEGER TVOVFU(5),TVOVF1(5)
      COMMON /CONFIG/NODE,UNIT
      COMMON /TVCOM/TVOVFU(5),TVOVF1(5)
C
C INPUTS TO 8X8 SWITCH MATRICES
C
      FETCH=0
      FAISRU=6
      SOURCE=1
      DEST=0
      EXEC=1
      SERVIC=5
C
C
C TIMER & EVENT COUNTERS (BUFFER 0 & 1)
C
      TV1B0=0
      TV1B1=1
      TV2B0=2
      TV2B1=3
      TV3B0=0
      TV3B1=1
      TV4B0=2
      TV4B1=3
      TV5B0=6
      TV5B1=7
C
C SET UP SWITCH MATRICES.
C
      UNIT=1
      CALL SW8DIS
      CALL SW8CON(FETCH,TV1B0,FAISRU,TV1B1,SOURCE,TV2B0,SOURCE,TV2B1,
     1 SERVIC,TV5B0,SERVIC,TV5B1)
C
      UNIT=2
      CALL SW8DIS
      CALL SW8CON(DEST,TV3B0,DEST,TV3B1,EXEC,TV4B0,EXEC,TV4B1)
C
C
C SET UP LOGIC UNIT #1 = -(E+F+G+H) =
C  -(FETCH+SOURCE+DEST+EXEC) = SERVICE STATE = 1SW8I5
C
      L1:=-(E+F+G+H)
C
C SET UP LOGIC UNIT # 2 = C = FETCH*ISRU = 1SW8I6
```

```
      L2:=C
C
C
C SET UP TIMER & EVENT COUNTERS.
C
      DO 20 UNIT=1,5
20    CALL TVSET(1,0,1,3,1,1)
C
C SET UP THE INTERRUPT GENERATORS AND ZERO OVERFLOW
C COUNTS IN CASE THERE'S TIMER & EVENT COUNTER OVERFLOW.
C
      DO 30 UNIT=1,4
      TVOVF0(UNIT)=0
      TVOVF1(UNIT)=0
30    CALL IGSET(TVOVFL,UNIT)
      TVOVF0(5)=0
      TVOVF1(5)=0
C
C BLOCK INTERRUPTS GENERATED BY QUADRA-COMPARATOR USED IN
C DEXP3.EXP
C
      UNIT=1
      CALL QCSET(0)
      UNIT=2
      CALL QCSET(0)
C
      RETURN
      END
```

```
C
C                FILE NAME: DEND1.EXP
C
C THIS IS THE ANALYSIS ROUTINE FOR EXPERIMENT #1
C
      SUBROUTINE DEND1
C
      IMPLICIT INTEGER(A-Z)
      INTEGER TVU(2),TV1(2),TVOVFU(5),TVOVF1(5)
      LOGICAL IGFLG
      REAL*8 DPTVU(5),DPTV1(5),TSUM,TBYTS(5),EBYFET(5),
     1 MAX,AVGTIS(5),AVGITM
C
      COMMON /CONFIG/NODE,UNIT
      COMMON /TVCOM/TVOVFU(5),TVOVF1(5)
      COMMON /PHMSYS/OUTUNT,IGFLG
      DATA MAX/4294967296.D0/
C
      WRITE(OUTUNT,500)
500   FORMAT(' *** EXPERIMENT #1 RESULTS ***'/' ',/' ',
     1'TIME UNITS = MICROSECONDS')
C
C READ RESULTS, ADJUST FOR POSSIBLE OVERFLOW, AND
C CONVERT TIME TO MICROSECONDS
      TSUM=0.D0
      DO 505 UNIT=1,5
      CALL TVREAD(TVU,TV1)
      CALL DI2DF(TVU,DPTVU(UNIT),TV1,DPTV1(UNIT))
      DPTVU(UNIT)=DPTVU(UNIT) + TVOVFU(UNIT)*MAX
      DPTV1(UNIT)=DPTV1(UNIT) + TVOVF1(UNIT)*MAX
      DPTVU(UNIT)=DPTVU(UNIT) /10.D0
505   TSUM=TSUM + DPTVU(UNIT)
C
      WRITE(OUTUNT,520)TSUM
520   FORMAT(' ',/' ','TOTAL TIME SPENT IN ALL MAJOR STATES = ',D17.10)
C
      DO 530 UNIT=1,5
      TBYTS(UNIT)=DPTVU(UNIT) / TSUM * 100.D0
      EBYFET(UNIT)=DPTV1(UNIT) / DPTV1(1) * 100.D0
      AVGTIS(UNIT)=0.D0
530   IF (DPTV1(UNIT).NE.0.D0)  AVGTIS(UNIT)=DPTVU(UNIT) / DPTV1(UNIT)
C
      WRITE(OUTUNT,540)
540   FORMAT(' ',/' ',13X,'FETCH',5X,'SOURCE',5X,'DESTINATION',3X,
     1 'EXECUTE',5X,'SERVICE')
      WRITE(OUTUNT,544) DPTVU,DPTV1,AVGTIS,TBYTS,EBYFET
544   FORMAT(' ',/' ','TOTAL TIME',/' ','IN STATE: ',4(D11.4,1X),D11.4,
     1 //' # OF TIMES',/' ENTERED:  ',4(D11.4,1X),D11.4,//' AVERAGE TIME',
     1 /' IN STATE: ',4(D11.4,1X),D11.4,//' % TIME',/' IN STATE: ',
     1 4(D11.4,1X),D11.4,//' % OF INSTRUCTIONS',/' THAT ENTER',
     1 /' STATE:    ',4(D11.4,1X),D11.4)
C
C COMPUTE AVERAGE INSTRUCTION TIME
C
      AVGITM=TSUM / DPTV1(1)
      WRITE(OUTUNT,546) AVGITM
546   FORMAT(' ',/' AVERAGE INSTRUCTION TIME = ',/'  (TOTAL',
     1 ' TIME IN ALL STATES) / (# OF FETCHES) = ',D11.4)
C PRINT OVERFLOW COUNTS.
C
      WRITE(OUTUNT,550)
550   FORMAT(' ',/' ','OVERFLOW COUNTS FOR TIMER & EVENT COUNTERS',
     1 /' ',3X,'UNIT',3X,'BUFFER',3X,'COUNT')
      DO 552 UNIT=1,5
552   WRITE(OUTUNT,554) UNIT,TVOVFU(UNIT),UNIT,TVOVF1(UNIT)
```

```
      WRITE(IOUT,554) UNIT,IVOVFO(UNIT),UNIT,IVOVFA(UNIT)
554   FORMAT(' ',4X,I2,7X,'0',1X,I5,/' ',4X,I2,7X,'1',I6)
C
      IF (IGFLG) CALL IGRES(1,2,3,4)
C
      RETURN
      END
```

Experiment #2

The UNIBUS activity on Machine #2 is monitored. Actually, the same experiment is performed twice, with four different UNIBUS control signals monitored each time. The selection of signals is made using the programmable switch matrices.

The UNIBUS signals monitored, and the observations that can be obtained are:

1) MSYN: tells how fast the UNIBUS is operating. The result can be compared with the claimed maximum of 1.6x10**6 word transfers per second.

2) BBSY: indicates that the UNIBUS is busy.

3) NPR: indicates the amount of cycle stealing performed by the disc.

4) SACK: provides a lower bound on the time to wait before receiving control of the UNIBUS.

5) DATI,DATIP,DATO,DATOB: indicate the type of transfers between the UNIBUS master and its slave. Typically, the processor is bus master, and is fetching instructions from memory, the slave. Data transfers in and out are with respect to the master.

MEASUREMENT OF MACRO COMPILER ON MACHINE #2

*** EXPERIMENT #2 RESULTS ***
METHOD # 1

TIME UNITS = MICROSECONDS

TOTAL EXPERIMENT TIME = 0.3133287870D 08

SIGNAL:	MSYN	BBSY	NPR	SACK
TOTAL TIME ASSERTED:	0.9994D 07	0.3129D 08	0.8160D 05	0.2699D 05
# OF TIMES ASSERTED:	0.1849D 08	0.1164D 06	0.5579D 05	0.5818D 05
AVERAGE TIME ASSERTED:	0.5405D 00	0.2689D 03	0.1463D 01	0.4640D 00
% TIME ASSERTED:	0.3190D 02	0.9987D 02	0.2604D 00	0.8615D-01
# OF TIMES ASSERTED PER MICRO-SECOND:	0.5901D 00	0.3714D-02	0.1780D-02	0.1857D-02

MEASUREMENT OF MACRO COMPILER ON MACHINE #2

*** EXPERIMENT #2 RESULTS ***
METHOD # 2

TIME UNITS = MICROSECONDS

TOTAL EXPERIMENT TIME = 0.3079345110D 08

SIGNAL:	DATI	DATIP	DATO	DATOB
TOTAL TIME ASSERTED:	0.2652D 08	0.2752D 07	0.1372D 07	0.2088D 06
# OF TIMES ASSERTED:	0.1920D 07	0.1870D 07	0.1041D 07	0.2274D 06
AVERAGE TIME ASSERTED:	0.1381D 02	0.1472D 01	0.1318D 01	0.9181D 00
% TIME ASSERTED:	0.8612D 02	0.8939D 01	0.4456D 01	0.6779D 00
# OF TIMES ASSERTED PER MICRO-SECOND:	0.6234D-01	0.6074D-01	0.3382D-01	0.7384D-02

```
C
C                FILE NAME: DEXP2.EXP
C
C PURPOSE OF EXPERIMENT #2:
C
C       -TO MONITOR UNIBUS ACTIVITY ON A PDP11/20.
C
C THE EXPERIMENT WILL BE RUN TWICE.
C
C FOR METHOD=1, THE SIGNALS TO BE MONITORED ARE:
C       MSYN, BBSY, NPR, SACK
C
C FOR METHOD=2, THE SIGNALS TO BE MONITORED ARE:
C       DATI, DATIP, DATO, DATOB
C
C THE DATA TRANSFER SIGNALS ARE OBTAINED BY DECODING THE
C CONTROL SIGNALS C0,C1.
C
C
C
        SUBROUTINE DEXP2
C
        EXTERNAL TVOVFL
        IMPLICIT INTEGER(A-Z)
        INTEGER TVOVF0(5),TVOVF1(5)
        COMMON /CONFIG/NODE,UNIT
        COMMON /TVCOM/TVOVF0(5),TVOVF1(5)
        COMMON /PHMCOM/METHOD
C
C
C SET UP TIMER & EVENT COUNTERS, AND ALSO
C SET UP INTERRUPT GENERATOR FOR POSSIBLE OVERFLOW.
C
        DO 20 UNIT=1,4
        CALL TVSET(1,0,1,3,1,1)
        CALL IGSET(TVOVFL,UNIT)
        TVOVF0(UNIT)=0
20      TVOVF1(UNIT)=0
        UNIT=5
        CALL TVSET(1,0,1,3,1,1)
        TVOVF0(5)=0
        TVOVF1(5)=0
C
30      WRITE(6,31)
31      FORMAT(' METHOD?',/' ')
        READ(6,32) METHOD
32      FORMAT(I1)
        IF (METHOD.GT.2 .OR. METHOD.LT.1) GO TO 30
C
C
C SWITCH MATRIX LINES.
C
        MSYN=2
        BBSY=5
        NPR=2
        SACK=3
        DATI=6
        DATIP=3
        DATO=4
        DATOB=5
        RTIME=7
C
```

```
C TIMER & EVENT COUNTERS - BUFFER 0 AND BUFFER 1.
      TV1B0=0
      TV1B1=1
      TV2B0=2
      TV2B1=3
      TV3B0=0
      TV3B1=1
      TV4B0=2
      TV4B1=3
      TV5B0=6
C
C SET UP SWITCH MATRICES AND LOGIC UNITS FOR
C APPROPRIATE METHOD.
C
      UNIT=1
      CALL SW8DIS
      CALL SW8CON(RTIME,TV5B0)
      UNIT=2
      CALL SW8DIS
C
      IF (METHOD.EQ.2) GO TO 200
C
C LOGIC UNIT #1 = C = BBSY = 1SW8I5
C
      L1:=C
C
      UNIT=1
      CALL SW8CON(MSYN,TV1B0,MSYN,TV1B1,BBSY,TV2B0,BBSY,TV2B1)
      UNIT=2
      CALL SW8CON(NPR,TV3B0,NPR,TV3B1,SACK,TV4B0,SACK,TV4B1)
      GO TO 1000
C
C
200   CONTINUE
C
C LOGIC UNIT #2 = D = DATI = 2SW8I6
C
      I2:=-D
C
      UNIT=1
      CALL SW8CON(DATI,TV1B0,DATI,TV1B1,DATIP,TV2B0,DATIP,TV2B1)
      UNIT=2
      CALL SW8CON(DATO,TV3B0,DATO,TV3B1,DATOB,TV4B0,DATOB,TV4B1)
C
C
C
C BLOCK OFF INTERRUPTS GENERATED BY QUADRA-COMPATRATOR FROM
C DEXP3.EXP
C
      UNIT=1
      CALL QCSET(0)
      UNIT=2
      CALL QCSET(0)
C
1000  RETURN
      END
```

```
C
C                FILE NAME: DEND2.EXP
C
      SUBROUTINE DEND2
C
      IMPLICIT INTEGER(A-Z)
      INTEGER TVU(2),TV1(2),TVOVFU(5),TVOVF1(5)
      LOGICAL IGFLG
      REAL*8 DPTVU(5),DPTV1(5),MAX,PTSA(4),AVGTSA(4),
     1 SIGSUM,SIGAVG,NOAS,SIGNAL(8),NOASPM(4)
      COMMON /CONFIG/NODE,UNIT
      COMMON /TVCOM/TVOVFU(5),TVOVF1(5)
      COMMON /PHMSYS/OUTUNT,IGFLG
      COMMON /PHMCOM/METHOD
      DATA MAX/4294967296.D0/
      DATA SIGNAL/'MSYN','EBSY','NPR','SACK','DATI','DATIP',
     1 'DATO','DATOB'/
C
      WRITE(OUTUNT,300) METHOD
300   FORMAT(' *** EXPERIMENT #2 RESULTS ***',/' ',10X,'METHOD # ',
     1 I1,//' TIME UNITS = MICROSECONDS')
C
C READ TIMER & EVENT COUNTERS, ADJUST FOR POSSIBLE OVERFLOW,
C AND CONVERT TIMES TO MICROSECONDS.
C
      DO 304 UNIT=1,5
      CALL TVREAD(TVU,TV1)
      CALL DI2DF(TVU,DPTVU(UNIT),TV1,DPTV1(UNIT))
      DPTVU(UNIT)=DPTVU(UNIT) + TVOVFU(UNIT)*MAX
      DPTV1(UNIT)=DPTV1(UNIT) + TVOVF1(UNIT)*MAX
304   DPTVU(UNIT)=DPTVU(UNIT) / 10.D0
C
      WRITE(OUTUNT,306) DPTVU(5)
306   FORMAT(' ',/' TOTAL EXPERIMENT TIME = ',D17.10)
      IF (METHOD.EQ.2) GO TO 320
C
C METHOD #1.
C
      WRITE(OUTUNT,311)
311   FORMAT(' ',/' ','SIGNAL:',6X,'MSYN',9X,'EBSY',10X,'NPR',9X,'SACK')
      GO TO 322
C
C METHOD #2.
C
320   WRITE(OUTUNT,321)
321   FORMAT(' ',/' ','SIGNAL:',6X,'DATI',9X,'DATIP',9X,'DATO',8X,
     1 'DATOB')
C
C COMPUTE AVERAGE TIME SIGNAL ASSERTED (AVGTSA),
C PERCENT TIME SIGNAL ASSERTED (PTSA), AND
C NUMBER OF ASSERTIONS PER MICROSECOND (NOASPM).
C
322   DO 328 UNIT=1,4
      AVGTSA(UNIT)=0.D0
      IF (DPTV1(UNIT) .NE. 0.D0)
     1 AVGTSA(UNIT)=DPTVU(UNIT) / DPTV1(UNIT)
      NOASPM(UNIT)=DPTV1(UNIT) / DPTVU(5)
328   PTSA(UNIT)=DPTVU(UNIT) / DPTVU(5) * 100.D0
      WRITE(OUTUNT,329) DPTVU(1),DPTVU(2),DPTVU(3),DPTVU(4),
     1 DPTV1(1),DPTV1(2),DPTV1(3),DPTV1(4),AVGTSA,PTSA,NOASPM
329   FORMAT(' ',/' TOTAL TIME',/' ASSERTED: ',3(D11.4,2X),D11.4,
     1 //' # OF TIMES',/' ASSERTED: ',3(D11.4,2X),D11.4,
     1 //' AVERAGE TIME',/' ASSERTED: ',3(D11.4,2X),D11.4,
     1 //' % TIME',/' ASSERTED: ',3(D11.4,2X),D11.4,
     1 //' # OF TIMES',/' ASSERTED PER',/' MICRO-',/' SECOND: ',
```

```
      1 3(D11.4,2X),D11.4)
C
C
C
C RESET INTERRUPTS IF NECESSARY.
C
      IF (IGFLG) CALL IGRES(1,2,3,4)
      WRITE(OUTUNT,1010)
1010  FORMAT(' ',/' OVERFLOW COUNTS FOR TIMER & EVENT COUNTERS',
      1 /' ',3X,'UNIT',3X,'BUFFER',2X,'COUNT')
      DO 1012 UNIT=1,5
1012  WRITE(OUTUNT,1014) UNIT,TVOVF0(UNIT),UNIT,TVOVF1(UNIT)
1014  FORMAT(' ',I6,7X,'0',I6,/' ',I6,7X,'1',I6)
      RETURN
C
      END
```

Experiment #3

A simple communication mechanism is set up between the two machines, and monitoring of the communication is done at one end. One line messages are typed in at the keyboard of a machine, and sent across the communication link to the other machine. If no message is being typed in at the destination, the message received is printed at the keyboard. Otherwise, the message is put in a queue. Some of the measurements done are:

1) # of characters sent/received.

2) # of messages sent/received.

3) interarrival time of messages.

4) time taken to send and receive messages.

These are some of the typical attributes of a computer network which the monitor might measure. Note that some of these measurements require the monitor to generate an interrupt each time a certain instruction is executed in the object system. For example, the instruction might be the first one in the sequence that puts a message onto a queue. The use of the interrupt generator in this manner is a 'brute force' technique, which will be replaced when the time stamp and character detector are installed.

MEASUREMENT OF A SIMPLE COMMUNICATION SYSTEM.

TIME CONSTRAINTS IMPLIED THAT THERE WAS NOT ENOUGH TIME AVAILABLE TO GET EXPERIMENT #3 RUNNING IN TIME FOR THE DEMONSTRATION. THE DATA BELOW WAS GENERATED TO ILLUSTRATE THE TYPE OF RESULTS THAT WILL BE OBTAINABLE.
THUS, THE NUMBERS THEMSELVES ARE MEANINGLESS IN THIS COPY.
ONLY A SINGLE MACHINE IN THE TWO-COMPUTER COMMUNICATION SYSTEM IS MONITORED.

*** EXPERIMENT #3 RESULTS ***

TIME UNITS = MICROSECONDS

TOTAL EXPERIMENT TIME: 0.5532988930D 08
IDLE TIME: 0.5527981500D 08

STATISTICS ON MESSAGES RECEIVED

MESSAGE NUMBER	TIME OF FIRST CHARACTER	TIME OF LAST CHARACTER	TRANSMISSION TIME	TIME IN QUEUE	# OF CHARACTERS
1	0.7563D 07	0.1130D 08	0.3736D 07	0.3855D 07	0.0000D 00
2	0.1805D 08	0.2326D 08	0.5206D 07	0.4212D 07	0.0000D 00

	MINIMUM	MAXIMUM	AVERAGE
CHARACTERS PER MESSAGE:	0.0000D 00	0.0000D 00	0.0000D 00
TRANSMISSION TIME:	0.3736D 07	0.5206D 07	0.4471D 07
TIME IN QUEUE:	0.1515D 08	0.2747D 08	0.2131D 08
TIME BETWEEN MESSAGES:	0.1730D 09	0.1730D 09	0.1730D 09

TOTAL # OF MESSAGES: 2
TOTAL # OF CHARACTERS: 0

STATISTICS ON MESSAGES SENT

MESSAGE NUMBER	TIME OF FIRST CHARACTER	TIME OF LAST CHARACTER	TRANSMISSION TIME	TIME IN QUEUE	# OF CHARACTERS
1	0.3085D 07	0.3395D 07	0.3099D 06	0.0000D 00	0.0000D 00
2	0.3741D 07	0.4070D 07	0.3298D 06	0.0000D 00	0.0000D 00
3	0.4364D 08	0.4666D 08	0.3020D 07	0.0000D 00	0.0000D 00
4	0.4984D 08	0.0000D 00	-0.4984D 08	0.0000D 00	0.0000D 00

	MINIMUM	MAXIMUM	AVERAGE
CHARACTERS PER MESSAGE:	0.0000D 00	0.0000D 00	0.0000D 00
TRANSMISSION TIME:	-0.4984D 08	0.3020D 07	-0.1155D 08
TIME IN QUEUE:	0.0000D 00	0.0000D 00	0.0000D 00
TIME BETWEEN MESSAGES:	0.3432D 08	0.4548D 09	0.3073D 09

TOTAL # OF MESSAGES: 4
TOTAL # OF CHARACTERS: 0

```
C                FILE NAME: DEXP3.EXP
C
        SUBROUTINE DEXP3
C
C PURPOSE OF EXPERIMENT #3:
C
C       -TO MONITOR A SIMPLE COMMUNICATION SYSTEM INVOLVING
C        TWO PDP11/20'S AND A COMMUNICATION LINK BETWEEN THEM.
C        ONLY ONE ENT (MACHINE #1) OF THE SYSTEM IS MONITORED.
C        ONE LINE MESSAGES ARE SENT BETWEEN THE TWO MACHINES, AND
C        ARE QUEUED AT THE DESTINATION UNTIL THEY CAN BE PRINTED.
C
C       -ATTRIBUTES MEASURED INCLUDE:
C               # OF CHARACTERS RECEIVED
C               # OF CHARACTERS SENT
C               # OF MESSAGES RECEIVED
C               # OF MESSAGES SENT
C               # OF CHARACTERS PER MESSAGE
C               INTERARRIVAL TIME OF MESSAGES
C               INTERDEPARTURE TIME OF MESSAGES
C               TIME MESSAGE SPENDS IN QUEUE
C               AMOUNT OF TIME SYSTEM IS IDLE
C
C TIMER AND EVENT USAGE IS:
C       TV1B0 - # OF CHARACTERS IN A MSG RECEIVED
C       TV1B1 -          "       "       SENT
C       TV5B0 - REAL TIME
C       TV5B1 - AMOUNT OF TIME SYSTEM IS IDLE
C
C THE RANGES OF QUADRA-COMPARATOR #1 ARE SET TO:
C       0 - 1'ST CHARACTER OF MESSAGE ARRIVES OR
C           LAST        "       "       "
C       1 - 1'ST CHARACTER OF MESSAGE IS SENT OR
C           LAST        "       "       "
C       2 - MESSAGE TAKEN OFF QUEUE
C       3 - IDLE LOOP
C
C WHEN ANY OF THE FIRST THREE ADDRESSES ARE REFERENCED, AN
C INTERRUPT IS GENERATED. SOFTWARE IS THEN USED TO READ AND
C SAVE THE APPROPRIATE DATA. FOR RANGES 0 AND 1, THE FIRST
C ADIRESS IS REPLACED BY THE SECOND ADDRESS. BY DYNAMICALLY
C CHANGING THE QUADRA-COMPARATOR RANGES IN THIS WAY, IT IS
C POSSIBLE TO MEASURE TIME INTERVALS.
C
C
        EXTERNAL MSGOUT,MSGIN,MSGOFQ,TVOVFL
        IMPLICIT INTEGER(A-Z)
        INTEGER TVOVF0(5),TVOVF1(5)
        LOGICAL MIFLAG,MOFLAG
        REAL*8 MISTRT(20,2),MIEND(20,2),MOSTRT(20,2),MOEND(20,2),
       1 MOFQ(20,2),CIN(20),COUT(20)
C
        COMMON /CONFIG/NODE,UNIT
        COMMON /TVCOM/TVOVF0(5),TVOVF1(5)
        COMMON /DEMCOM/MISTRT(20,2),MIEND(20,2),MINUM,MOSTRT(20,2),
       1 MOEND(20,2),MONUM,MOFQ(20,2),MQNUM,MIFLAG,MOFLAG,
       1 CIN(20),COUT(20)
        DATA MASK/0177777/,IDLELO/007U052/,IDLEHI/0070114/,
       1 SOMILO/0070530/,SOMIHI/0070534/,
       1 SOMOLO/0070316/,SOMOHI/0070322/
C
        NODE=1
        UNIT=1
C
C SET UP INTERRUPT GENERATORS
```

```
C SET UP INTERRUPT OPERATIONS
C
      CALL IGSET(MSGIN,1)
      CALL IGSET(MSGOUT,2)
      CALL IGSET(MSGOFQ,3)
C PREPARE FOR POSSIBLE OVERFLOW
      UNIT=4
      CALL IGSET(TVOVFL,4)
      TVOVF0(5)=0
      TVOVF1(5)=0
C
C SWITCH MATRIX LINES
C
      CHRIN=6
      CHROUT=5
      RTIME=7
      CTIME=4
      TV1B0=0
      TV1B1=1
      TV5B0=6
      TV5B1=7
C
C SET UP SWITCH MATRIX
C
      UNIT=1
      CALL SW8DIS
      CALL SW8CON(CHRIN,TV1B0,CHROUT,TV1B1,RTIME,TV5B0,
     1 CTIME,TV5B1)
C
C SET UP LOGIC UNITS TO GET SIGNAL FOR CHARACTERS ON
C COMMUNICATION LINK (BOTH DIRECTIONS)
C
C LOGIC UNIT #1 = D = REQUEST B(MACHINE #1) = CHROUT
C
      I1:=D
C
C LOGIC UNIT #2 = E = REQUEST B(MACHINE #2) = CHFIN
C
      I2:=E
C
C SET UP TIMER & EVENT COUNTERS
C
      UNIT=1
      CALL TVSET(0,0,0,3,1,1)
      UNIT=5
      CALL TVSET(1,1,1,3,1,1)
C
C SET UP QUADRA-COMPARATOR RANGES FOR MACHINE #1.
C
      UNIT=1
      CALL QCSET(MASK,0,SOMILO,SOMIHI,1,SOMOLO,SOMOHI,
     1 2,MOFQLO,MOFQHI,3,IDLELO,IDLEHI)
C
C INITIALIZE COUNTERS AND FLAGS
C
      MINUM=0
      MONUM=0
      MQNUM=0
      MIFLAG=.TRUE.
      MOFLAG=.TRUE.
C
      RETURN
      END
```

```
C
C                FILE NAME: DEND3.EXP
C
      SUBROUTINE DEND3
C
      IMPLICIT INTEGER(A-Z)
C
      INTEGER TVOVFU(5),TVOVFL(5),SAVEU(2),SAVEL(2)
      REAL*8 MISTRT(20,2),MIEND(20,2),MOSTRT(20,2),MOEND(20,2),
     1 MOFQ(20,2),CIN(20),COUT(20),ITIME,RTIME
      REAL*8 TTIME,TIMINQ(3),CHPMSG(3),TRTIM(3),TIMBM(3),
     1 TBMSUM,TIQSUM,TRTSUM,MAX,LENGTH,TBM,CPMSUM,TIMONQ
      LOGICAL MIFLAG,MOFLAG,FLAG,IGFLG
      COMMON /DEMCOM/MISTRT(20,2),MIEND(20,2),MINUM,MOSTRT(20,2),
     1 MOEND(20,2),MONUM,MOFQ(20,2),MQNUM,MIFLAG,MOFLAG,
     1 CIN(20),COUT(20)
C
      COMMON /CONFIG/NODE,UNIT
      COMMON /PHMSYS/OUTUNT,IGFLG
      COMMON /TVCOM/TVOVFU(5),TVOVFL(5)
C
      DATA MAX/4294967296.D0/,LENGTH/600.D6/,FLAG/.FALSE./
C
      WRITE(OUTUNT,300)
300   FORMAT(' *** EXPERIMENT #3 RESULTS ***',/' ',
     1/ ' TIME UNITS = MICROSECONDS')
C
C CONVERT TIMES INTO MICROSECONDS, TAKING INTO ACCOUNT
C POSSIBLE OVERFLOW.
C
      UNIT=5
      CALL IVREAD(SAVEU,SAVEL)
      CALL DI2DF(SAVEU,RTIME,SAVEL,ITIME)
      ITIME= (ITIME + TVOVFL(5)*MAX) / 10.D0
      RTIME= (RTIME + TVOVFU(5)*MAX) / 10.D0
      WRITE(OUTUNT,306) RTIME,ITIME
306   FORMAT(' ',/' TOTAL EXPERIMENT TIME: ',D17.10,
     1 /' IDLE TIME: ',D17.10)
C
      IF (MQNUM.EQ.0) GO TO 400
      DO 310 I=1,MQNUM
310   MOFQ(I,1)= ( MOFQ(I,1) + MOFQ(I,2)*MAX) / 10.D0
C
C
C INITIALIZE VARIABLES
C
400   CPMSUM=0
      TRTSUM=0.D0
      TIQSUM=0.D0
      TBMSUM=0.D0
      CPMSUM=0
C
C INITIALIZE MIN(1) AND MAX(2)
C
      CHPMSG(1)=100
      CHPMSG(2)=0
      TRTIM(1)=LENGTH
      TRTIM(2)=0.D0
      TIMINQ(1)=LENGTH
      TIMINQ(2)=0.D0
      TIMBM(1)=LENGTH
      TIMBM(2)=0.D0
      T3M=0.D0
C
      IF (.NOT.FLAG) WRITE(OUTUNT,401)
```

```
      IF (...) WRITE(OUTUNT,401)
      IF (FLAG) WRITE(OUTUNT,402)
401   FORMAT(' ',///' ',8X,'STATISTICS ON MESSAGES RECEIVED'
     1 ,/' ',8X,31('-'),/' ')
402   FORMAT(' ',///' ',8X,'STATISTICS ON MESSAGES SENT'
     1 ,/' ',8X,27('-'),/' ')
      IF (MINUM.NE.0) GO TO 420
      WRITE(OUTUNT,416)
416   FORMAT(' ',/' # OF MESSAGES = 0')
      GO TO 700
C
C
420   WRITE(OUTUNT,421)
421   FORMAT(' ','MESSAGE',1X,'TIME OF',6X,'TIME OF',6X,'TRANSMISSION',
     1 1X,'TIME IN',6X,'# OF',/' ','NUMBER',2X,'FIRST',8X,'LAST',9X,
     1 'TIME',9X,'QUEUE',8X,'CHARACTERS',/' ',8X,'CHARACTER',4X,
     1 'CHARACTER')
C
C
      DO 500 I=1,MINUM
C
C CONVERT TIMES TO MICROSECONDS
C
      MISTRT(I,1) = ( MISTRT(I,1) + MISTRT(I,2)*MAX) / 10.D0
      MIEND(I,1) = ( MIEND(I,1) + MIEND(I,2)*MAX) / 10.D0
      MOSTRT(I,1) = ( MOSTRT(I,1) + MOSTRT(I,2)*MAX) / 10.D0
      MOEND(I,1) = ( MOEND(I,1) + MOEND(I,2)*MAX) / 10.D0
C
      TTIME=MIEND(I,1)-MISTRT(I,1)
C
      TIMONQ=MOFQ(I,1) - MIEND(I,1)
      IF (FLAG) TIMONQ=0.D0
C
      WRITE(OUTUNT,425) I,MISTRT(I,1),MIEND(I,1),TTIME,TIMONQ,CIN(I)
425   FORMAT(' ',I4,3X,4(D11.4,2X),D11.4)
C
C CALCULATE MINIMUMS
C
      IF (CIN(I).LT.CHPMSG(1)) CHPMSG(1)=CIN(I)
      IF (TTIME.LT.TRTIM(1)) TRTIM(1)=TTIME
      IF (MOFQ(I,1).LT.TIMINQ(1)) TIMINQ(1)=MOFQ(I,1)
      TBM=0.D0
      IF (MINUM.EQ.1 .OR. I.EQ.MINUM) GO TO 430
      TBM=MISTRT(I+1,1)-MISTRT(I,1)
      IF (TBM.LT.TIMBM(1)) TIMBM(1)=TBM
C
C CALCULATE MAXIMUMS
C
430   IF (CIN(I).GT.CHPMSG(2)) CHPMSG(2)=CIN(I)
      IF (TTIME.GT.TRTIM(2)) TRTIM(2)=TTIME
      IF (MOFQ(I,1).GT.TIMINQ(2)) TIMINQ(2)=MOFQ(I,1)
      IF (TBM.GT.TIMBM(2)) TIMBM(2)=TBM
C
C CALCULATE SUMS, FOR USE IN AVERAGES LATER
C
440   CPMSUM=CPMSUM+CIN(I)
      TRTSUM=TRTSUM+TTIME
      TIQSUM=TIQSUM+TIMONQ
      TBMSUM=TBMSUM+TBM
C
C
C
500   CONTINUE
C
C CALCULATE AVERAGES
C
      CHPMSG(3)=CPMSUM / MINUM
```

```
      CHPMSG(3)=CPMSUM / MINUM
      TRTIM(3)=TRTSUM / MINUM
      TIMINQ(3)=TIQSUM / MINUM
      IF (.NOT.FLAG) GO TO 580
      TIMINQ(1)=0.D0
      TIMINQ(2)=0.D0
      TIMINQ(3)=0.D0
580   IF (MINUM.GT.1) GO TO 590
      TIMBM(1)=0.D0
      TIMBM(2)=0.D0
      TIMBM(3)=0.D0
      GO TO 600
590   TIMBM(3)=TBMSUM / (MINUM-1)
C
C PRINT OUT MAX,MIN, AND AVG
C
600   WRITE(OUTUNT,601)
601   FORMAT(///' ',16X,'MINIMUM',6X,'MAXIMUM',6X,'AVERAGE',
     1 /' ',16X,'-------',6X,'-------',6X,'-------')
C
      WRITE(OUTUNT,602) CHPMSG,TRTIM,TIMINQ,TIMBM
602   FORMAT(' CHARACTERS',/' PER MESSAGE:',2X,2(D11.4,2X),D11.4,
     1 //' TRANSMISSION',/' TIME:',9X,2(D11.4,2X),D11.4,
     1 //' TIME IN',/' QUEUE:',8X,2(D11.4,2X),D11.4,//
     1 ' TIME BETWEEN',/' MESSAGES:',5X,2(D11.4,2X),D11.4)
C
      WRITE(OUTUNT,603) MINUM,CPMSUM
603   FORMAT(' ',/' TOTAL # OF MESSAGES:  ',2X,I5,
     1 /' TOTAL # OF CHARACTERS:  ',D11.4)
C
C
700   IF (FLAG) GO TO 1000
C
C COPY 'MSG OUT' DATA TO 'MSG IN' ARRAYS FOR PROCESSING
C
      DO 750 I=1,MONUM
      MISTRT(I,1)=MOSTRT(I,1)
      MISTRT(I,2)=MOSTRT(I,2)
      MIEND(I,1)=MOEND(I,1)
      MIEND(I,2)=MOEND(I,2)
      MOFQ(I,1)=0.D0
      MOFQ(I,2)=0.D0
750   CIN(I)=COUT(I)
      MINUM=MONUM
      FLAG=.TRUE.
      GO TO 400
C
1000  IF (IGFLG) CALL IGRES(1,2,3,4)
C
      RETURN
      END
```

```
C
C                FILE NAME: MSGIN.EXP
C
      SUBROUTINE MSGIN
C
C MSGIN IS ENTERED WHEN A MESSAGE IS JUST STARTING TO BE
C RECEIVED, OR AFTER THE ENTIRE MESSAGE HAS BEEN RECEIVED.
C
      IMPLICIT INTEGER(A-Z)
      INTEGER SAVE(2),IVOVFU(5),TVOVFI(5)
      LOGICAL MIFLAG,MOFLAG
      REAL*8 MISTRT(20,2),MIEND(20,2),MOSTRT(20,2),MOEND(20,2),MOFQ(20,2),
     1 CIN(20),COUT(20)
C
      COMMON /DEMCOM/ MISTRT(20,2),MIEND(20,2),MINUM,
     1 MOSTRT(20,2),MOEND(20,2),MONUM,MOFQ(20,2),MQNUM,MIFLAG,MOFLAG,
     1 CIN(20),COUT(20)
C
      COMMON /CONFIG/NODE,UNIT
      COMMON /IVCOM/IVOVFU(5),IVOVFI(5)
C
      DATA SOMILO/00705300/, SOMIHI/00705340/, EOMILO/00705100/,
     1 EOMIHI/00705140/
C
C READ CURRENT TIME
C
      PAUSE 0
      UNIT=5
      CALL IVREDU(SAVE)
C
C MIFLAG=TRUE => START OF MESSAGE
C
      IF (.NOT.MIFLAG) GO TO 50
C
C START OF MESSAGE RECEIVED.
C
      MINUM=MINUM+1
      CALL DI2DF(SAVE,MISTRT(MINUM,1))
      MISTRT(MINUM,2)=IVOVFU(5)
C
C RESET QUADRA-COMPARATOR RANGE FOR
C 'END OF MSG IN' ADIRESS
C
      UNIT=1
      CALL QCSET(0,EOMILO,EOMIHI)
      MIFLAG=.FALSE.
C
      GO TO 100
C
C END OF MESSAGE RECEIVED
C
50    CALL DI2DF(SAVE,MIEND(MINUM,1))
      MIEND(MINUM,2)=IVOVFU(5)
C
C RECORD # OF CHARACTER RECEIVED
C
      UNIT=1
      CALL IVREDU(SAVE)
      CALL DI2DF(SAVE,CIN(MINUM))
C
C CLEAR BUFFER
C
      CALL IVSET(0,0,1,3,1,0)
C
C RESET QUADRA-COMPARATOR RANGE TO
```

```
C 'START OF MSG IN' ADDRESS
C
      CALL QCSET(0,SOMILO,SOMIHI)
C
      MIFLAG=.TRUE.
C
100   RETURN
      END
```

```
C
C                FILE NAME: MSGOUT.EXP
C
        SUBROUTINE MSGOUT
C
C MSGOUT IS ENTERED WHEN A MESSAGE IS JUST STARTING TO
C BE SENT FROM THE KEYBOARD, OR AFTER THE ENTIRE
C MESSAGE HAS BEEN SENT TO THE DESTINATION
C
        IMPLICIT INTEGER(A-Z)
        INTEGER SAVE(2),TVOVFU(5),TVOVFL(5)
        LOGICAL MIFLAG,MOFLAG
        REAL*8 MISTRT(20,2),MIEND(20,2),MOSTRT(20,2),MOEND(20,2),MOFQ(20,2),
        1 CIN(20),COUT(20)
C
        COMMON /DEMCOM/ MISTRT(20,2),MIEND(20,2),MINUM,MOSTRT(20,2),
        1 MOEND(20,2),MONUM,MOFQ(20,2),MQNUM,MIFLAG,MOFLAG,
        1 CIN(20),COUT(20)
C
        COMMON /CONFIG/NODE,UNIT
        COMMON /TVCOM/TVOVFU(5),TVOVFL(5)
C
        DATA SOMOLO/0070316/,SOMOHI/0070322/,EOMOLO/0070434/,
        1 EOMOHI/0070440/
C
C READ CURRENT TIME
C
        PAUSE 1
        UNIT=5
        CALL TVREDU(SAVE)
C
C MOFLAG=TRUE => START OF MESSAGE OUT
C
        IF (.NOT.MOFLAG) GO TO 50
C
C START OF MESSAGE OUT
C
        MONUM=MONUM+1
        CALL DI2DF(SAVE,MOSTRT(MONUM,1))
        MOSTRT(MONUM,2)=TVOVFU(5)
C
C RESET QUADRA-COMPARATOR FOR
C 'END OF MSG OUT' ADDRESS
C
        UNIT=1
        CALL QCSET(1,EOMOLO,EOMOHI)
        MOFLAG=.FALSE.
        GO TO 100
C
C
50      CALL DI2DF(SAVE,MOEND(MONUM,1))
        MOEND(MONUM,2)=TVOVFU(5)
C
C RECORD # OF CHARACTERS SENT OUT
C
        UNIT=1
        CALL TVREDL(SAVE)
        CALL DI2DF(SAVE,COUT(MONUM))
C
C CLEAR BUFFER
C
        CALL TVSET(0,0,1,3,0,1)
C
C RESET QUADRA-COMPARATOR FOR
C 'START OF MSG OUT' ADDRESS
```

```
C
      CALL QCSET(1,SOMOLO,SOMOHI)
C
      MOFLAG=.TRUE.
C
C
100   RETURN
      END
```

```
C
C                FILE NAME: MSGOFQ.EXP
C
      SUBROUTINE MSGOFQ
C
C MSGOFQ IS ENTERED EACH TIME A MESSAGE AT THE DESTINATION
C HAS BEEN PRINTED AND THEN TAKEN OFF OF THE QUEUE
C
      IMPLICIT INTEGER(A-Z)
      INTEGER SAVE(2),IVOVFU(5),IVOVFL(5)
      LOGICAL MIFLAG,MOFLAG
      REAL*8 MISTRT(20,2),MIEND(20,2),MOSTRT(20,2),MOEND(20,2),
     1 MOFQ(20,2),CIN(20),COUT(20)
C
      COMMON /DEMCOM/MISTRT(20,2),MIEND(20,2),MINUM,MOSTRT(20,2),
     1 MOEND(20,2),MONUM,MOFQ(20,2),MQNUM,MIFLAG,MOFLAG,CIN(20),
     1 COUT(20)
C
      COMMON /CONFIG/NODE,UNIT
      COMMON /IVCOM/IVOVFU(5),IVOVFL(5)
C
C READ CURRENT TIME
C
      PAUSE 2
      VOID>6
      CALL IVREDU(SAVE)
      MQNUM=MQNUM+1
      CALL DI2DF(SAVE,MOFQ(MQNUM,1))
      MOFQ(MQNUM,2)=IVOVFU(5)
C
      RETURN
      END
```

Experiment #4

Both machines are monitored simultaneously. A load generator runs on each machine, causing the computer to think that the communication link is a user terminal. Using the monitor, we are able to construct a state vector:

<line busy, CPU#2 busy, CPU#1 busy>

The output is a table of the eight possible states (0-7), and the total time spent in each state. In addition, the duration of the experiment is used in determining the percentage of real time spent in each state. The user must provide the address of the idle loop on each machine. A machine is busy if it is executing outside of the idle loop. The system is defined to be doing useful computing if one or both of the computers are busy. Thus, results are also given for the percentage of compute time spent in each state.

Parameters controlling the rate at which messages and individual characters are transmitted may be varied. By performing further measurements, we hope to determine the correctness of an analytic model of the two-computer system.

Note that the state vector table could be produced for any program that happened to be running in the object system.

STATE VECTOR MEASUREMENTS OF LOAD GENERATOR OPERATING ON
TWO-COMPUTER SYSTEM

*** EXPERIMENT #4 RESULTS ***

THE FOLLOWING TABLE INDICATES THE TIME SPENT
IN EACH OF THE 8 POSSIBLE STATES INVOLVING
2 CPU'S AND A COMMUNICATIONS LINK BETWEEN THEM.

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.738712186000000D 08
TOTAL COMPUTE TIME: 0.400495004000000D 08
STATE VECTOR IS
<LINE BUSY,CPU #2 BUSY,CPU #1 BUSY>
CPU #1 = MATH PDP11/20
CPU #2 = ENGINEERING PDP11/20

STATE	STATE VECTOR	TIME IN STATE	TIME/ REAL TIME	TIME/ COMPUTE TIME
0	000	0.336518667000 08	0.4555D 02 %	0.8403D 02 %
1	001	0.104672410000 08	0.1417D 02 %	0.2614D 02 %
2	010	0.211263681000 08	0.2860D 02 %	0.5275D 02 %
3	011	0.848773930000 07	0.1149D 02 %	0.2119D 02 %
4	100	0.000000000000 00	0.0000D 00 %	0.0000D 00 %
5	101	0.000000000000 00	0.0000D 00 %	0.0000D 00 %
6	110	0.185800000001 03	0.2515D-03 %	0.4639D-03 %
7	111	0.172000000000 03	0.2328D-03 %	0.4295D-03 %

	TOTAL TIME	TIME/ REAL TIME	TIME/ COMPUTE TIME
CPU #1 BUSY:	0.189551523000 08	0.2566D 02 %	0.4733D 02 %
CPU #2 BUSY:	0.296144652000 08	0.4009D 02 %	0.7394D 02 %
LINE BUSY:	0.357800000000 03	0.4844D-03 %	0.8934D-03 %

THE FOLLOWING MEASUREMENTS ILLUSTRATE THE TWO-COMPUTER SYSTEM WITH BOTH MACHINES IN THE IDLE STATE AND NO TRANSMISSION ON THE COMMUNICATIONS LINK.

*** EXPERIMENT #4 RESULTS ***

THE FOLLOWING TABLE INDICATES THE TIME SPENT IN EACH OF THE 8 POSSIBLE STATES INVOLVING 2 CPU'S AND A COMMUNICATIONS LINK BETWEEN THEM.

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.2025315000000000D 07
TOTAL COMPUTE TIME: 0.7279300000000000D 04
STATE VECTOR IS
<LINE BUSY,CPU #2 BUSY,CPU #1 BUSY>
CPU #1 = MATH PDP11/20
CPU #2 = ENGINEERING PDP11/20

STATE	STATE VECTOR	TIME IN STATE	TIME/ REAL TIME	TIME/ COMPUTE TIME
0	000	0.2005261100D 07	0.9901D 02 %	0.2755D 05 %
1	001	0.3758300000D 04	0.1861D 00 %	0.5177D 02 %
2	010	0.3511400000D 04	0.1734D 00 %	0.4824D 02 %
3	011	0.3600000000D 01	0.1778D-03 %	0.4946D-01 %
4	100	0.0000000000D 00	0.0000D 00 %	0.0000D 00 %
5	101	0.0000000000D 00	0.0000D 00 %	0.0000D 00 %
6	110	0.0000000000D 00	0.0000D 00 %	0.0000D 00 %
7	111	0.0000000000D 00	0.0000D 00 %	0.0000D 00 %

	TOTAL TIME	TIME/ REAL TIME	TIME/ COMPUTE TIME
CPU #1 BUSY:	0.3771900000D 04	0.1862D 00 %	0.5182D 02 %
CPU #2 BUSY:	0.3515000000D 04	0.1736D 00 %	0.4829D 02 %
LINE BUSY:	0.0000000000D 00	0.0000D 00 %	0.0000D 00 %

MEASUREMENT OF FILE TRANSFER FROM MACHINE #1 TO MACHINE #2

*** EXPERIMENT #4 RESULTS ***

THE FOLLOWING TABLE INDICATES THE TIME SPENT IN EACH OF THE 8 POSSIBLE STATES INVOLVING 2 CPU'S AND A COMMUNICATIONS LINK BETWEEN THEM.

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.97723467000000000D 07
TOTAL COMPUTE TIME: 0.48486980000000000D 07
STATE VECTOR IS
<LINE BUSY,CPU #2 BUSY,CPU #1 BUSY>
CPU #1 = MATH PDP11/20
CPU #2 = ENGINEERING PDP11/20

STATE	STATE VECTOR	TIME IN STATE	TIME/ REAL TIME	TIME/ COMPUTE TIME
0	000	0.45763595000D 07	0.4683D 02 %	0.9438D 02 %
1	001	0.73560010000D 06	0.7527D 01 %	0.1517D 02 %
2	010	0.10521093000D 07	0.1077D 02 %	0.2170D 02 %
3	011	0.23034536000D 07	0.2357D 02 %	0.4751D 02 %
4	100	0.31808860000D 06	0.3255D 01 %	0.6560D 01 %
5	101	0.58159200000D 05	0.5951D 00 %	0.1199D 01 %
6	110	0.31752550000D 06	0.3249D 01 %	0.6549D 01 %
7	111	0.38223860000D 06	0.3911D 01 %	0.7883D 01 %

	TOTAL TIME	TIME/ REAL TIME	TIME/ COMPUTE TIME
CPU #1 BUSY:	0.34794515000D 07	0.3561D 02 %	0.7176D 02 %
CPU #2 BUSY:	0.40553270000D 07	0.4150D 02 %	0.8364D 02 %
LINE BUSY:	0.10760119000D 07	0.1101D 02 %	0.2219D 02 %

MEASUREMENT OF FILE TRANSFER FROM MACHINE #2 TO MACHINE #1

*** EXPERIMENT #4 RESULTS ***

THE FOLLOWING TABLE INDICATES THE TIME SPENT IN EACH OF THE 8 POSSIBLE STATES INVOLVING 2 CPU'S AND A COMMUNICATIONS LINK BETWEEN THEM.

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.90910316000000000D 07
TOTAL COMPUTE TIME: 0.39923385000000000D 07
STATE VECTOR IS
<LINE BUSY,CPU #2 BUSY,CPU #1 BUSY>
CPU #1 = MATH PDP11/20
CPU #2 = ENGINEERING PDP11/20

STATE	STATE VECTOR	TIME IN STATE	TIME/ REAL TIME	TIME/ COMPUTE TIME
0	000	0.48969495000D 07	0.5387D 02 %	0.1227D 03 %
1	001	0.27514470000D 06	0.3027D 01 %	0.6892D 01 %
2	010	0.57698360000D 06	0.6347D 01 %	0.1445D 02 %
3	011	0.20753734000D 07	0.2283D 02 %	0.5198D 02 %
4	100	0.16956910000D 06	0.1865D 01 %	0.4247D 01 %
5	101	0.58643130000D 06	0.6451D 01 %	0.1469D 02 %
6	110	0.75856000000D 05	0.8344D 00 %	0.1900D 01 %
7	111	0.40314820000D 06	0.4435D 01 %	0.1010D 02 %

	TOTAL TIME	TIME/ REAL TIME	TIME/ COMPUTE TIME
CPU #1 BUSY:	0.33400976000D 07	0.3674D 02 %	0.8366D 02 %
CPU #2 BUSY:	0.31313612000D 07	0.3444D 02 %	0.7843D 02 %
LINE BUSY:	0.12350046000D 07	0.1358D 02 %	0.3093D 02 %

```
C
C                FILE NAME: DEXP4.EXP
C
C PURPOSE OF EXPERIMENT #4
C
C
C       -TO DETERMINE THE TIME SPENT IN EACH OF
C        8 POSSIBLE STATES AS A RESULT OF USING THE
C        STATE VECTOR [LINE BUSY,CPU #2 BUSY,CPU #1 BUSY]
C        WHERE THE LINE IS A COMMUNICATIONS LINK BETWEEN
C        THE TWO CPU'S.
C
C       -TO FILL IN THE FOLLOWING TABLE:
C        STATE     TIME IN       TIME/          TIME/
C                    STATE      REAL TIME     COMPUTE TIME
C
C        WHERE REAL TIME=>TOTAL EXPERIMENT TIME, AND
C        COMPUTE TIME => CPU #1 OR CPU #2  BUSY.
C
C THE SYSTEM IS DEFINED TO BE BUSY, OR DOING USEFUL
C COMPUTING, IF ONE OR BOTH OF THE MACHINES IS
C EXECUTING OUTSIDE OF THE PROGRAM WAIT LOOP.
C
C
C
        SUBROUTINE DEXP4
C
        EXTERNAL TVOVFL
        IMPLICIT INTEGER(A-Z)
        INTEGER TVOVFU(5),TVOVFL(5)
        COMMON /CONFIG/NODE,UNIT
        COMMON /TVCOM/TVOVFU(5),TVOVFL(5)
C
C QUADRA-COMPARATORS MUST BE SET UP, USING
C THE TEST PROGRAM. FOR COMPUTE TIME, SET RANGE #3 TO
C THE ADDRESSES OF THE WAIT LOOP.
C
        WRITE(6,12)
12      FORMAT(' USE TEST PROGRAM TO SPECIFY QC RANGE #3 = WAIT LOOP',/
        1 '  ON BOTH MACHINES',/' ')
C
C COMPUTE TIME => CPU#1 BUSY  OR  CPU #2 BUSY
C       => (OUTSIDE CPU #1 WAIT LOOP)  OR
C          (OUTSIDE CPU #2 WAIT LOOP)
C
C
C
C
C SET UP LOGIC UNITS.
C
C LOGIC UNIT #1 = -B = STATE 2 = 1SW8I5
C
        L1:=-B
C
C LOGIC UNIT #2 = -B = STATE 3 = 1SW8I6
C
        L2:=-B
C
C SET UP 8X8 SWITCH MATRICES
C
C STATES
        S0=2
        S1=3
        S2=5
        S3=6
```

```
      S3=3
      S4=4
      S5=5
      S6=6
      S7=7
C
C TIMER & EVENT COUNTERS
C
      TV1B0=0
      TV1B1=1
      TV2B0=2
      TV2B1=3
      TV3B0=0
      TV3B1=1
      TV4B0=2
      TV4B1=3
      TV5B0=6
      TV5B1=7
      RTIME=7
      CTIME=4
C
      UNIT=1
      CALL SW8DIS
      CALL SW8CON(S0,TV1B0,S1,TV1B1,S2,TV2B0,S3,TV2B1,
     1 RTIME,TV5B0,CTIME,TV5B1)
C
      UNIT=2
      CALL SW8DIS
      CALL SW8CON(S4,TV3B0,S5,TV3B1,S6,TV4B0,S7,TV4B1)
C
C SET UP TIMER & EVENT COUNTERS
C
      DO 35 UNIT=1,5
35    CALL TVSET(1,1,1,3,1,1)
C
C
C SET UP INTERRUPT GENERATOR AND CLEAR OVERFLOW COUNTS.
C
      DO 40 UNIT=1,4
      TVOVF0(UNIT)=0
      TVOVF1(UNIT)=0
40    CALL IGSET(TVOVFL,UNIT)
C
C TV #5 OVERFLOW LINES ARE  'OR'ED WITH THOSE OF TV #4, SINCE
C ONLY HAVE FOUR INTERRUPT LINES. A SPECIAL CHECK IS MADE IN
C 'TVOVFL'.
      TVOVF0(5)=0
      TVOVF1(5)=0
C
      RETURN
      END
```

```
C
C                FILE NAME: DEND4.EXP
C
      SUBROUTINE DEND4
C
      IMPLICIT INTEGER (A-Z)
      INTEGER TVOVF0(5),TVOVF1(5),TV0(2),TV1(2),S(8,2)
C
      DOUBLE PRECISION DPTV0(5),DPTV1(5),PRTIME,PCTIME,MAX,
     1 CPU1,CPU2,LINE,CPU1R,CPU1C,CPU2R,CPU2C,LINER,LINEC
      LOGICAL IGFLG
C
      COMMON /CONFIG/NODE,UNIT
      COMMON /TVCOM/TVOVF0(5),TVOVF1(5)
      COMMON /PHMSYS/OUTUNT,IGFLG
C
      DATA S(1,1),S(1,2),S(2,1),S(2,2),S(3,1),S(3,2),
     1 S(4,1),S(4,2),S(5,1),S(5,2),S(6,1),S(6,2),S(7,1),S(7,2),
     1 S(8,1),S(8,2) /'00','0','00','1','01','0','01','1','10','0',
     1 '10','1','11','0','11','1'/
C MAX = 2.D0**32
      DATA  MAX/4294967296.D0/
C
C ZERO TOTALS
      CPU1=0.D0
      CPU2=0.D0
      LINE=0.D0
C
      WRITE(OUTUNT,10)
10    FORMAT(' ','*** EXPERIMENT #4 RESULTS ***',/' ',/' ',
     1 '   THE FOLLOWING TABLE INDICATES THE TIME SPENT ',
     1 /' ','IN EACH OF THE 8 POSSIBLE STATES INVOLVING ',/' ',
     1 '2 CPU''S AND A COMMUNICATIONS LINK BETWEEN THEM.',/' ')
C
C READ AND CONVERT ALL TIMES TO DOUBLE PRECISION
C
      DO 20 UNIT=1,5
      CALL TVREAD(TV0,TV1)
      CALL DI2DF(TV0,DPTV0(UNIT))
      CALL DI2DF(TV1,DPTV1(UNIT))
C ADJUST FOR POSSIBLE OVERFLOW
      DPTV0(UNIT)=DPTV0(UNIT) + TVOVF0(UNIT)*MAX
      DPTV1(UNIT)=DPTV1(UNIT) + TVOVF1(UNIT)*MAX
C CONVERT TO MICRO-SECONDS
      DPTV0(UNIT)=DPTV0(UNIT)/10.D0
20    DPTV1(UNIT)=DPTV1(UNIT)/10.D0
C
C
      WRITE(OUTUNT,22) DPTV0(5),DPTV1(5)
22    FORMAT(' ',/' ','ALL TIMES IN MICROSECONDS',/' ',
     1 'TOTAL REAL TIME: ',D25.16,/' ',
     1 'TOTAL COMPUTE TIME: ',D25.16)
C
      WRITE(OUTUNT,24)
24    FORMAT(' ','STATE VECTOR IS ',/' ',
     1 3X,'<LINE BUSY,CPU #2 BUSY,CPU #1 BUSY>',/' ',
     1 3X,'CPU #1 = MATH PDP11/20',/' ',3X,'CPU #2 = ',
     1 'ENGINEERING PDP11/20',/' ')
C
      WRITE(OUTUNT,32)
32    FORMAT(' ',3X,'STATE',3X,
     1 'STATE',3X,   'TIME',16X,    'TIME/',11X,    'TIME/',/' ',11X,
     1 'VECTOR',2X,'IN STATE',12X,'REAL TIME',7X,'COMPUTE TIME',/' ')
C LOOK AT ALL 4 TIMER & EVENT COUNTERS
C
```

```
C
      STATE=-1
C
      DO 100 UNIT=1,4
      STATE=STATE+1
      ST=STATE+1
C
C % REAL TIME FOR BUFFER 0
      PRTIME=DPTV0(UNIT) / DPTV0(5) * 100.00
C % COMPUTE TIME FOR BUFFER 1
      PCTIME=DPTV0(UNIT) / DPTV1(5) * 100.00
C
      WRITE(OUTUNT,40) STATE,S(ST,1),S(ST,2),DPTV0(UNIT),PRTIME,PCTIME
40    FORMAT(' ',I4,7X,A2,A1,4X,D17.10,3X,D11.4,' %',3X,D11.4,' %')
C COMPUTE TOTALS
      IF ((STATE/2)*2 .NE. STATE) CPU1=CPU1+DPTV0(UNIT)
      IF (STATE.EQ.2 .OR. STATE.EQ.3 .OR. STATE.EQ.6
     1 .OR. STATE.EQ.7) CPU2=CPU2+DPTV0(UNIT)
      IF (STATE.GE.4) LINE=LINE+DPTV0(UNIT)
C
      STATE=STATE+1
      ST=STATE+1
C
C % REAL TIME FOR BUFFER 1
      PRTIME=DPTV1(UNIT) /  DPTV0(5) * 100.00
C % COMPUTE TIME FOR BUFFER 1
      PCTIME=DPTV1(UNIT) / DPTV1(5) * 100.00
      WRITE(OUTUNT,40) STATE,S(ST,1),S(ST,2),DPTV1(UNIT),PRTIME,PCTIME
C COMPUTE TOTALS
      IF ((STATE/2)*2 .NE. STATE) CPU1=CPU1+DPTV1(UNIT)
      IF (STATE.EQ.2 .OR. STATE.EQ.3 .OR. STATE.EQ.6
     1 .OR. STATE.EQ.7) CPU2=CPU2+DPTV1(UNIT)
100   IF (STATE.GE.4) LINE=LINE+DPTV1(UNIT)
C
      CPU1R=CPU1 / DPTV0(5) * 100.00
      CPU2R=CPU2 / DPTV0(5) * 100.00
      LINER=LINE / DPTV0(5) * 100.00
      CPU1C=CPU1 / DPTV1(5) * 100.00
      CPU2C=CPU2 / DPTV1(5) * 100.00
      LINEC=LINE / DPTV1(5) * 100.00
C
C PRINT TOTALS AND %
C
      WRITE(OUTUNT,150)CPU1,CPU1R,CPU1C,CPU2,CPU2R,CPU2C,LINE,
     1 LINER,LINEC
150   FORMAT(' ',//' ',16X,'TOTAL TIME',13X,'TIME/',11X,'TIME/',/' ',
     1 36X,'REAL TIME',7X,'COMPUTE TIME',/' CPU #1 BUSY:',3X,
     1 D17.10,2(3X,D11.4,' %'),/' CPU #2 BUSY:',3X,D17.10,
     1 2(3X,D11.4,' %'),/' LINE BUSY:',5X,D17.10,2(3X,D11.4,' %'))
      WRITE(OUTUNT,200)
200   FORMAT(' ',/' ',/' ','OVERFLOW COUNTS FOR TIMER & EVENT COUNTER',
     1 /' ',3X,'UNIT',3X,'BUFFER',3X,'COUNT')
      DO 250 UNIT=1,5
250   WRITE(OUTUNT,251) UNIT,TVOVF0(UNIT),UNIT,TVOVF1(UNIT)
251   FORMAT(' ',4X,I2,7X,'0',1X,I5,/' ',4X,I2,7X,'1',1X,I5)
C
C
C RESET INTERRUPT GENERATOR IF NECESSARY
C (I.E. FIRST PASS THROUGH THIS CODE)
      IF (IGFLG) CALL IGRES(1,2,3,4)
      RETURN
      END
```

Experiment #5

The object system is the same as in experiment #4. In this experiment, memory activity on both machines is examined. The result is a table for each machine, showing the percentage of time spent in each of the four user-defined memory ranges. By using the last range of the quadra-comparators to specify the program's idle loop, compute time results may also be obtained.

For programs other than the load generator, the ranges could first be set to span all of available memory. Based on the results obtained, refinements could be made until the user could clearly tell where the program was spending most of its time.

MEASUREMENT OF MEMORY ACTIVITY WHILE LOAD GENERATOR IS
OPERATING ON TWO-COMPUTER SYSTEM

*** EXPERIMENT #5 RESULTS ***

THE FOLLOWING TABLE INDICATES HOW LONG EACH OF
2 CPU'S SPENDS EXECUTING IN THE SPECIFIED REGIONS OF
MEMORY, WHILE THEY INTERACT VIA A COMMUNICATIONS LINK.
CPU #1 = MATH-PDP11/20
CPU #2 = ENGINEERING PDP11/20

* RESULTS FROM MACHINE 1 *

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.12317633940000000D 09
TOTAL COMPUTE TIME: 0.58890636401000000D 08

RANGE	TIME IN RANGE	TIME/ REAL TIME	TIME/ COMPUTE TIME
33576 177777	0.1171313244D 09	0.9509D 02 %	0.1989D 03 %
67374 71512	0.9595317890D 08	0.7790D 02 %	0.1629D 03 %
64754 65304	0.4460720400D 07	0.3621D 01 %	0.7575D 01 %
70426 70502	0.9612703930D 08	0.7804D 02 %	0.1632D 03 %

* RESULTS FROM MACHINE 2 *

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.12317633940000000D 09
TOTAL COMPUTE TIME: 0.58890636401000000D 08

RANGE	TIME IN RANGE	TIME/ REAL TIME	TIME/ COMPUTE TIME
53576 177777	0.1131051429D 09	0.9182D 02 %	0.1921D 03 %
107374 111512	0.8259326630D 08	0.6705D 02 %	0.1402D 03 %
104754 105304	0.5157618800D 07	0.4999D 01 %	0.1046D 02 %
110426 110502	0.8259078210D 08	0.6705D 02 %	0.1402D 03 %

THIS IS THE START OF A SEQUENCE OF MEASUREMENTS TO DETERMINE
THE LOCATION OF THE DOS-11 OPERATING SYSTEM IDLE LOOP.

*** EXPERIMENT #5 RESULTS ***

THE FOLLOWING TABLE INDICATES HOW LONG EACH OF
2 CPU'S SPENDS EXECUTING IN THE SPECIFIED REGIONS OF
MEMORY, WHILE THEY INTERACT VIA A COMMUNICATIONS LINK.
CPU #1 = MATH PDP11/20
CPU #2 = ENGINEERING PDP11/20

FOR THESE MEASUREMENTS, ONLY MACHINE #1 IS CONSIDERED.
NO COMMUNICATION IS OCCURRING BETWEEN THE TWO MACHINES, AND THE
RESULTS WHICH PERTAINED TO MACHINE #2 HAVE BEEN DELETED TO
SAVE SPACE.

NO COMPUTE TIME RESULTS DETERMINED.

* RESULTS FROM MACHINE 1 *

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.203772180000000000D 07
TOTAL COMPUTE TIME: 0.000000000000000000D 00

RANGE		TIME IN RANGE	TIME/ REAL TIME	TIME/ COMPUTE TIME
0	17777	0.203772180000D 07	0.10000D 03 %	0.00000D 00 %
17776	37776	0.000000000000D 00	0.00000D 00 %	0.00000D 00 %
37775	57775	0.000000000000D 00	0.00000D 00 %	0.00000D 00 %
57774	77777	0.000000000000D 00	0.00000D 00 %	0.00000D 00 %

*** EXPERIMENT #5 RESULTS ***

NO COMPUTE TIME RESULTS DETERMINED.

* RESULTS FROM MACHINE 1 *

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.202233540000000000D 07
TOTAL COMPUTE TIME: 0.000000000000000000D 00

RANGE		TIME IN RANGE	TIME/ REAL TIME	TIME/ COMPUTE TIME
0	3777	0.201848200000D 07	0.99810D 02 %	0.00000D 00 %
3776	7776	0.000000000000D 00	0.00000D 00 %	0.00000D 00 %
7775	13775	0.384950000000D 04	0.19030D 00 %	0.00000D 00 %
13774	17777	0.000000000000D 00	0.00000D 00 %	0.00000D 00 %

*** EXPERIMENT #5 RESULTS ***

NO COMPUTE TIME RESULTS DETERMINED.

* RESULTS FROM MACHINE 1 *

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.20184338000000000D 07
TOTAL COMPUTE TIME: 0.00000000000000000D 00

RANGE		TIME IN RANGE	TIME/ REAL TIME	TIME/ COMPUTE TIME
0	777	0.00000000000D 00	0.00000D 00 %	0.00000D 00 %
776	1776	0.20131224000D 07	0.99741D 02 %	0.00000D 00 %
1775	2775	0.00000000000D 00	0.00000D 00 %	0.00000D 00 %
2774	3777	0.00000000000D 00	0.00000D 00 %	0.00000D 00 %

THE FOLLOWING MEASUREMENTS DETERMINE THAT THE ADDRESSES 000776->001176 MAY BE USED TO REPRESENT THE DOS-11 IDLE LOOP. THESE ADDRESSES ARE THEN USED IN SUBSEQUENT MEASUREMENTS.

*** EXPERIMENT #5 RESULTS ***

NO COMPUTE TIME RESULTS DETERMINED.

* RESULTS FROM MACHINE 1 *

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.20185712000000000D 07
TOTAL COMPUTE TIME: 0.00000000000000000D 00

RANGE		TIME IN RANGE	TIME/ REAL TIME	TIME/ COMPUTE TIME
776	1176	0.20146868000D 07	0.99810D 02 %	0.00000D 00 %
1175	1376	0.00000000000D 00	0.00000D 00 %	0.00000D 00 %
1375	1576	0.00000000000D 00	0.00000D 00 %	0.00000D 00 %
1575	1776	0.00000000000D 00	0.00000D 00 %	0.00000D 00 %

THE FOLLOWING RESULTS INDICATE HOW ACTIVITY IS DISTRIBUTED WITHIN THE IDLE LOOP.

*** EXPERIMENT #5 RESULTS ***

NO COMPUTE TIME RESULTS DETERMINED.

* RESULTS FROM MACHINE 1 *

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.20070238000000000D 07
TOTAL COMPUTE TIME: 0.00000000000000000D 00

RANGE		TIME IN RANGE	TIME/ REAL TIME	TIME/ COMPUTE TIME
776	1036	0.58669350000D 06	0.3421D 02 %	0.0000D 00 %
1035	1076	0.73145740000D 06	0.3644D 02 %	0.0000D 00 %
1075	1136	0.11410730000D 06	0.5685D 01 %	0.0000D 00 %
1135	1176	0.47049800000D 06	0.2344D 02 %	0.0000D 00 %

IN THE FOLLOWING RESULTS, THE IDLE LOOP ADDRESS HAS BEEN USED TO DETERMINE THE AMOUNT OF TIME SPENT DOING USEFUL COMPUTING. THE COMPUTING IS DEFINED TO BE USEFUL IF THE INSTRUCTIONS BEING EXECUTED ARE OUTSIDE THE IDLE LOOP.

FOR THE FIRST RESULTS, THE COMPUTER WAS LEFT IN ITS IDLE STATE, AND THE ADDRESSES FOR THE IDLE LOOP WERE PUT IN THE QUAIRA-COMPARATORS, SO THAT COMPUTE TIME COULD BE CALCULATED.

*** EXPERIMENT #5 RESULTS ***

THE FOLLOWING TABLE INDICATES HOW LONG EACH OF 2 CPU'S SPENDS EXECUTING IN THE SPECIFIED REGIONS OF MEMORY, WHILE THEY INTERACT VIA A COMMUNICATIONS LINK.
CPU #1 = MATH PDP11/20
CPU #2 = ENGINEERING PDP11/20

* RESULTS FROM MACHINE 1 *

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.20395295000000000D 07
TOTAL COMPUTE TIME: 0.21683700000000000D 05

RANGE	TIME IN RANGE	TIME/ REAL TIME	TIME/ COMPUTE TIME
0 776	0.00000000030D 00	0.0000D 00 %	0.0000D 00 %
1176 177777	0.38221000000D 04	0.1874D 00 %	0.1763D 02 %
0 177777	0.20395295000D 07	0.1000D 03 %	0.9406D 04 %
776 1176	0.20358029000D 07	0.9981D 02 %	0.9389D 04 %

THE FOLLOWING MEASUREMENTS WERE DONE ON THE MACRO COMPILER.

*** EXPERIMENT #5 RESULTS ***

THE FOLLOWING TABLE INDICATES HOW LONG EACH OF 2 CPU'S SPENDS EXECUTING IN THE SPECIFIED REGIONS OF MEMORY, WHILE THEY INTERACT VIA A COMMUNICATIONS LINK.
CPU #1 = MATH PDP11/20
CPU #2 = ENGINEERING PDP11/20

* RESULTS FROM MACHINE 1 *

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.257399631000000D 08
TOTAL COMPUTE TIME: 0.109531734000000D 08

RANGE	TIME IN RANGE	TIME/ REAL TIME	TIME/ COMPUTE TIME
0 776	0.000000000000D 00	0.0000E 00 %	0.0000D 00 %
1176 177777	0.108115039OD 08	0.4200D 02 %	0.9871D 02 %
0 177777	0.257399631OD 08	0.1000E 03 %	0.2350D 03 %
776 1176	0.149155075OD 08	0.5795D 02 %	0.1362D 03 %

MEASUREMENT OF FILE TRANSFER FROM MACHINE #1 TO MACHINE #2

*** EXPERIMENT #5 RESULTS ***

THE FOLLOWING TABLE INDICATES HOW LONG EACH OF 2 CPU'S SPENDS EXECUTING IN THE SPECIFIED REGIONS OF MEMORY, WHILE THEY INTERACT VIA A COMMUNICATIONS LINK.
CPU #1 = MATH PDP11/20
CPU #2 = ENGINEERING PDP11/20

* RESULTS FROM MACHINE 1 *

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.9474163000000000D 07
TOTAL COMPUTE TIME: 0.4829083000000000D 07

RANGE	TIME IN RANGE	TIME/ REAL TIME	TIME/ COMPUTE TIME
0 776	0.0000000000D 00	0.0000D 00 %	0.0000D 00 %
1176 177777	0.3502912700D 07	0.3697D 02 %	0.7254D 02 %
0 177777	0.9474163000D 07	0.1000D 03 %	0.1962D 03 %
776 1176	0.5968787500D 07	0.6300D 02 %	0.1236D 03 %

* RESULTS FROM MACHINE 2 *

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.9474163000000000D 07
TOTAL COMPUTE TIME: 0.4829083000000000D 07

RANGE	TIME IN RANGE	TIME/ REAL TIME	TIME/ COMPUTE TIME
0 776	0.0000000000D 00	0.0000D 00 %	0.0000D 00 %
1176 177777	0.3965379000D 07	0.4185D 02 %	0.8211D 02 %
0 177777	0.9474162800D 07	0.1000D 03 %	0.1962D 03 %
776 1176	0.5509380000D 07	0.5815D 02 %	0.1141D 03 %

MEASUREMENT OF FILE TRANSFER FROM MACHINE #2 TO MACHINE #1

*** EXPERIMENT #5 RESULTS ***

THE FOLLOWING TABLE INDICATES HOW LONG EACH OF 2 CPU'S SPENDS EXECUTING IN THE SPECIFIED REGIONS OF MEMORY, WHILE THEY INTERACT VIA A COMMUNICATIONS LINK.
CPU #1 = MATH PDP11/20
CPU #2 = ENGINEERING PDP11/20

* RESULTS FROM MACHINE 1 *

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.773030190000000000D 07
TOTAL COMPUTE TIME: 0.392873510000000000D 07

RANGE	TIME IN RANGE	TIME/ REAL TIME	TIME/ COMPUTE TIME
0 776	0.000000000000D 00	0.0000D 00 %	0.0000D 00 %
1176 177777	0.330269230000D 07	0.4272D 02 %	0.8407D 02 %
0 177777	0.773030190000D 07	0.1000D 03 %	0.1968D 03 %
776 1176	0.442526790000D 07	0.5725D 02 %	0.1126D 03 %

* RESULTS FROM MACHINE 2 *

ALL TIMES IN MICROSECONDS
TOTAL REAL TIME: 0.773030190000000000D 07
TOTAL COMPUTE TIME: 0.392873510000000000D 07

RANGE	TIME IN RANGE	TIME/ REAL TIME	TIME/ COMPUTE TIME
0 776	0.000000000000D 00	0.0000D 00 %	0.0000D 00 %
1176 177777	0.312410110000D 07	0.4041D 02 %	0.7952D 02 %
0 177777	0.773030150000D 07	0.1000D 03 %	0.1968D 03 %
776 1176	0.460640990000D 07	0.5959D 02 %	0.1172D 03 %

```
C
C                FILE NAME: DEXP5.EXP
C
C PURPOSE OF EXPERIMENT #5:
C       -TO MONITOR MEMORY ACTIVITY IN 2 PDP11/20 COMPUTERS
C        AT THE SAME TIME. TYPICALLY, THE TWO COMPUTERS MIGHT
C        BE TALKING TO EACH OTHER ACROSS A COMMUNICATIONS LINK.
C
C       -TO FILL IN THE FOLLOWING TABLE:
C        1) ADDRESS RANGE
C        2) TIME IN RANGE
C        3) TIME IN RANGE/TOTAL EXPERIMENT TIME
C        4) TIME IN RANGE/COMPUTE TIME
C THE SYSTEM IS DEFINED TO BE DOING USEFUL COMPUTING, HENCE
C COMPUTE TIME, IF ONE OR BOTH OF THE MACHINES IS EXECUTING
C OUTSIDE OF ITS WAIT LOOP.
C
C THE 4TH ENTRY IN THE TABLE IS OPTIONAL, DEPENDING ON
C WHETHER RANGE #3 OF THE QUADRA-COMPARATORS CONTAINS
C THE ADDRESS OF A PROGRAM WAIT (OR IDLE) LOOP.
C
C
      SUBROUTINE DEXP5
C
      FYUFSOBM UWUWGM
      IMPLICIT INTEGER (A-Z)
      INTEGER LOW(2,4),HIGH(2,4),TVOVFU(5),TVOVFL(5)
      LOGICAL COMPUT
      COMMON /CONFIG/NODE,UNIT
      COMMON /PHMCOM/COMPUT
      COMMON /TVCOM/TVOVFU(5),TVOVFL(5)
      DATA MASK/017777/
C
C QUADRA-COMPARATOR RANGES SHOULD BE SET UP USING THE
C TEST PROGRAM. IF COMPUTE TIME RESULTS ARE DESIRED,
C THEN RENGE #3 OF EACH QUADRA-COMPARATOR MUST CONTAIN
C THE ADDRESS FOR THE WAIT LOOP.
C
C
C SEE IF COMPUTE TIME IS BEING USED.
C
      COMPUT=.FALSE.
      WRITE(6,12)
12    FORMAT(' QC RANGE #3 = WAIT LOOP? Y,N',/' ')
      READ(6,14) REPLY
14    FORMAT(A1)
      IF (REPLY.EQ.'N') GO TO 20
C
      DUNQVU>/USVF/
C
31    DUOUJOVF
C
C SET UP LOGIC UNITS TO RECEIVE QUADRA-COMPARATOR OUTPUTS.
C
C LOGIC UNIT #1 = A => 1QC02->FF3->1SW8I5
C
      L1:=A
C
C LOGIC UNIT #2 = A => 1QC03->FF4->1SW8I6
C
      M3;>B
C                                  - 54D -
C SET UP 8X8 SWITCH MATRICES.
C
C QUADRA-COMPARATOR OUTPUTS (VIA FLIPFLOPS)
```

```
C QUADRATURE COMPARATOR OUTPUTS (VIA INTERFACE)
C
30      QC0=0
        QC1=1
        QC2=5
        QC3=6
C
C TIMER & EVENT COUNTER INPUTS.
C
        TV1B0=0
        TV1B1=1
        TV2B0=2
        TV2B1=3
        TV3B0=0
        TV3B1=1
        TV4B0=2
        TV4B1=3
        TV5B0=6
        TV5B1=7
C
C REAL TIME & COMPUTE TIME
C
        RTIME=7
        CTIME=4
C
        UNIT=1
        CALL SW8DIS
        CALL SW8CON(QC0,TV1B0,QC1,TV1B1,QC2,TV2B0,QC3,TV2B1,
        1 RTIME,TV5B0,CTIME,TV5B1)
C
        QC2=2
        QC3=3
        UNIT=2
        CALL SW8DIS
        CALL SW8CON(QC0,TV3B0,QC1,TV3B1,QC2,TV4B0,QC3,TV4B1)
C
C
C SET UP TIMER & EVENT COUNTERS
C
        DO 35 UNIT=1,5
35      CALL TVSET(1,1,1,3,1,1)
C
C TV #5 OVERFLOW IS 'OR'ED WITH TV #4 SINCE ONLY HAVE
C FOUR INTERRUPT LINES. SPECIAL TEST IS MADE IN TVOVFL.
        TVOVF0(5)=0
        TVOVF1(5)=0
C
C SET UP INTERRUPT GENERATOR AND OVERFLOW COUNTS.
C
        DO 40 UNIT=1,4
        TVOVF0(UNIT)=0
        TVOVF1(UNIT)=0
40      CALL IGSET(TVOVFL,UNIT)
C
C
        RETURN
        END
```

```
C
C                FILE NAME: DEND5.EXP
C
      SUBROUTINE DEND5
C
      IMPLICIT INTEGER (A-Z)
      INTEGER QCLOW(2,4),QCHIGH(2,4),TVOVFU(5),TVOVFL(5),
     1 TVU(2),TVL(2)
      LOGICAL IGFLG,COMPUT
C
      DOUBLE PRECISION DPTVU(5),DPTVL(5),PRTIME,PCTIME,MAX
C
      COMMON /CONFIG/NODE,UNIT
      DUNNUO pUWDUNpUWUWG1)6*-UWUWG2)6*
      COMMON /QCCOM/QCLOW(2,4),QCHIGH(2,4)
      COMMON /PHMCOM/COMPUT
      COMMON /PHMSYS/OUTUNT,IGFLG
C
C MAX = 2.D0**32
      DATA MAX/4294967296.D0/
C
      WRITE(OUTUNT,10)
10    FORMAT(' ',' *** EXPERIMENT #5 RESULTS ***',/' ',/' ',
     1 '   THE FOLLOWING TABLE INDICATES HOW LONG EACH OF ',
     1 /' ','2 CPU''S SPENDS EXECUTING IN THE SPECIFIED REGIONS OF',
     1 /' ','MEMORY, WHILE THEY INTERACT VIA A COMMUNICATIONS LINK.',
     1 /' ',3X,'CPU #1 = PATH PDP11/20',/' ',3X,'CPU #2 = ',
     1 'ENGINEERING PDP11/20',/' ',/' ')
      IF (.NOT.COMPUT) WRITE(OUTUNT,11)
11    FORMAT(' ',/' NO COMPUTE TIME RESULTS DETERMINED.')
C
C READ AND CONVERT ALL TIMES TO DOUBLE PRECISION
C
      DO 20 UNIT=1,5
      CALL TVREAD(TVU,TVL)
      CALL DI2DF(TVU,DPTVU(UNIT))
      CALL DI2DF(TVL,DPTVL(UNIT))
C
C ADJUST FOR POSSIBLE OVERFLOW
      DPTVU(UNIT)=DPTVU(UNIT) + TVOVFU(UNIT)*MAX
      IPTVL(UNIT)=IPTVL(UNIT) + TVOVFL(UNIT)*MAX
C CONVERT TO MICRO-SECONDS
      IPTVU(UNIT)=IPTVU(UNIT)/10.D0
20    DPTVL(UNIT)=DPTVL(UNIT)/10.D0
C
C LOOK AT RESULTS LEADING FROM EACH QUADRA-COMPARATOR UNIT
C ON DIAGRAM.
C
      DO 100 UNIT=1,2
C
      WRITE(OUTUNT,24) UNIT
24    FORMAT(' ',/' ','^ RESULTS FROM MACHINE ',I2,' *')
      IF (.NOT.COMPUT) DPTVL(5)=0.D0
      WRITE(OUTUNT,30) DPTVU(5),DPTVL(5)
30    FORMAT(' ',/' ','ALL TIMES IN MICROSECONDS',/' ',
     1 'TOTAL READ TIME: ',D25.16,/' ',
     1 'TOTAL COMPUTE TIME: ',D25.16)
      WRITE(OUTUNT,32)
32    FORMAT(' ',/' ',5X,'RANGE',10X,
     1    'TIME',13X,    'TIME/',11X,      'TIME/',/' ',20X,
     1 'IN RANGE',9X,'READ TIME',7X,'COMPUTE TIME',/' ')

C
D UW2 ' UW3 BSF UO RO VOJJ 2          - 56D -
C TV3 & TV4 PRE ON QC UNIT 2
```

```
      ITVLO=2*UNIT-1
      ITVHI=2*UNIT
IQC IS USED TO INDEX INTO 'LOW' & 'HIGH' ARRAYS.
      IQC=0
C
C LOOK AT BOTH TIMER & EVENT COUNTERS ON EACH UNIT, AND
C BOTH BUFFERS OF EACH.
C
      DO 40 ITV=ITVLO,ITVHI


      IQC=IQC+1

C % REAL TIME FOR BUFFER 0
      PRTIME=DPTV0(ITV) / DPTV0(5) * 100.D0
C % COMPUTE TIME FOR BUFFER 0
      PCTIME=0.D0




QCLOW(UNIT,IQC),QCHIGH(UNIT,IQC),DPTV0(ITV),
     1 PRTIME,PCTIME
44    FORMAT(' ',05,1X,06,3X,D17.10,3X,D11.4,' %',3X,D11.4,' %')
C
      IQC=IQC+1
C
C % REAL TIME FOR BUFFER 1
      PRTIME=DPTV1(ITV) / DPTV0(5) * 100.D0
C % COMPUTE TIME FOR BUFFER 1
      PCTIME=0.D0
      IF (COMPUT) PCTIME=DPTV1(ITV) / DPTV1(5) * 100.D0
40    WRITE(OUTUNT,44) QCLOW(UNIT,IQC),QCHIGH(UNIT,IQC),DPTV1(ITV),
      1 PRTIME,PCTIME
C
100   CONTINUE
C
      WRITE(OUTUNT,200)
200   FORMAT(' ',/' ',/' ','OVERFLOW COUNTS FOR TIMER & EVENT COUNTERS',
      1 /' ',3X,'UNIT',3X,'BUFFER',3X,'COUNT')
      DO 250 UNIT=1,5
250   WRITE(OUTUNT,251) UNIT,TVOVF0(UNIT),UNIT,TVOVF1(UNIT)
251   FORMAT(' ',3X,I2,8X,'0',6X,I2,/' ',3X,I2,8X,'1',6X,I2)
C
C RESET INTERRUPT GENERATOR IF NECESSARY
C (I.E. FIRST PASS THROUGH THIS CODE)
      IF (IGFLG) CALL IGRES(1,2,3,4)
      RETURN
      END
```

References

1) Banks, W., Morgan, D., "A Computer Controlled Hardware Monitor: Hardware Aspects", Proceedings of International Meeting on Mini-Computers and Data Communications, Liege, Belgium, January, 1975.

2) Goodspeed, D., "Experiences With a Programmable Hardware Monitor", Master's essay, University of Waterloo, 1974, (CCNG Internal Report).

3) Mellor, P., "A General Purpose Load Generator", Master's essay, University of Waterloo, 1974, (CCNG External Report).

4) Morgan, D., Banks, W., Colvin, W., Sutton, D., "A Performance Measurement System for Computer Networks", Proceedings of the IFIP Congress, 1974.

5) Morgan, D., Banks, W., Goodspeed, D., Kolanko, R., "A Computer Network Monitoring System", submitted for publication in "IEEE Transactions on Computers".

6) Sutton, D., "A Summary and Proposal for the Monitoring of Computer Systems", Master's thesis, University of Waterloo, 1974, (CCNG External Report).

APPENDIX E

OPERATING SYSTEM RELIABILITY

OPERATING SYSTEM RELIABILITY

I INTRODUCTION

Although difficult to define precisely, the basic concept of software reliability is clear: that the software system should perform its intended function. This definition is equally applicable to operating systems and other types of software.

IBM (30) makes a useful distinction between "reliability" and "availability." "Reliability" is used to indicate the absence of errors and "availability," the ability to continue system operation in spite of errors. The term "recoverability" has also been used to describe the concept of availability. From a user's viewpoint there is essentially no distinction between the two--both represent the ability of the system to perform the user's task correctly. There may, however, actually be a tradeoff between the two, since increased availability usually implies a larger system, which is thus likely to contain more errors. In this document "reliability" will be used in a sense including both these concepts.

At one time, efficiency of all types of programs, and operating systems in particular, was the principal consideration in program design. More recently, reliability has come to be considered a primary goal.

Tsichritzis and Ballard (34) offer the following reasons for emphasizing reliability rather than efficiency:

As equipment becomes cheaper and faster, the pressure to drive it "hard" is diminishing, thus programs which are not as efficient as possible can be tolerated. Unreliable software is not effective no matter how efficient it is. In some applications the cost of a system failure is much higher than the cost of the system itself, for example, process control applications. It is usually possible to tune an inefficient system to achieve a greater degree of efficiency, but in most cases it is very difficult to rescue an unreliable system. An unreliable system may corrupt data which are very expensive to recreate. The result of inefficiency is obvious--one has to wait longer; unreliable software may have hidden errors which can violate system and user data without any outward indication. The results of an error might only be discovered much later.

A variety of approaches to system reliability have been used. Some of the major ones are: design of the system in "levels," proof of correctness, use of structured programming, protection of parts of the system from each other, improved debugging techniques, in-line checking for correct functioning, audit programs to check system function periodically, and recovery programs to allow continued operation in spite of errors.

Other considerations, such as choice of an implementation language, management of large software projects, and the impact of hardware errors on software

systems, although important, are outside the scope of this discussion.

Historically, the need for reliable systems was first recognized by the designers of special purpose real-time systems which had to be continuously available to service a time-critical application. One of the first such systems was No. 1 ESS (5, 10), developed by Bell Labs for controlling local telephone exchanges. (It is also notable as one of the first major applications of audit programs, although the audit routines were initially developed primarily to detect corruption of data caused by hardware errors.) More recently, computer manufacturers have begun to place much more emphasis on reliability in their general purpose operating systems. For example, IBM's latest system, OS/VS2 Release 2, places much more emphasis on reliability of the system and protection of the system from users than previous IBM operating systems.

II MODULARITY

The design of software systems in a "modular" fashion has been recognized as desirable for many years. Modularity is considered an important part of design for reliability, but the importance of how functions are assigned to modules is emphasized.

An important extension of the concept of modularity is the design of a system as a hierarchy of "levels of

abstraction." This concept has been proposed both in connection with bottom up and top down system design. In the bottom up approach to "level" design, successive levels are designed to provide added facilities using the facilities provided by lower levels; the lowest level uses only the facilities provided by the hardware. In the top down approach to "level" design, first the highest level, which provides the desired features is designed. During its design, the need for lower levels is identified; these are then designed, and so on, until the last level designed requires only facilities provided by the hardware.

Whether the levels are designed top down or bottom up, the objective is to restrict interactions between levels to calls from higher to lower levels, and to restrict inter-level access to data to explicit parameters passed by calls. If this is successfully carried out, then testing of the system or attempts to prove its correctness will be greatly simplified. This advantage is gained since the independence of levels makes it possible to consider levels individually for proof of correctness or testing purposes.

Usually, a "level" structure will be very useful in the design of a system, but problems may be encountered. The usual central difficulty is assigning functions to levels so that all required conditions are satisfied (18). A common problem is that the innermost level of the system will not have access to I/O devices since this is provided by other

levels, thus causing difficulties in writing statistical or error messages from the innermost level. This problem may be circumvented, as was done in SUE (31), without breaking any of the technical rules about interaction between levels, but the concept of requests flowing only inward is still clearly violated. Other difficulties in such systems involve starting up and shutting down the system and fitting debugging aids into the system.

The "virtual machine" concept is a different form of separation of operating system components to achieve greater reliability. The term "virtual machine" is often used to describe the combination of hardware and software facilities provided by a level of a system designed as a hierarchy of levels of abstraction. Here, the term is used to mean "a hardware-software duplicate of a real existing computer system in which a statistically dominant subset of the virtual processor's instructions execute directly on the host processor in native mode" (8). (The reasons for the requirements in the definition need not be considered at length--the basic idea is that a program running on a virtual machine appears to be running on a real machine and most of the program's instructions are actually executed directly by the real machine.) The object is to make one real computing system appear to be several independent computing systems not only to the users but also to the operating system(s). The virtual machines are created by a

small "Virtual Machine Monitor" which, because it is small, can be made quite reliable. Then, except for the implicit competition for resources such as CPU and channel time, several operating systems can run independently on one CPU. In particular, if one system crashes, it affects only its own users. The idea of a reliable Virtual Machine Monitor running several possibly unreliable systems is intuitively appealing as a mechanism for increasing overall system reliability. It has in fact been shown (8) that under reasonable assumptions about how to quantify reliability, N single user operating systems running under a Virtual Machine Monitor are more reliable than a single N user multiprogramming system.

This virtual machine concept can be very useful in some circumstances, particularly when debugging a new or changed operating system, but it must be used cautiously in a production environment because if used unwisely it may produce an unacceptable overhead.

III PROOF OF CORRECTNESS

The only way to be certain that a software system functions correctly is a "proof of correctness." In general it is not possible to prove the correctness of a large system directly, unless it has been written with the idea that its correctness will later be proved. Even then, a proof may not be possible, but proof techniques may still

be used to provide test data which can be shown to test all parts of the system, or, those parts of the system believed to be "critical" may be proven correct and more traditional debugging techniques applied to the remainder of the system.

Appropriate modularization of the system is critical to a proof of correctness. Present proof techniques cannot be applied directly to large programs, but if a system has been properly modularized, the individual modules may be proved correct and their correctness will then imply the correctness of the complete system. The conclusion that the correctness of components implies the correctness of the complete system is not an easy one to make however, since it is usually very difficult to demonstrate that modules do not have any unexpected interactions on data which would invalidate the criteria for a proper modularization of the system. Even if modules can be proven correct individually, the effort required will be considerable. Current estimates indicate that about three man-months are typically required to prove the correctness of a 200 line routine. Since an operating system will be three or four orders of magnitude larger than this, it can be seen that a very large investment in time would be required to prove the correctness of an operating system, even if special problems did not cause proofs to become more difficult than those of programs typically used in proof of correctness attempts.

Instead of using analytical methods to prove program

correctness, which is usually quite difficult, similar analytical methods may be used to determine an exhaustive set of test cases. If the set of test cases can be proven to be exhaustive and the program processes them correctly, the program is then known to be correct. This should be contrasted with the methods considered in Section IV, in which a set of test cases is developed which is expected to exercise all parts of the program but which is insufficient to prove the program correct.

Perhaps the best-known example of an operating system designed using the proof of correctness approach is Dijkstra's THE multiprogramming system for the EL X8 (9). The approach used was to design the system bottom-up, as a hierarchy of abstract machines, prove the correctness of the system a priori, and use exhaustive testing bottom-up to locate coding errors (the testing method used is similar to the method used by Brinch Hansen, as described in Section IV). Because of the design of the system, particularly the use of synchronizing primitives and the structuring of the nucleus as a group of co-operating sequential processes, the set of relevant test cases was small enough to allow exhaustive testing. The proof that co-operation between processes was correct was carried out in three main stages: It was demonstrated that in performing a task, a process could generate at most a finite number of tasks for other processes. Then it was shown that if some task was waiting

to be performed, not all of the processes could be idle. Finally, it was shown that the system could not become deadlocked.

There are, however, various difficulties which prevent the extensive use of correctness proofs to improve the reliability of operating systems. One of the most obvious is the effort required with present proof techniques to prove the correctness of large programs. Although several non-trivial programs have been proven correct, operating systems are so large that even if an operating system could be proven correct using available methods, the effort required to do so might not be justifiable. (It has been suggested, however, that even if a proof is not carried to completion, the effort of stating the properties to be proven leads to sufficiently increased understanding of the program to make the attempt worthwhile (4).)

A second difficulty is the parallelism which exists at least conceptually within an operating system. Proving the correctness of a routine becomes extremely complex if other routines or a second activation of the same routine may modify data used by the routine while it is executing. This difficulty can be circumvented most easily by restricting parallelism in the system. For example, all interrupts are disabled whenever a routine in the SUE kernel is executing, thus guaranteeing that each kernel routine will run to completion before any other routine is activated (3). Such

a simple technique cannot be used in all systems because of performance difficulties. Notably, in a multiprocessor configuration it would be necessary to prevent more than one processor from executing any part of the kernel at one time. Such a "single lock" approach has been applied in multiprocessing systems and has proven to be a significant bottleneck in such systems. Although it is significantly more complex, a "multiple lock" approach to restricting parallel execution of individual parts of the kernel is now considered necessary, at least for multiprocessing systems. For example, the multiprocessing version of OS/VS2 Release 2 (20) uses multiple locks. In addition to complicating proofs, this also raises the possibility of deadlock within the kernel.

A third difficulty is that "traditional" proofs of program correctness are formulated in terms of a functional relationship between initial and final values (it also being necessary to prove that the program halts). We do not normally expect an operating system to halt, so there are no final values and additionally, input and output are interleaved in a complex manner. Therefore, we must prove that individual routines, which either terminate and are re-activated or execute cyclically with a recognizable "idle" point, perform their particular functions correctly. We must also demonstrate that proper relationships exist between the various operations.

A fourth difficulty is that many operating system properties are time-dependent. (Even servicing a queue on a first in-first out basis involves a timing relationship.) Traditional proof techniques are not well-suited to proving the correctness of such timing relationships.

A fifth difficulty is that no proof is possible that the formal assertions about a program used in the proof process actually represent the desired properties. Thus, a proof of correctness of a system may actually prove, not that it does what is wanted, but that it does something else, as specified in the formal assertions.

Another difficulty is that "proofs" of program correctness may not actually be proofs in the mathematical sense: Harlan Mills states "'proof of program correctness' is a relative term by today's editorial standards, which means that a certain level of formality has been shown in a plausibility argument and that is all it means" (23). There is a great danger that informal or sketchy "proofs" may miss important details, as was discovered, for example, in the proof of the SUE timer manager (4), in which some important insights occurred at the most detailed level of the proof. Finally, proofs, at least if prepared by hand, are themselves subject to error. "There is no foolproof proof of the correctness of a program or a program segment" (23).

An interesting variation on proof of correctness methods has been used by Harlan Mills (2, 23) in the design

of large programs which must be reliable (though not specifically operating systems). The technique, used in conjunction with a project organization called "the chief programmer team," makes use of top-down, structured programming. Although the programs are not formally proven correct, programmers are taught proof of correctness methods and can think in terms of correctness proofs as they are designing and coding the system. The objective is to produce error-free code initially, rather than to find the errors after the code has been written. (As Dr. Mills has pointed out, one is likely to have more confidence in a program in which no bug has ever been found than in a program in which several bugs have been located and corrected, but which now contains no known bugs.) The use of this method in the production of a complex on-line system for the New York Times was remarkably successful; however, it has been suggested (19) that other factors influenced the success of this system and thus, that although the method is an encouraging development, its worth has not yet been clearly demonstrated.

IV TESTING

Probably the oldest technique for improving the reliability of software is debugging--running a program with test data and checking the output to determine if the program functioned properly, and correcting errors as they

appear. Unless accompanied by a formal proof that a set of test data has completely exercised the program, no "perfect" debugging run or set of runs proves that a program is correct, but for simple programs it is often straightforward to develop test data which will be sufficient to detect at least most of the errors in a program. When debugging large programs, particularly a program such as an operating system, whose behaviour depends on the timing of inputs, simple methods of applying test data to a program may be completely inadequate.

Even for a trivial program it is usually not possible to test all possible combinations of inputs, so it is clearly necessary to develop methods for generating test data which will be likely to isolate most of the errors in a program. Next to attempting all possible inputs, the most thorough type of test will attempt to exercise all possible control paths in a program. This may be possible for small programs, and automatic methods can be used to identify the control paths (13), although human aid will likely be required to identify some "impossible" paths which appear possible to the automatic method, and to generate the data which will cause each path to be executed.

Unfortunately, for most programs, the number of possible control paths is so large that testing all of them also becomes infeasible. In this case, the closest approximation to an exhaustive test which is possible is to

test all sections of code and all conditional branch possibilities. Here, automatic methods may be used to construct a set of paths through the program which will exercise all branches, choosing the paths so that a small set of test data can be used. Again, human aid is required to determine if a constructed path is actually possible and to devise the test data which will cause a path to be executed.

Probably the most difficult type of problem discovered by software testing is one which depends on the timing of inputs to the system. Even when such a problem is discovered during testing, it may not be resolved quickly (or at all), if it cannot be reproduced at will. This type of problem usually arises from incorrect synchronizing of processes within the system, so a well-designed system should rarely suffer from such problems. If such a problem does occur, usually the ability of the system to trace its own operations will be critical in determining the difficulty of locating the error. Routines which trace the activity of the system are useful in resolving many types of system problems, but when the sequence of operations within the system is the source of a problem, standard debugging aids such as dumps may prove nearly useless.

If a plan for testing of a system is developed before coding of the system begins, testing will usually be greatly simplified, and thus reliability of the finished system can

be achieved more easily. Brinch Hansen has described the testing of the RC4000 operating system (6). The method he used is a good general technique, particularly for systems designed as a series of levels. The system was tested beginning at the innermost level, and once each level was completely tested, it could be used in testing of higher levels. Determining whether the system was functioning properlyzwas made easy by a small trace routine which was devised before the system was coded. Although this approach is certainly a useful testing method, the reverse approach of designing, coding, and testing a system in a top-down manner has also been advocated and used in the design of large reliable systems, as described in the previous section.

Although, as Dijkstra has pointed out, debugging can show the presence of errors, but not their absence, in many instances it may not be considered practical to attempt to remove all the errors from a large software system. Thus standard debugging techniques may be applied in an attempt to remove as many of the errors as possible from the system. If such an approach is used, it is especially valuable to be able to estimate how many errors are in the system, both during testing and when the system is "completed" and made available for general use. To obtain such estimates, a model of error discovery and correction may be used in conjunction with data collected during system testing.

One such model is described in (32). A number of simplifying assumptions are made, principally: that the failure rate of the program is proportional to the number of errors it contains, that the program does not increase in size as testing progresses, and that the correction of errors does not introduce any new errors. The first assumption is clearly at best approximately true and is probably justifiable only because the failure rate is the most easily obtained measure of a program's error content. The second assumption will likely be satisfied quite closely in most cases. The third assumption, however, is almost certainly not true--at best one can hope that when the model is applied during initial system testing, the rate of error removal will exceed the rate of introduction of new errors sufficiently for the model to be useful.

Using the model one can obtain a simple relation between the initial number of errors in the system, the rate of error removal, the number of instructions in the system, and the failure rate of the system. Using failure rate data, for at least two separate times during system testing, one can then obtain estimates of the initial number of errors in the system and the rate at which they are being removed. Making use of the estimated values of the parameters, it is now possible to estimate either the present number of errors in the system, or the time at which the number of errors will have been reduced to a specific

value. Clearly, however, due to difficulties with the third assumption of the model, estimates of the time required to achieve an extremely small number of remaining errors are likely to be of no value. While this model is obviously very approximate, there are many difficulties in using a more exact model, since the parameters which would be needed are difficult to estimate.

It has been stated that "the reliability of a software product usually depends on the effectiveness of testing procedures during the development stage" (22). Although future developments in proof of correctness methods may make this statement untrue, the statement clearly reflects the importance of testing in the development of present operating systems.

V SELF-CHECKING

In the near future, it is likely that very few, if any, large software systems will be proven correct. All operating systems are therefore likely to contain unknown errors. At present, because software is so easy to change, the number of errors in a system does not even approach zero through extended use of the system, but instead decreases asymptotically to some positive value. Even in a system which is ultimately to be proven correct, errors will exist when the system is first coded. (Also, it is unwise to assume that any system will necessarily remain free from

errors when subjected to "improvements" and local modifications.) Thus in any system, whether or not it is ultimately to be proven correct, it is important to minimize the effect of errors on the system.

The following actions are typically required in a system which may contain errors but which is intended to operate reliably in spite of them: The behaviour of the system is observed and compared with expected behaviour. (The fact that certain types of behaviour are unexpected is actually a subtle form of redundancy in the system. Redundancy will be seen as a key element throughout the consideration of error detection and recovery.) When a discrepancy is detected, an attempt is made to diagnose its cause and the occurrence of a discrepancy is recorded. If the cause is an error in the system, appropriate actions are performed to recover from the error.

This section and the two following sections consider techniques for detecting errors. Section VIII considers possible error recovery actions.

The reliability of an operating system can be improved by including code in the system to check the validity of data structures before and/or after they have been processed by system routines. If data structures are checked before they are used, errors previously introduced will not be propagated. If data structures are checked after they have been modified, the routine causing an error will be

immediately identified. Although cleverly constructed tests can catch many errors with a small amount of processing, frequently extensive checking will introduce an unacceptable overhead, so it is often necessary to restrict the checks to be made. Designing simple data structures will make checking easier and will also tend to reduce the number of errors in the system, since complex data structures provide greater opportunity for error in their use.

Clearly, all parameters passed to system routines by user programs must be checked for validity, but parameters passed from one system routine to another might also be checked. A common result of failing to check parameters is that an error in one routine causes another routine to fail because of an invalid parameter, thus concealing the original source of the error. Even checking of all parameters passed between system routines is likely to introduce too much overhead, so decisions must be made concerning which parameters are to be checked. One advantage of a "level" structure is that it makes such a decision straightforward--parameters should be checked as they are passed from a higher level to a lower level, but not in the reverse direction, and probably not between routines on the same level. This approach has been applied in the SUE operating system (31). The SUE nucleus contains a number of processes arranged in a tree structure. Communication can occur only between ancestors and

descendants on the tree (not between brothers, etc.) and all data passed toward the root of the tree is carefully validity checked, whereas data passed away from the root of the tree is usually assumed to be correct.

Various techniques may be used to make a program check its own operation to some extent. Probably the most thorough form of self-checking would be to have two separate algorithms perform the same function and then compare results. Unfortunately, this would essentially double the size of the program and halve its execution speed, and would also create problems when a single data structure is both the input and output of an operation, so this appears to be of little practical value in general.

The most common form of checking the correct operation of a routine is examining the integrity of data structures affected by the routine. The question of determining the integrity of data structures is considered in the next section, and so will not be considered here.

Many self-checking techniques are concerned with preventing an unexpected occurence from causing disaster. For example, branching to select one of a number of alternatives should be done only as the result of a positive test--if all but one of the expected possibilities have been tested for, it should not be assumed that the remaining possibility holds. Instead, if none of the expected situations holds, an error should be indicated. This is

particularly useful if input parameters have not been thoroughly checked, but also may detect internal logic problems of the routine. Other coding tricks can be used to help make a routine self-checking, but no general principles seem to underly them.

A further technique which may be used is to monitor the control flow of the system, usually by observing subroutine calls. Valid control flow sequences can be determined from the structure of the system and actual sequences can then be checked against these. This would usually be quite time-consuming, so it has been proposed to perform such checking using an external chip processor with a mini-disk storing the valid control sequences (24, 25).

Another technique which can be used to check for correct operation of a system is the use of "program exercisers." These are programs which provide a routine to be tested with a set of inputs for which the expected results are known and compare the expected and actual results. The major difficulty with this approach is that few routines execute with no side effects (that is, routines frequently modify global variables or save information internally between calls). The effort of updating and maintaining a large number of exercisers and the amount of processing time consumed by this approach are also significant difficulties.

VI AUDITING

Frequently, in-line checking for errors introduces an unacceptably high overhead, particularly in systems operating under real-time constraints. An alternative technique is auditing--periodically checking the correct functioning of the system.

Auditing usually requires less overhead than in-line checking, but cannot provide as timely detection of errors. If an error occurs in a system which uses in-line checking, the error may be detected by the routine in error or the first routine which accesses data which has become invalid. In an audited system, many routines may access invalid data before an audit program determines that an error has occurred. Subsequent operations with the erroneous data may cause other system data to become corrupted, making recovery more difficult and obscuring the original cause of the problem. In the worst case, the system may crash as the result of an error before the appropriate audit program is invoked.

Usually, audit programs are invoked periodically, either after the expiration of a specified time interval or after a specified number of executions of a system routine, such as the dispatcher. Some audit programs, often called "emergency" audits, are invoked only when there is reason to believe a problem exists--either because another audit program has detected an error or some measure of system

performance has not been satisfied. Invoking emergency audit programs when performance degradation is detected must be planned very carefully, since the audit program will cause further performance degradation by occupying the CPU with "unproductive" work. This may produce a serious performance loss if the detected degradation was actually due to an overload rather than software errors. This difficulty has been encountered in No. 1 ESS installations, in which an overload has caused sufficient performance degradation to invoke an emergency audit, which has then caused enough further degradation to invoke a higher level emergency audit (35). Ideally, one would like to ignore performance degradation caused by overloading when deciding whether to invoke an emergency audit, but in practice it is usually very difficult to determine the reason for loss of system performance.

The purpose of audit programs is to detect erroneous system operation, usually as reflected in erroneous data structures. Although the principle of audit programs is straightforward, many considerations are involved in the design of a good system of audit programs (1).

It is tempting to design audit routines to check for specific error conditions which have been observed or are expected to occur. The audit routines, however, should be designed not to detect specific problems which may be anticipated, but to determine whether the audited data

structure is correct, so that unexpected problems will be detected. The philosophy should be to determine whether data is correct not to determine how it became mutilated.

Often data structures used by the system will be difficult to audit. It might then seem simplest to create special supplemental data which would be easier to audit. The danger of this approach is clear: the audited data may be correct when the "real" data is in error, or errors in the audited data may not indicate any actual problem, except in the special code for creating the audit data. In addition, the "for audit only" data approach creates extra work for the system, decreasing efficiency, and wastes storage.

It is important to audit all system data, even if not apparently vital. Difficulties with obscure data will likely impact other system data eventually and may produce problems which are hard to track down if the originally erroneous data has not been audited.

Auditing of data structures can be simplified and made more effective if data structures are designed to be audited. Some features which aid auditing are: Storing a code in each element of a data structure which identifies the type of element. Using both forward and backward pointers in lists, or at least terminating chains by linking the last element to the first, forming a cycle, rather than using a special value to terminate the chain. Using other

forms of data redundancy, and using standard patterns in designing data structures.

The two most commonly used audit techniques are checking for consistency of redundant information and range checks on values. Range checks involve advance knowledge of the values which are valid in a particular field. The ranges are usually peculiar to the particular situation. For example, a field which is expected to contain an address must contain a value within the addressing range of the machine, and a field which is expected to contain a day of the year must have a value in the range 1-366. The commonest examples of verifying the correctness of redundant information are checking "point to, point back?" in doubly linked lists and checking for closure of lists which are supposed to be circular. Usually, subtler types of redundancy also exist but are harder to check. For example, all the main storage in the system should either be on an "available" list or assigned to some activity in the system. It should be possible to check whether any storage has become "lost" or has been assigned twice; however, in many systems the time to perform such a check would be prohibitive.

One type of redundancy which can sometimes be usefully added to data structures is a simple checksum of all the fields in the data structure. Each routine which modifies the data structure also modifies the checksum, preferably by

computing a change to it rather than simply recomputing it by examining all the fields in the data structure. Audit routines can then recompute the checksum and compare it with the stored checksum to detect some types of errors in the data structure. Unfortunately, many types of program logic errors will generate incorrect data "early" enough in their processing so that the checksum will be correctly modified and no error will be detected by this method. One type of problem almost certain to be detected is a "wild" store not even intended for this data structure. This type of problem may not be common enough to justify the extra storage cost and processing overhead involved in storing and computing checksums. (In OS/360 for some models (12), an analogous technique is used to restore code damaged by hardware malfunctions. Presumably it could also be used to restore code damaged by software malfunctions, but OS/360 does not contain routines to detect such software-caused damage.)

Audit programs can also check for activities in the system which are not advancing properly and for expected measures of system performance which are not being met. This technique has been used extensively in No. 1 ESS, in which reasonably accurate expected timings are known for most system functions. In a general purpose operating system many functions can reasonably be active for arbitrarily long times. In all systems, however, there should be some functions which can be expected to complete

in a specific time. For example, an I/O operation (except on a teleprocessing device) should not be active for more than a few seconds--if an audit program which was invoked at intervals of a few seconds discovered the same I/O operation marked in progress on two successive activations, it could conclude that the function was not being performed properly and corrective action was required.

The corrective action to be taken when an audit program discovers an error depends heavily on the system and the type of error discovered. Most error recovery techniques are common to errors discovered by audit programs, in-line checks, and protection mechanisms; they are discussed in Section VIII. One special consideration when an error is detected by an audit program is that the error may have caused further errors, or that it may have resulted from some other, as yet undetected, error. Thus, it is often desirable to invoke other audit programs, particularly those concerned with related data structures.

VII PROTECTION

If an operating system may contain errors, one technique to help minimize the effect of those errors is to protect parts of the system from each other.

Early in the development of supervisory systems, the desirability of protecting the system from accidental or malicious damage by user programs was recognized. Although

In most systems there is no danger of malicious damage to one part of the system by another part, errors give rise to the possibility of accidental damage. Techniques for protecting the system from itself usually rely on the same hardware features which are used to protect the system from users, and similar software techniques. The essential idea is to restrict each system component to the use of those hardware features which it requires.

For example, all large computers currently being manufactured have a "supervisor" mode for use by the operating system; user programs run in a "problem" mode which prevents them from accessing I/O devices, the protection hardware, etc. The supervisory mode is not required, however, by all of the operating system. Simply by restricting supervisory mode to those parts of the system which require it will allow earlier detection of some errors, since they will result in an attempt to execute a "privileged" instruction by an unauthorized routine. Also, errors in routines operating in supervisory mode tend to have much more serious effects than those in other routines, thus by concentrating all "privileged" instructions into a few routines and executing only those routines in supervisory mode, the impact of many errors will be decreased. This principle can usually be applied easily to a system designed in "levels" since it should only be

necessary to execute the lowest level or a very few of the lowest levels in supervisory mode.

Hardware memory protection can be used to protect the code and data of each routine from modification by other routines; however, if many routines access common data it may not be possible to protect data so that only authorized routines can modify it. On a machine such as a System/360 with a limited number of "protect keys" it may not be possible to provide any such protection since the number of protect keys not used by the operating system effectively determines the number of concurrent user jobs which may be executed. This restriction does not apply to "virtual" System/370's, which can provide protection through the address translation mechanism, so that Release 2 of OS/VS2 makes use of eight different protect keys for parts of the operating system (20).

In some systems it may be possible to protect programs and unchanging data from accidental modification by placing them in read-only memory. This technique has been used in No. 1 ESS, which uses a read-only "program store" to hold program code and data which is not normally changed during system operation, and a read-write "call store" to hold changing data. The technique used in No. 1 ESS involves a lengthy process to modify the read-only memory offline and would thus not be appropriate for systems subject to frequent changes. Hardware which would allow more rapid

loading of read-only memory would make this technique appropriate for protecting almost any operating system.

Very elaborate protection structures within operating systems have also been proposed, usually in conjunction with a sophisticated protection algorithm for use relative to user programs. Typically, such an organization restricts supervisory mode and complete memory access to a small "kernel" program. Then, in order to transfer control between system routines, or obtain access to common data structures, all parts of the system must obtain permission from the kernel, which checks all requests against the authorization of the requesting routine. While this provides a very thorough protection mechanism and may be suitable for use with user programs, it tends to produce a very high overhead, which may be unacceptable in a production system and is almost certainly unacceptable in a real-time system. Although such thorough protection may not be practical at the present, it is possible that special hardware may in the future reduce the overhead sufficiently to make the technique more widely applicable.

A similar proposal (37) is to restrict all access to interface data structures to a special interface system, so that no other routines physically access the data structures. All modifications of or references to interface structures are made symbolically. This provides a degree of independence from the form of the data structure as well as

protection of the data structure. By implementing primitives for stack and queue handling within the interface system, other components of the system can be simplified somewhat. Although this technique has several attractive features it is clear that there is a high overhead in repeatedly resolving symbolic references during execution.

VIII ERROR RECOVERY

Although detection of errors is a very important aspect of system reliability, by itself it is of limited value. If the system is to continue operating in spite of errors, it must contain routines which will provide recovery from errors. The corrective action to be taken when an error is discovered depends heavily on the system and the type of error discovered. Some general possibilities are outlined below.

In some cases, it may be possible to repair the error and continue normal operation. This may be possible, for instance, if a list has become "unlinked" and enough correct information remains to recreate the appropriate list structure. For example, list elements may fill a contiguous area of core, so any "lost" elements may be found and replaced on the appropriate list, or the elements may be locatable through some other list structure to which they also belong and which is still valid. If this can be done, ideal recovery has been achieved, since system users will be

essentially unaware of the occurrence of an error. Even if the error has not been repaired it may be possible to continue operation from the point of error, effectively ignoring the error. This is usually a very dangerous alternative.

If the system cannot repair the error, it may be possible for the system to continue to provide some level of service while a diagnostician attempts to repair the system software, using services provided by the system to aid his diagnosis and repair.

In other instances, it may be possible to return the system, or some part of the system, to a checkpoint at which the system was functioning correctly. (Some provision must be made to ensure that this does not result in an infinite loop between the checkpoint and the point at which the error is discovered. Since many system errors result from the occurrence of an unusual combination of circumstances, which is not likely to recur immediately, there is usually a good chance of re-establishing correct functioning of the system.) Particular care must be taken if online files are being updated, since such a procedure could result in duplicating a change. If a return to a checkpoint would involve "backing up" a user program which might be updating an online file, it is generally preferable to abort the program and thus allow human intervention rather than risk destroying a file.

Otherwise, it may be necessary to restart or abort some system routines. This allows continued operation of the system but will usually have some impact on system users. In a transaction oriented system, some transactions in progress may be lost; in a batch job system, some jobs may be aborted. However, if restarting of system routines is well planned, it should be possible in most cases to reconnect a system routine to the function being performed when the routine failed. If the routine then fails again immediately, while still attempting to perform the same function, it will usually be necessary to abort the process requesting service from the system routine (either the request is invalid but is not being detected as such or it is valid but has encountered a bug in the system routine).

If all else fails, operation of the system may be shut down. Although this is not a desirable alternative, an orderly shutdown is still preferable to allowing errors to propagate through the system, causing unknown damage before eventually producing a messy system crash. Also, if the system can be stopped and restarted in an orderly manner, it may be possible to salvage some system tables from the failing system, so that when a fresh system is loaded it may resume processing at nearly the point where the old system failed. This technique has been used in several XDS-940 systems (36), and a related method has been used in the Distributed Computing System (See Section IX).

In general, it is desirable to design a hierarchy of recovery routines, so that if the recovery routine initially invoked is unable to effect recovery, another routine will be available. This philosophy has been applied in the design of OS/VS2 Release 2 (11), in which a stack of "Functional Recovery Routines" is maintained, which contains entries corresponding to system routines invoked but not yet terminated. Status information is also stored to aid the processing of the recovery routines. When a software failure occurs, the routines on the stack are executed, beginning with the routine most recently placed on the stack, until one recovers successfully. Curiously, if all system recovery routines fail to recover from a software error, an error recovery routine supplied by the user program may be invoked. It is unlikely that a user program will be able to take any effective recovery action, but this may provide an opportunity for the program to terminate gracefully, leaving any files it was modifying undamaged.

Error recovery routines may be executed either on an emergency basis, locking out normal system functions, or in parallel with normal system operation. The former is usually simpler to implement, since problems of other routines referencing erroneous data are avoided. Complex recovery routines may, however, dictate use of the latter technique in order to maintain acceptable response time.

Error recovery routines may themselves contain errors,

so a well-designed system should be prepared to cope with such errors. Clearly this is a difficult problem, since designing a second level of detection and recovery routines simply pushes the problem to another level. In particular, OS/VS2 Release 2 apparently makes no attempt to cope with this problem, although the number of error recovery routines is very large: "a system failure can be considered to be the result of two program errors: the first, in the program that started the problem; the second, in the recovery routine that could not protect the system" (30). While this is certainly preferable to a single error crashing the system, improvement is still needed.

IX COMPUTER NETWORKS

Computer networks provide both additional challenges in the design of reliable systems and greater opportunities for achieving reliability at reasonable cost. In this section, consideration of computer networks will be limited principally to homogeneous networks of closely co-operating computers. In such cases, we may consider the entire network to be under the control of a single distributed operating system.

Some of the additional challenges to reliable operation are due to the presence of communication lines in the system, which normally cause more hardware problems than CPU

hardware does. This aspect is outside the scope of the present discussion.

Other challenges to reliability are due to the increased complexity of the system, caused by the need to co-ordinate multiple CPU's and the necessity of preventing a software failure at one node from propagating to other nodes and thus disabling the entire network.

The chief opportunity for increasing reliability afforded by computer networks is due to the presence of multiple, reasonably independent CPU's. This allows reliability to be increased in two different ways. Firstly, when a software failure does occur which disables a node of the network, provided the failure can be confined to that node, the network can continue to provide at least some service to some of its users and may even continue operation so as to make the failure transparent to all or nearly all users. If all service were being provided by a single central computer, an unrecoverable failure in the operating system running on the one CPU would interrupt service to all users until the system could be restarted. Secondly, if appropriate hardware is available, some other processor in the network may be able to restart a failed processor, eliminating the manual intervention normally required following a critical operating system error.

Clearly, if a network is so designed that one node is responsible for overall control and supervision of the

network, the above advantages are diminished since failure of the supervisory node will halt the entire network. In the following, only networks in which control functions are distributed throughout the network or can be switched to different nodes will be considered.

The Distributed Computing System (28) is an example of a system designed to make use of the opportunities in networking for increasing the reliability of the total computer system. In the DCS, each processor is controlled by a nucleus which must be functional for that processor to be operative in the system. All other system functions, such as resource allocation, are performed by processes which can execute in any node. For example, the system process which allocates processes to CPU A can execute on CPU B.

If a node fails, either because of hardware problems or an error in the nucleus, processes executing on that processor are lost. Once failure of the node is recognized, a new copy of the nucleus is loaded into the failed processor by a bootstrap routine, which is loaded by one of the other processors. This loading process does not affect system tables in the processor being loaded. The first task of the new copy of the nucleus is to attempt to notify the initiators of processes which were running in the failed CPU that the initiated process has failed. Since data in the system tables may have been damaged, precautions are taken

to prevent erroneous data from causing further difficulties. If the data is correct, appropriate processes will be notified of the failure and thus fresh copies of system (and possibly user) processes can be started. The possibility of missed notification of process failure is taken into account by arranging for each process to send, periodically, a status check message to each process it has initiated to ensure that the process is still running.

An important type of process in the DCS is the "status checker." Messages are sent to the status checkers when unusual conditions, such as the failure of a process to accept a message, are detected. The status checkers attempt to determine whether such conditions are caused by a processor overload or a processor failure. When a processor's nucleus fails to respond to several messages sent by status checkers and a sufficient percentage of the active status checkers have certified this condition, processor failure is assumed to have occurred and a nucleus restart, as outlined above, is initiated. Several status checkers execute simultaneously, on different processors, in order to make the failure detection mechanism insensitive to failure.

There are two ways in which the recovery mechanism of the above system might be extended. Recovery of a failed processor might be made quicker or more complete. These two goals could be pursued independently or together.

Recovery could be made more complete by attempting to use more information from system tables in the failed processor. Clearly there are dangers in this approach since as more use is made of possibly erroneous data, there is an increasing chance of creating further problems, either for the processor being recovered or the network as a whole. The objective would be to obtain more information about processes executing on the failed processor so that more sophisticated recovery than simply creating fresh copies of processes and restarting them from the beginning could be attempted.

The simplest technique to achieve this objective might seem to be the use of checkpoint facilities, which would normally enable all processes to be restarted from a point in their execution shortly before the failure of the processor. In addition to possible difficulties with checkpointing considered in the previous section, here there is the problem of restarting part of the network from a checkpoint while the rest of the network continues to run. What happens, for example, if the process allocator for CPU A is running on CPU B and the nucleus on CPU B fails, causing all processes on CPU B to be restarted from a checkpoint, thus causing the process allocator for CPU A to use an outdated process table? Some problems of this type can be avoided by running system processes only on the processors for which they are "responsible," but the same

problem arises whenever two communicating processes are running on different processors. It may be more practical to pursue the goal of improved recovery in conjunction with the techniques for faster recovery, described below.

In order to provide faster recovery in a computer network it is usually necessary to have processors designated as backups for other processors, so that when one processor fails, another is immediately available to assume responsibility for its processing. In order to avoid severe overloading of individual processors due to failure of other processors, it may be advisable to divide the workload of a failed processor among two or more other processors. The choice of processors to take over the function of a failed processor is usually determined by the geographic configuration of the network.

If a processor is to take over rapidly the functions of another processor, it must contain information about the current state of the other processor. Generally this does not mean that all system data must be duplicated and that all functions performed by a processor must be immediately transmitted to its backup processor. Rather, sufficient data must be maintained in a backup processor to allow the data in a failed processor to be reconstructed rapidly. Usually it is possible to maintain such a set of skeleton data without creating undue overhead.

An example of a network system in which these

techniques could be profitably applied is an air traffic control system for a large area. Clearly, rapid takeover of functions from a failed processor is critical, and the geographic pattern of the network makes easy a logical assignment of processors to back up other processors. This example is discussed more extensively in (25).

X CHOOSING RELIABILITY TECHNIQUES

Even if it were possible to build an operating system which contained no bugs and thus could run indefinitely without errors, using present techniques the cost of such a system would prevent anyone from undertaking such a project.

A completely satisfactory measure of the reliability of a system does not seem to be available, but a measure which is easy to obtain and which seems to be useful is the percentage "up-time" of the system. (For example, this is the measure used as a target in the design of No. 1 ESS--2 hours downtime in 40 years.) At least in terms of this measure, the cost of a system increases quite rapidly as one attempts to eliminate the last few percent, and then tenths of a percent, of downtime. Thus, organizations must be satisfied with systems which are not totally reliable, but which have a sufficient degree of reliability for the particular application.

Unfortunately, there is no accurate way to predict the degree of reliability of a system until it has been coded

and tested, and possibly not until it has been used under production conditions. There is thus still a considerable amount of guesswork and intuition required in choosing the reliability techniques to be used in a system. Still, it is possible to state certain guidelines for the choice of techniques for reliability when designing a system within cost constraints.

As in most phases of system design, it is first essential to understand the purpose of the system, the environment in which it is to be used, and its relationship with that environment. Then, within this framework, it is necessary to define the consequences of system failures of various types, in terms of the effect on system service, and to consider the effects of failures of various durations and frequencies.

Next, one must decide how much it is worth to avoid the consequences of the types of system outages considered. This places an upper bound on the cost which can be expended in improving the reliability of the system.

It is also necessary to consider design constraints on the system, other than reliability, which may affect the reliability methods which can be used. For example, the amount of core available for the system may be limited, thus placing a limit on the amount of code for in-line checking or audit programs which can be included. More importantly, in many applications the response time of the system is

extremely important, so the overhead introduced by reliability techniques must be considered. For example, it may be determined that the overhead produced by in-line checks would degrade response time unacceptably, so extremely timely error detection must be sacrificed for the lower overhead created by audit programs.

Finally, in terms of all the preceding considerations, the appropriate reliability tools and techniques must be chosen, in terms of their costs. Clearly, the set which achieves the desired degree of reliability at the lowest cost, and meets all other system restrictions, should be chosen.

XI CONCLUSION

Although the need for reliability of operating systems was first recognized in conjunction with the development of real-time control systems, it has since been realized that reliability is important for all types of operating systems. The reliability of general purpose operating systems has been recognized as important partly because of the increasing dependence of many organizations on general-purpose computer systems. Several operating systems have been designed in an academic environment which had reliability as one of their prime objectives, and computer manufacturers are also now beginning to place greater emphasis on the reliability of their operating systems.

Many techniques have been proposed and used to improve the reliability of operating systems, but the effort required for the proof of correctness approach and the system overhead produced by most other methods are still problems preventing the routine creation of reliable software. More sophisticated hardware may decrease the overhead produced by some reliability methods, and advances in proof techniques may make proofs of operating system correctness practical in the future. The combination of advances in techniques for increasing the reliability of operating systems and greater dependence on computer systems, and thus greater need for reliable systems, will doubtless lead to the creation of commercial operating systems much more reliable than most of those in use today.

BIBLIOGRAPHY

1. Almquist, R. P., J. R. Hagerman, R. J. Hass, R. W. Peterson, and S. L. Stevens. Software protection in No. 1 ESS. Proceedings of the International Switching Symposium, 1972. p565-569.

2. Baker, F. T. Chief programmer team management of production programming. IBM Systems Journal, vol. 11, no. 1. p56-73.

3. Ballard, Alan and Dennis Tsichritzis. Structure and correctness of software systems. Proceedings, Session '73, The Annual Conference of the Canadian Information Processing Society. p324-340.

4. Ballard, Alan and Dennis Tsichritzis. System correctness. SIGPLAN Notices, vol. 8, no. 9. p38-41.

5. Beuscher, Hugh J., George E. Gessler, D. Wayne Huffman, Peter J. Kennedy, and Eric Nussbaum. Adminstration and maintenance plan. Bell System Technical Journal, vol. 48 (October 1969). p2765-2815.

6. Brinch Hansen, Per. Testing a multiprogramming system. Software--Practice and Experience, vol. 3, no. 2. p145-150.

7. Buxton, J. N., and B. Randell. Software Engineering Techniques; Report on a conference sponsored by the NATO science committee, Rome, Italy, 27th to 31st October, 1969.

8. Buzen, Jeffrey P., Peter P. Chen, and Robert P. Goldberg. Virtual machine techniques for improving system reliability. Record, IEEE Symposium on Computer Software Reliability, New York City, April 30-May 2, 1973. p12-17.

9. Dijkstra, E. W. The structure of the "THE" multiprogramming system. CACM, vol. 11, no. 5. p341-346.

10. Downing, R. W., J. S. Nowak, and L. S. Tuomenoksa. No. 1 ESS maintenance plan. Bell System Technical Journal, vol. 43 (September 1964). p1961-2019.

11. IBM Systems Reference Library. Introduction to OS/VS2 Release 2 (GC28-0661).

12. IBM Systems Reference Library. Machine Check Handler for System/370 Models 155 and 165, Program Logic Manual (GY27-7198).

13. Krause, K. W., R. W. Smith, and M. A. Goodwin. Optimal software test planning through automated network analysis. Record, IEEE Symposium on Computer Software Reliability, New York City, April 30-May 2, 1973. p18-22.

14. Lampson, B. W. On reliable and extendable operating systems. International computer state of the art report: volume 1, The fourth generation. Infotech Limited. p421-442.

15. Linden, Theodore A. Proving the adequacy of protection in an operating system. SIGPLAN Notices, vol. 8, no. 9. p97-99.

16. Linden, Theodore A. A summary of progress toward proving program correctness. Proceedings of the Fall Joint Computer Conference, 1972, (vol. 41, part 1). p201-211.

17. Liskov, B. H. A design methodology for reliable software systems. Proceedings of the Fall Joint Computer Conference, 1972 (vol. 41, part 1). p191-199.

18. Liskov, B. H. Guidelines for the design and implementation of reliable software systems. Mitre Corporation, Bedford, Massachusetts, February 1973.

19. Liskov, B. H., and E. Towster. The proof of correctness approach to reliable systems. Mitre Corporation, Bedford, Massachusetts, July 1971.

20. MacKinnon, R. A. Advanced function extended with tightly coupled multiprocessing. IBM Systems Journal, vol. 13, no. 1. p32-59.

21. MacWilliams, W. H. Reliability of large real-time control software systems. Record, IEEE Symposium on Computer Software Reliability, New York City, April 30-May 2, 1973. p1-6.

22. Meeker, Robert Edward, jr., and C. V. Ramamoorthy. A study in software reliability and evaluation. Electronics Research Centre, Univerisity of Texas at Austin, Austin, Texas, 1973.

23. Mills, Harlan D. On the development of large reliable programs. Record, IEEE Symposium on Computer Software Reliability, New York City, April 30-May 2, 1973. p155-159.

24. Morgan, D. E. and G. F. Clement. An automated maintenance system for computer networks. Computer Communications Network Group, University of Waterloo, 1973.

25. Morgan, D. E. and G. F. Clement. Unpublished paper.

26. Naur, Peter and Brian Randell. Software Engineering; Report on a conference sponsored by the NATO science committee, Garmisch, Germany, 7th to 11th October, 1968.

27. Randell, B. Operating systems: The problems of performance and reliability. Information processing 71, Proceedings of IFIP conference 71, Ljubijana, Yugoslavia, August 23-28, 1971. p281-290.

28. Rowe, Lawrence A., Marsha D. Hopwood, and David J. Farber. Software methods for achieving fail-soft behavior in the Distributed Computing System. Record, IEEE Symposium on Computer Software Reliability, New York City, April 30-May 2, 1973. p7-11.

29. Scherr, A. L. The design of OS/VS2 Release 2. Proceedings of the National Computer Conference, 1973 (vol. 42). p387-394.

30. Scherr, A. L. Functional structure of IBM virtual storage operating systems, part II: OS/VS2-2 concepts and philosophies. IBM Systems Journal, vol. 12, no. 4. p382-400.

31. Sevcik, K. C., J. W. Atwood, M. S. Grushcow, R. C. Holt, J. J. Horning, and D. Tsichritzis. Project SUE as a learning experience. Proceedings of the Fall Joint Computer Conference, 1972, (vol. 41 part 1). p331-339.

32. Shooman, Martin L. Operational testing and software reliability estimation during program development. Record, IEEE Symposium on Computer Software Reliability, New York City, April 30-May 2, 1973. p51-57.

33. Spooner, C. R. A software architecture for the 70's: Part I--The general approach. Software--Practice and Experience, vol. 1, no. 1. p5-37.

34. Tsichritzis, D. and A. J. Ballard. Software reliability. INFOR, vol. 11, no. 2. p113-124.

35. Ulrich, W. Design of high reliability continuous operation systems. Software Engineering Techniques; Report on a conference sponsored by the NATO science committee, Rome, Italy, 27th to 31st October, 1969. p149-153.

36. Watson, Richard W. Time-sharing System Design Concepts. McGraw-Hill, New York, New York, 1970. Chapter 7.

37. Weissman, L. and G. M. Stacey. An interface system for improving reliability of software systems. Record, IEEE Symposium on Computer Software Reliability, New York City, April 30-May 2, 1973. p136-142.

APPENDIX F

A SURVEY OF COMPUTER NETWORK DEPENDABILITY TECHNIQUES

1. INTRODUCTION

The more organizations become dependent on computers, the more sensitive they become to computer system failures. The importance of computer dependability in real time control systems (e.g., communications systems and traffic control systems) has been recognised for some time. Many general purpose computer users are now becoming aware that they accomplish more on systems that seldom "crash" because of malfunctions than on systems that run very rapidly (and correctly) between frequent "crashes". Consequently, increasing emphasis is being placed by users and vendors on the reliability of the system components and on the dependability of the complete system, including hardware and software.

Care and redundancy are the keystones in building a dependable system. Experience indicates that most potential sources of errors can be eliminated by exercising care when designing, implementing, or modifying the hardware, software, or data structures of a system. However, human beings error and components suffer failures, so despite the amount of care exercised, some malfunctions will occur during productive use of the system. These should at least be detected when they occur so that some appropriate action can be taken to prevent avoidable damage from occurring to the system, data, or innocent users.

Despite progress made in recent years in component

reliability, redundant hardware still is normally necessary to achieve the desired level of system hardware dependability. Redundant data in the system is essential in order to detect and recover from many types of hardware or software malfunctions. Other types of malfunctions can only be detected by observing behaviour of the system, and comparing this with known or expected behaviour derived from analysis of a system model or experience using the system. This information about the behaviour of the system can be kept inside or outside the system, and is, in one sense, redundant information about the system. Thus, there are three forms of redundancy that can be used to enhance system dependability: redundant hardware, redundant data, and redundant information about the system's behaviour. Extra software and hardware of a special nature are usually necessary to make effective use of this redundancy.

Redundancy must be organised and managed to effectively and efficietly achieve a level of dependability. Effective use of redundancy implies the ability to detect errors, both on a system and a unit basis. Cost effective use also requires the ability to locate (diagnose) the source of error and quickly repair or replace the failed unit.

Error detection, diagnosis,and recovery can be performed completely automatically, completely manually, or with some compromise between manual and automatic operation.

The amount of hardware, software, and redundant data

required to achieve a level of dependability is the result of a compromise between the cost of failures and the cost of the facilities necessary to handle them. Few organisations can (or should) afford an eternal machine. Instead, most find that something less than the ideal system is adequate. Costs increase quite rapidly as the ideal system is approached. Experience indicates that the dependability vs. cost curve has approximately the characteristics of the curve of Figure 1.1.

Recoverability of a system is defined as the ability of the system to perform its intended function in the face of faults or errors in the system components. The definition implies that, from a user's viewpoint, the system is transparent to errors or faults.

The degree to which a system is recoverable can be measured in several ways, but from a user's viewpoint, the important parameters are dependability and error free operation.

The ideal recovery process would respond to a fault or error in the system so rapidly that the user could access or be accessed by the system as if the error or fault had not occurred. Furthermore, it would, by reason of speed of recovery or reconstruction, not affect or mutilate any transactions in progress at the time the fault or error occurred--it would be error free.

Thus, recoverability is not necessarily synonymous with

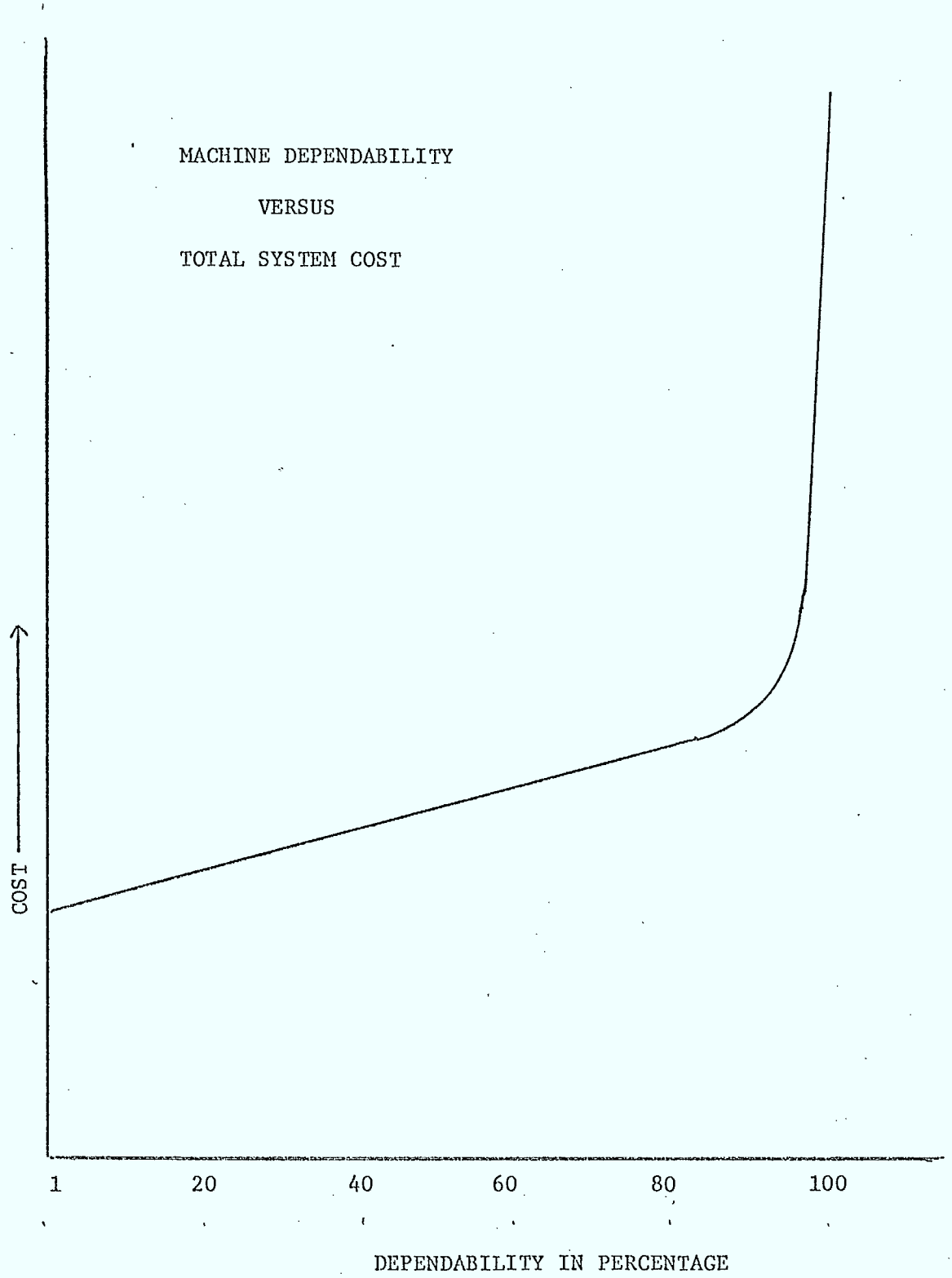

MACHINE DEPENDABILITY
VERSUS
TOTAL SYSTEM COST
COST
1
20
40
60
80
100
DEPENDABILITY IN PERCENTAGE

FIGURE 1.1

reliability. Typically, the technique of adding redundant components to obtain dependability trades reliability (there are more components than if redundancy were not used) for recoverability. From an economic viewpoint, it may well be possible to trade recovery system sophistication for reliability. That is, trade less expensive hardware for more elegant recovery software.

Telephone companies, with their reputation for ultra-dependable service to maintain, were among the first to try to implement dependable hardware and software <1,2,3,4>. Consequently, their electronic switching systems are as dependable as their electromagnetic predecessors. Unfortunately, no commercially available computer system approaches the dependability of such systems. Designers of commercial computers emphasize rapid error-free calculations rather than continuous service, whereas telephone companies emphasize the latter. Reports of ten or more hardware or software "crashes" per week are not infrequent for commercially available computers. Some installations report as many as ten "crashes" per day. In contrast, No. 1 ESS of A.T.&T. is achieving a dependability of twelve hours downtime in forty years <16>.

The techniques that have been used in electronic switching systems to achieve such dependability can be applied to computer systems and networks. This paper reviews some of these techniques, suggests some new tools and tech-

niques, suggests some guidelines for a designer to use in developing dependable computer systems, and finally illustrates the use of these ideas in a computer network. Section 2 describes some sources of errors, section 3 presents some techniques for achieving dependability, and sections 4 through 7 survey tools and techniques for preventing, detecting, diagnosing, correcting, and recovering from hardware and software malfunctions. Section 8 presents guidelines for designing a maintenance system for a computer system or network. Section 9 summarizes the paper and presents several areas requiring study.

2. SOURCES OF ERRORS

There are three categories of error sources in a system: a mistake in design or implementation, a failure in a component, or an error introduced by a human operator. Noise is usually included in the first category.

Design mistakes are not predictable, except that they exist in every system. They are not recurrent, once corrected; thus, the number of such errors in a system decreases with time. The number of hardware design errors approaches zero asymptotically with time because the design becomes stable. However, because software is so easily changed, it rarely becomes stable; thus, the number of software design errors decreases asymptotically with time to some fixed positive number. The value of this number is a function of the rate at which new capabilities are added to the software. The number of software errors increases temporarily after each change. Hardware design errors are sometimes difficult to detect using hardware, so they are usually treated as software design errors.

Hardware component failures are statistically predictable, so their effect on dependability can be predicted analytically. Errors caused by component failure can repeat until the cause of the failure is fixed or the component is replaced. In time, component failures produce more errors than are caused by design mistakes. Software components do not decay with time, but hardware does, so

only hardware contributes to component failures.

Human operator errors are statistically unpredictable, and recur at an unpredictable rate. If they are concerned with hardware, they are treated as component failures.

3. TECHNIQUES FOR ACHIEVING DEPENDABILITY

A dependable system requires techniques to deal with errors from each type of source. By careful application of error prevention techniques such as those discussed in Section 4, many design and implementation mistakes can be caught before the hardware or software is productively used. The other errors must be detected and handled as the module is used. Whereas care is the key to error prevention, redundancy (when properly and carefully used) is the key to error detection, diagnosis, and recovery.

At least one automobile with which we are familiar contains a redundant brake system arranged so that, when a malfunction occurs, the brakes will continue to function; however, the driver is not notified until both the primary and secondary systems have developed failures. He then is rather rudely made aware that there is a problem as his car fails to stop.

As this example illustrates, redundancy can be mismanaged. Although redundancy can be used to achieve some level of dependability without concern for error detection by the equivalent of 'ORing' the outputs of redundant units, this approach has two problems. One is that disaster strikes without warning when the last component fails; the other is economy, for doing without error detection maximises the number of redundant units required to achieve a level of dependability. Efficient and effective use of

redundancy requires the ability to detect errors both on a system and a unit basis. The number of redundant units required for a given level of dependability can be minimised by designing into the system a capability for rapid location and repair of the error source.

In order to keep a system operating effectively, the following cycle of functions should be performed:

A. Observe behaviour of the system, looking at performance and malfunction indicators particularly, in order to detect trouble.

B. Compare observed behaviour with expected behaviour derived from a system model, experience using the system, and/or redundancy within the system.

C. Decide whether a discrepancy exists between observed and expected behaviour. If not, continue observation at step A. Otherwise, notify the maintenance portion of the system so that diagnosis can be performed.

D. Diagnose the cause of the discrepancy by directed observation of system status and behaviour (See below for a more detailed description of a diagnosis procedure).

E. Decide what corrective and/or recovery actions to take, if any. (If deemed necessary, information about the failure and the action taken can be

recorded so that it can be analysed later to validate and/or correct the system model on which some of the expected behaviour is based.)

F. Take action to correct, recover, or bypass the cause of the problem. Continue observation at step A.

Each of these functions, of course, can be performed manually or automatically.

Once an error has been detected by steps A-C above, its cause is diagnosed by performing the following cycle:

A. Select a test appropriate to the type of error;

B. Observe behaviour and status of the system as appropriate for the test;

C. Compare behaviour and status with that expected, based on redundant information and/or components;

D. Decide whether enough information has been obtained to isolate the cause of error;

E. If so, notify maintenance portion of the system; if not, continue diagnostic testing.

In many systems, such diagnosis would be separated into two phases: first, the functional unit that can be replaced with a redundant unit would be identified, then after reconfiguring and/or patching the system so that it can continue to provide service, the necessary detailed diagnosis would be performed to determine the exact cause to allow correc-

tive action to be taken.

Once either temporary or permanent corrective actions and/or repairs are complete, the system could be restarted from where it was when the error was detected (sometimes dangerous), from the beginning (often wasteful), or from the last previous checkpoint, the last assuming facilities are available to save periodically the state of the system to facilitate recovery.

4. MALFUNCTION PREVENTION TOOLS AND TECHNIQUES

Selecting a language that is appropriate for the application and selecting suitably simple algorithms and heuristics are obvious acts to prevent amny software design and implementation mistakes. It is easier to make programming mistakes in a language that is less restrictive in svntax than in more restrictive languages. While useful for dependability, such restrictions can be crippling for the programmer.

Dijkstra, Hoare, and others <5,6,7> have shown that some program structures are more error-prone than others. Programs structured as they recommend are easily understood, so the vast majoritv of bugs are eliminated before the programs are put into productive use. Minimising the number of unconditional transfers is a highlv recommended technique for simplifving program structure. Such structure programming techniques also aid proving programs to be correct. Although such proofs of correctness are not practical now, they may become a viable error prevention technique in the future.

Like program structures, data structures can and should be designed with dependability in mind; hence, thev should be as simple as possible, for very complex data structures are often sources of errors. Programmers are often tempted to save space bv making words in tables serve several different purposes, each purpose determined by the setting of

flags in another word in the table. This practice is incompatible with a dependable system, for such tables cause far more than their share of errors.

Whenever data are entered or modified, or data structures are created or modified, the new data or structure should be checked for being reasonable, in the proper format, and within permissible range(s) of values, and being consistent with existing data or with redundant information inside or outside the system.

Language translators, linkers, and loaders can be written to detect many possible sources of software errors. For example, they can be written to find instructions that could transfer or store wildly as well as instructions that can never be reached.

Tests need not be exhausting to be exhaustive. Complete, yet cleverly constructed tests can catch many design and implementation mistakes before they cause catastrophes. Experimental design techniques and tools such as Latin Squares can be used to reduce the number of tests while giving some level of confidence in the completeness of the testing. Programs can be written to generate the graph of a program, then produce a set of paths to test to cover all the possible paths through the program. This would minimise the number of paths necessary to test to achieve a very high level of confidence in the program.

5. MALFUNCTION DETECTION TOOLS AND TECHNIQUES

Several techniques have been developed for detecting hardware errors in computer systems. Among these are error codes (e.g., parity, Hamming), state validity checks, matrix select validity checks<1>, and matching the results of components performing the same function with the same data and input stimuli <1,3>. Matching yields rapid and highly efficient error detection (i.e., the percentage of errors detected is very high); however, it is expensive, and in the case of duplicated components can lead to ambiguity in identifying which unit failed. Improved dependability can be achieved by voting among three or more identical components. Error codes are used by most manufacturers of computers to detect (and sometimes correct) errors in data when the data are transmitted or stored. The STAR computer designed by Avizienis and Rennels at JPL uses residue codes to achieve dependable computation <14>.

When computer hardware detects problems, the software is normally notified by the interrupt mechanism. Such hardware-detected errors include software-caused problems such as attempts to execute instructions with invalid operation codes or invalid addresses, or to use invalid or incorrectly formatted data. In many systems, this is the only error defense mechanism provided. Nearly all errors can be detected eventually in this way; however, if the error source is software, it usually is some time before the ef-

fects of the error are detected. Often the damage is sufficiently extensive to be catastrophic. There are other techniques to detect software errors which buy time and minimise damage.

An obvious technique for detecting software errors is to build defensive checks into the software so that every entry and exit of every routine is checked for consistency and validity of parameters. Such checks can also verify that routines are being executed in a reasonable and valid sequence, and that the data structures involved with the routines are valid, intact, and consistent. These checks are included in the routines, and consequently are executed whenever the routines are executed. As discussed in a later paragraph, such checks can be performed periodically by software, or by an external system, thus reducing or avoiding the real time penalty.

Another approach is to execute periodically, or whenever trouble is suspected, specially-constructed programs called AUDIT PROGRAMS to check validity, completeness, and consistency of data structures in main and auxiliary storage <1,2,3,4>. Such audit programs base their comparisons on redundant information and structure contained in the data structures or available from the system or from that part of the system's environment which can be sensed by the system. These checks are cheaper than the in-line defensive checks just described because they are not ex-

ecuted as often; however, in-line checks detect trouble more quickly, so damage is less widespread than with audit programs. To maximise the effectiveness of audit programs, data structures should be designed with auditing in mind.

In particular, error detecting can be aided by using data structuring techniques such as: A. An identity code in each data entity; B. Forward and backward pointers in each list; C. Point to -- point back schemes between data structures so that each path between structures is part of a cycle which can be checked for closure; D. Redundant data within the structures so that mangled data can be detected and reconstructed if necessary; E. Standard patterns employed in all data structures.

Not only data but programs can be overwritten, either by software or hardware errors. To determine whether a program still executes properly, it can be exercised using known input data. An error is detected if the output of the program differs from the correct output. This technique is rarely used because of the difficulty of writing and updating such exercisers, the real time used, and because few programs look like black boxes (i.e., all inputs and outputs are passed as parameters, and only volatile local variables are used). Error codes such as hash sums are effective in determining the integrity of programs, and are often used. Such tests can be run periodically and/or whenever trouble

is suspected.

Kane <8> has shown that many errors can be detected by noting when flow of control deviates from expected patterns. Such 'wild transfers' can be detected by observing all transfers of control (or just subroutine calls and coroutine resumes) and comparing the actual control sequence with known valid sequences. These known valid sequences can be stored as valid relationships between routines (for example, X can call A or B; X can be called by B, C, or E; X can be interrupted for I, O, M, or S, but not for P or T). We think that such sequence checking can be done by an external chip processor connected to a mini-disk that holds the relationships, and to a hardware monitor that observes the actual control sequences. In an effort to verify this conjecture, one of the authors has designed and is implementing such a control sequence error detection system. It is based on the hardware monitor described in papers by Morgan, et al <9,10>.

Certain characteristic performance parameters (e.g., response time, throughput, resource utilisation) are useful to indicate when the system is in trouble. When these indicators pass through threshold values, the maintenance system can automatically alerted by a monitoring system so that appropriate techniques can be used to determine the cause of the problem. Software and/or hardware monitoring techniques are capable of providing such parameters.

Most manufacturers use specially-constructed hardware diagnostic programs to diagnose faults and sometimes to detect errors. They are often executed routinely as part of preventive maintenance to detect failing components before they cause damage. These programs are sometimes implemented using special instructions designed to exercise aspects of the hardware for diagnostic purposes.

6. MALFUNCTION DIAGNOSTIC TOOLS AND TECHNIQUES

Bell Labs has had good success using Audit Programs as software diagnostic tools (See Section 5 and <1,2,3,4>). Besides detecting trouble, these programs can determine the extent of damage to data structures and thus help point to possible causes of the problem.

Bell Labs has also used program exercisers as a diagnostic tool. Suspected programs are exercised by executing them with known inputs and comparing the results with known outputs. A dictionary of symptoms and their probable cause is used to decide what the probable cause(s) of any problems is(are). Such maintenance dictionaries are also used with considerable success with diagnosing hardware faults.

Most software diagnosticians find event logs to be a gold mine of helpful information, especially if the system automatically saves a snapshot of the event log immediately atfter the error was detected and before several unimportant events occurred. Besides interrupts, event logs should contain records of the occurrence of selected significant events, such as failing to satisfy a request for main memory.

A debugging subsystem similar to to TSSS in IBM's TSS/360/67 or DDT of DECSYSTEM 10 dramatically reduces mean time to repair (MTTR) for software problems <10,11>. Diagnosticians use this tool to display system status, the event log, registers, and selected areas of storage. It enables

them to modify programs and data, and allows them to observe execution of selected sequences of code. They can use the debugging system to execute selected audit programs to determine the extent of damage.

Chang, Manning, and Metze have discussed many hardware error diagnostic techniques and tools <12>. Such diagnostic techniques have been employed in all critical components in the Bell System's electronic switching systems, and evidence of their use is occasionally found in commercially available computers. Many manufacturers include special diagnostic instructions to aid their diagnsoticians in locating trouble.

All manufacturers have some form of diagnostic programs for their hardware. They range from simple stand-alone exercisers to complex on-line diagnostic systems.

7. MALFUNCTION CORRECTION AND RECOVERY

As mentioned in Section 3, several recovery actions are possible after an error has been detected. The simplest procedure is to ignore the error. The most complicated is to attempt to diagnose and repair the failed component automatically while the system continues to provide a normal level of service.

Two major components of system dependability are hardware dependability and integrity of program and data structure. Both can be achieved by duplication of hardware. If data and program integrity can be provided by another means such as back up from which stable data can be rebuilt, then hardware replication need only be provided to secure the necessary degree of hardware dependability. In general this can be achieved by less than full duplication. It should be noted that it is generally either either impractical or impossible to recreate transient data; hence, duplication must be used where transient data is important.

If all critical units are duplicated, when an error occurs, the standby unit that has been performing the same functions as the active unit, now automatically becomes the active unit. Meanwhile, the failed unit can be daignosed. Assuming an error detection and response mechanism which prevents propagation of the error, this technique, though expensive, nearly eliminates system "crashes" caused by component failures in critical units. Data need not be

recreated after a failure because it is always available and current in both units <1,2,3,4>.

Economies in cost can be realized by a redundancy scheme where roving spares are maintained to replace a failed active unit. The number of spares needed is a function of the dependability and the repair time required to fix a failed unit, but the number of spares is less than the number of active units. Data residing in a failed unit may be lost. The time required to initialise the activated spare makes this scheme slower to recover than the complete duplication scheme.

Neither of these redundancy schemes requires that the replicated units be located near the active units, except for massive data transfers at high rates. Thus, a network of computers of the same type has inherent replication.
Some failures of hardware are intermittent or transient so that the function can be successfully retried. Some errors can be automatically corrected by self-correcting hardware, but such hardware is often very expensive.

If the system can continue to provide service without the failed unit, the failed unit may be automatically bypassed until it can be repaired and placed back in service.

Suppose we have a set of components in the system, each capable of performing the functions of the other components. Suppose the set of functions to be performed are ordered ac-

cording to the importance of continuing to perform them should malfunctions occur. When a malfunction occurs, the failed unit is automatically taken out of service for diagnosis and repair. If necessary, the lowest priority function(s) is(are) dropped, and the system's components are reorganised to perform the remaining functions.

When software fails, an attempt may be made to repair the damage and continue to provide service automatically. This implies the need of sufficient redundancy in the data to locate all the damage and sufficient intelligence in the software to do the repairs, but all this is expensive.

A more modest approach is to provide tools so that a diagnostician can determine the damage and its cause, make the necessary repairs to the software, then allow the system to continue. If appropriate software were available, the system could, in many cases, continue to provide some service while the diagnosis and repairs are being done.

Some systems periodically store the state of the system automatically so that the system can be restarted from one of these points should a failure occur. This procedure is known as checkpoint/restart or rollback. IBM's OS/360 provides this facility as an option, but an informal poll indicates that few installations elect to use it.

Some software errors are the result ofa rare sequence of events. In such cases, merely restarting the system from a convenient checkpoint is often sufficient to get around

the problem.

8. GUIDELINES FOR MAINTENANCE SYSTEM DESIGN

Many techniques and tools have been developed to improve system dependability. Most of these have been discussed briefly in the previous sections.

When designing a maintenance system, how does one decide which of these tools and techniques to employ to achieve a specified level of dependability within a specified cost constraint? Here are some guidelines that should help: A. Define the system, its environment, and the relationships between them. B. Define the consequences of each kind of failure as a function of the completeness of the resulting outage, mean outage time, and distribution of this kind of failure. C. Define what it's worth to avoid these consequences. This provides an upper bound on the cost of protecting the system from these consequences. D. Considering the preceding, determine which tools and techniques are applicable. Define achievable dependability with some combination of the available tools and techniques as a function of their cost. Pick the set the achieves the required dependability at minimum cost.

To illustrate the application of these guidelines, consider the following simple example which, while admittedly less complex than most practical problems, will serve to demonstrate the principles. The example was contrived by

simplifying some real applications studied by the authors.

Assume a traffic control system which covers a geographical area large enough to require ten or more surveillance sectors. Furthermore, the size of the area limits the problem of control of a given unit to three contiguous sectors at any one time. The depth of space occupied by a unit is related to its speed so that system control cannot be lost for more than 30 seconds without danger of collisions between units. If control is lost for more than this time, assume that flow of traffic through the effected sector will be halted; thus, the penalty of loss of control is loss of traffic for half a day at a port of call. Loss of life is not considered in this example. This paragraph has functionally defined the system, its environment, and the relations between the two considering dependability. It is also assumed that the control function is implemented by one or more digital computers.

This simple system description allows a statement of failure consequences to be made as a function of outages. An outage of more than 30 seconds in a sector is not tolerable. A failure that can lose more than one sector (e.g., all sectors in the system) could halt traffic flow completely, thus having severe economic (and perhaps social) consequences. Hence, it is desirable to avoid loss of control and/or surveillance in more than one sector at a time. Outages of more than 30 seconds twice a day render the

system inoperative.

Now what of cost? Assume the revenue lost by sector shutdown is that which might be expected at a large urban airport such as O'hare in Chicago. Thus, a shutdown of operation of half a day represents a loss of more than one million dollars, or in another context, loss of ten percent of the weekly revenue. As a bound, assume that the mean time between such outages of less than a year is not acceptable. This represents an effective cost of 0.5% of the weekly revenue. Furthermore, it requires, considering today's processor technology, a recovery system capable of restoring operations automatically in 30 seconds or less, and not capable of coping with failures more often than once a year.

What then are the possible configurations and techniques? No matter what the implementation, it is a fact of life that programs are never error free. Paraphrasing Dijkstra, testing indicates the presence of errors, not their absence; thus, a recovery system must contain a form of defensive programming. It has been established that audits are economical in real time usage and usually in code space. Several alternative techniques are shown in Figures 8.1a through 8.1c to achieve data integrity and hardware dependability. In each figure, the surveillance units are overlapped of necessity to assure complete coverage of the area. The overlap also helps to achieve dependability of surveil-

lance. Processing in Figure 8.1a would be done at a single centre for the entire system, and the processor at that centre would be duplexed. Each element in its subsystems is provided with an error detection capability , and recovery and diagnostic software are provided as part of the system. The processor must have the capacity to handle the entire system. Outages of this processor (failures to recover in 30 seconds or less) halt traffic in the whole system, and so would affect more than one port of call and create an economic disaster. To prevent this the recovery and error detection systemswill have to be quite sophisticated.

The system of Figure 8.1b has a processor associated with each surveillance unit and communication links only to the extent of passing handover data from one unit to the next. A processor outage does not cause failure of control through a sector since sector surveillance overlaps. The penalty is that each processor must be capable of handling twice the traffic in a sector. Dependability is achieved because two adjacent processors must fail in order to lose sector control. Cost is approximately the same as for the machines in Figure 8.1a, assuming that processor cost and capacity are linearly related. An additional penalty here is that insufficient data exists in adjacent machines to effect a 30 second recovery. An advantage is that, since failures of adjacent machines affect only one sector, the economic penalty is less than in Figure 8.1a, and the

recovery system can be less sophisticated.

The system of Figure 8.1c also has a processor associated with each surveillance unit, but uses communications links to keep the two neighboring data bases up to date so that when an outage occurs, one of the neighboring processors can take over and recover the system in less than 30 seconds.

The data base required in neighboring processors is comprised of only that information required for the 'take over' processor to rebuild path and velocity data for each element in the failed sector. It does not comprise the entire data base. Thus, at an added cost for data links and storage, the objective of a 30 second recovery is achieved. Since each processor is smaller than the ones used in the Figure 8.1a configuration, they are inherently more reliable. If one assumed equal sophistication in the recovery software and hardware, the costs would be about equal. However, since failure of the function of two adjacent surveillance sectors affects only one sector in the scheme of Figure 8.1b, the recovery hardware and software need not be as sophisticated for b as in a. This simplicity can only result in further enhancement of dependability as well as a cost advantage. One also gains the performance advantages that often go with a gracefully degrading system.

The system shown in Figure 8.1c can be thought of as a set of triads, at least from a dependability point of view.

Justification for this division of a computer network into triads is presented in Attachement A. In particular, Attachment A proves that so long as the link failure probability is lower than unity, the system is more reliable than the one shown in Figure 8.1b.

FIGURE 8.1a CENTRALIZED SYSTEM

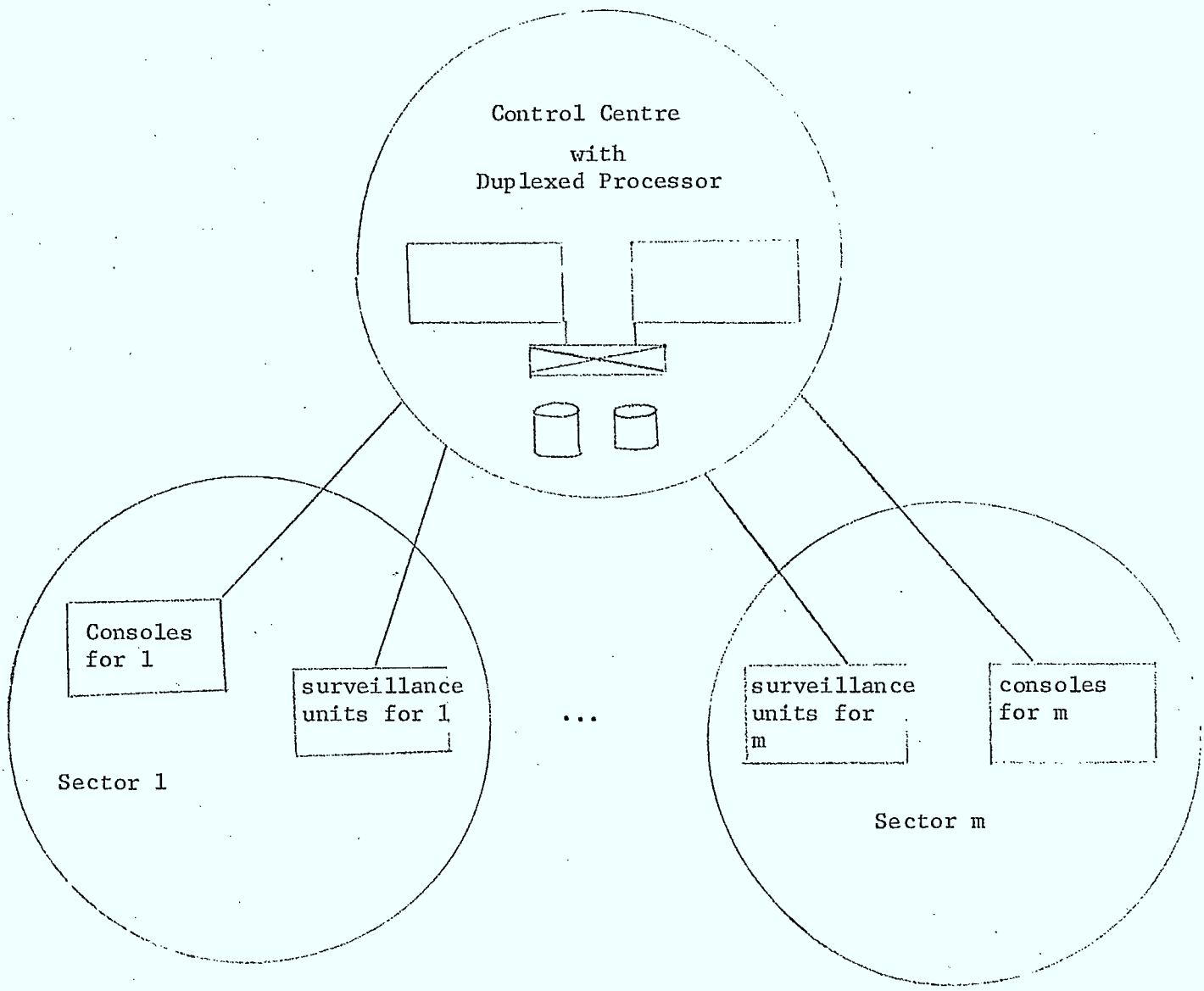


FIGURE 8.1b DECENTRALIZED SYSTEM

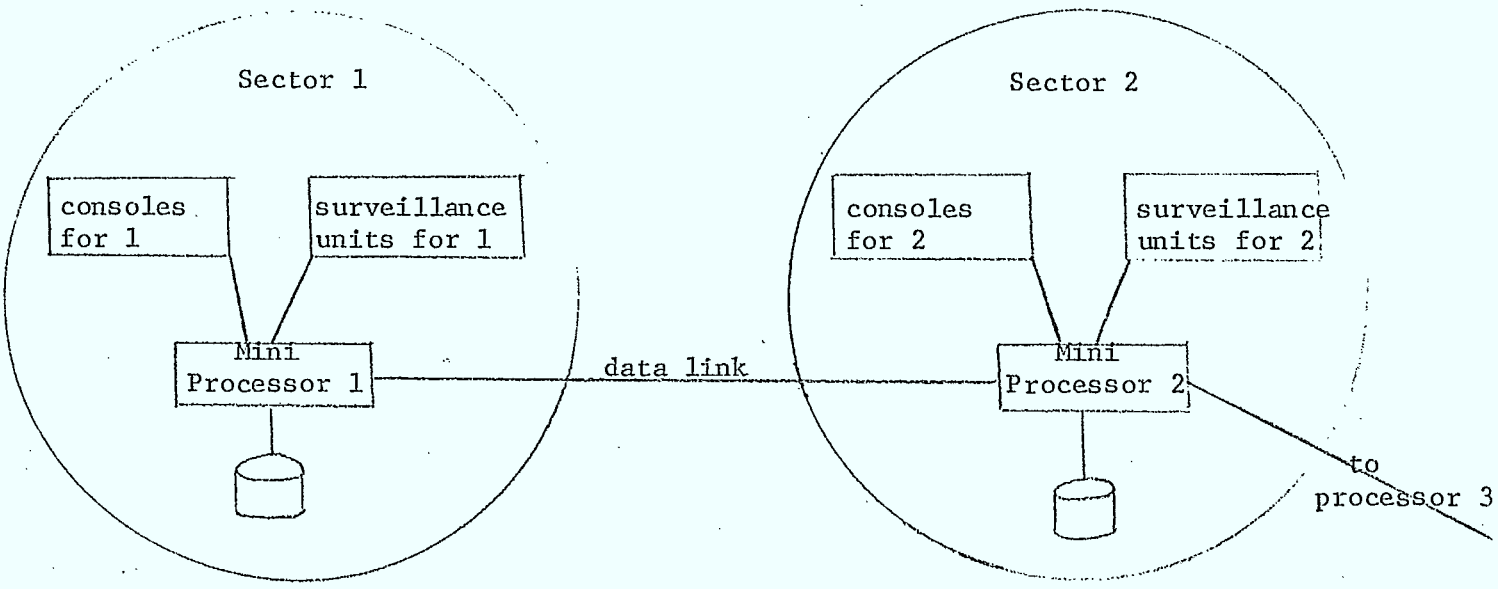
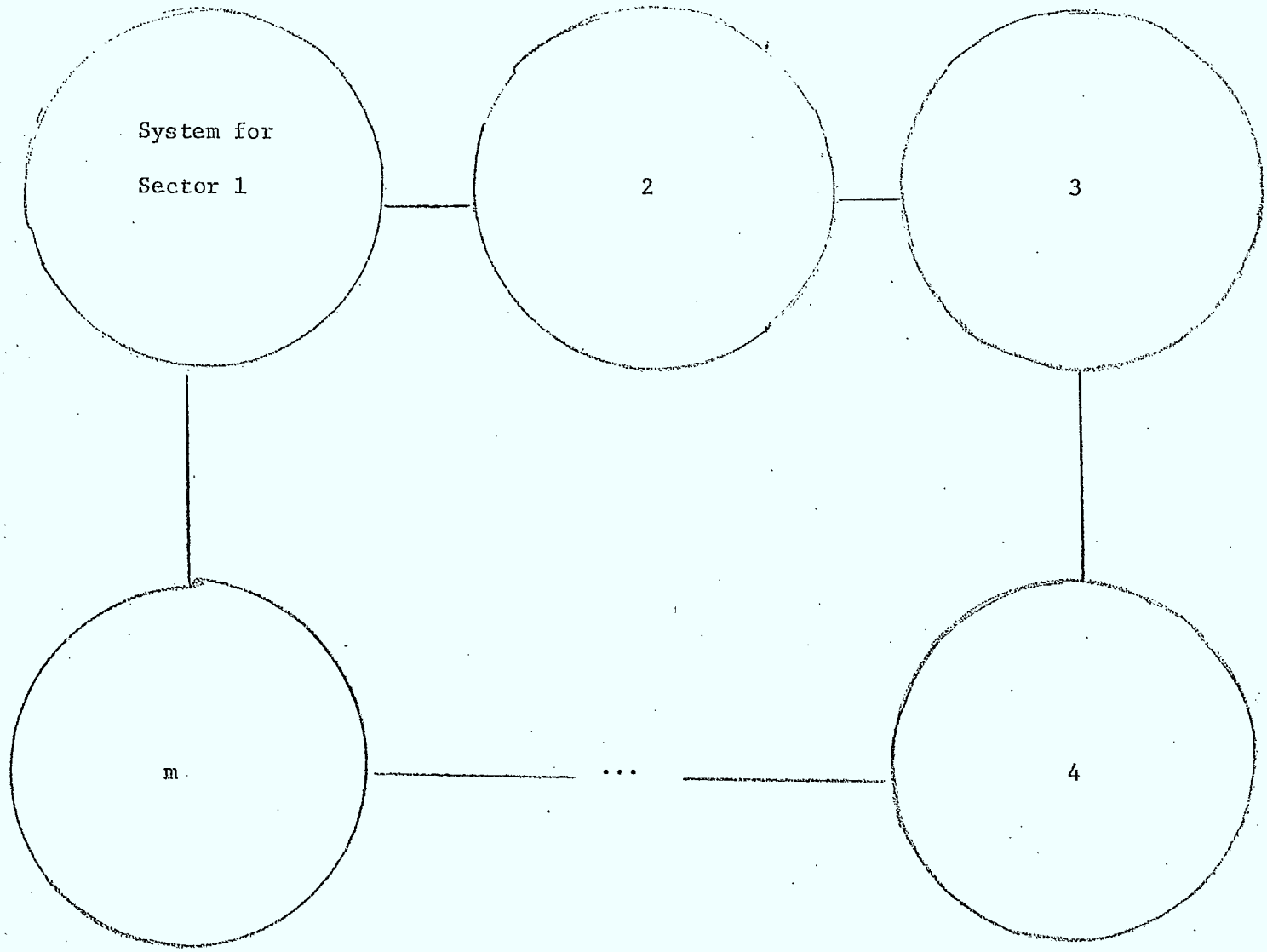


FIGURE 8.1c



9. SUMMARY, CONCLUSIONS, AND FURTHER WORK

Table 1 summarises our review of techniques and tools which have been formulated to prevent, detect, and dalgnose malfunctions and recover from them. A tool for detecting malfunctions bv externally monitoring the behaviour of the system was suggested. Organising a network into triads was shown to be a good way of simplifying the problem of structuring a network to enhance dependabilitv. Section 8 presented a procedure for designing a maintenance system.

As Table 1 illustrates, a wide variety of techniques and tools have been developed over the years to enhance the dependability of computer systems. Audit techniques developed bv Bell Labs are quite widely applicable. Their use in operating systems for commercial computers is strongly recommended. The methods for separating a network into triads to enhance dependabilitv appears quite promising.

While writing this paper, a number of areas requiring work became evident: A. Develop the external monitoring system suggested in Section 5. B. Implement the triad approach experimentallv. C. Build an operating system using audit techniques. D. Develop tools to aid in designing and implementing a maintenance system. E. Create a language (or extend an existing language) to aid in generating audit programs. F. Develop tools and

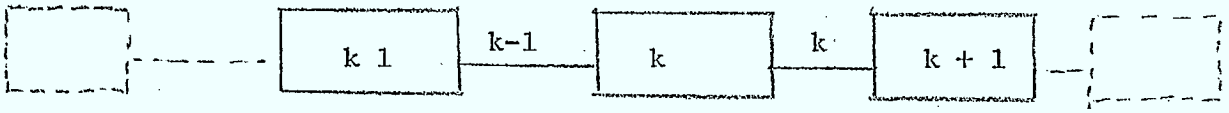
techniques for aiding programmers in testing their programs. A program could be written to find a covering set of paths through a program or set of programs using graph theoretic techniques, for example. 6. Develop a methodology for recovery systems. The approach to developing the combination of software and hardware to permit a system to continue performing its intended functions in the face of errors or faults has been and is heuristic. As a consequence, there is a lack of understanding, from a recovery viewpoint, of the interactions between components, subsystems, and the tradeoffs available between complexity, recoverability, and reliability. A methodology is required to allow the necessary understanding to optimise such systems relative to selected design parameters.

ATTACHEMENT A

Assume that the k^{th} machines function can be performed by either the $(k + 1)^{th}$ as the $(k - 1)^{th}$ machine provided that the data link connecting the k^{th} machine to the takeover machine is also functioning. Also define a system failure as a complete loss of the function of the k^{th} machine.

For a system of this nature it is only necessary to determine the probability of failure of a group of three contiguous machines and a triad.

FIGURE A.1



Let the probability of failure of machine k be p_Δ, and the probability of failure of a data link be p_ℓ

Then the probability pk of loss of function k is the sum of the checked states. Since function k is lost it follows that only those states with k = 1 (meaning failure of the k^{th} machine) need be considered. Then

$$pk = 2p_\Delta^2(1-p_\Delta)\,p_\ell(1-p_\ell) + 2p_\Delta^2(1-p_\Delta)\,p_\ell^2 + 2p_\Delta^3 p_\ell(1-p_\ell)$$

$$+ \; p_\Delta{}^3(1-p_\ell)^2$$
$$+ \; p_\Delta{}^3 p_\ell{}^2$$
$$+ \; p_\Delta(1-p_\Delta)^2 p_\ell{}^2$$

$$= pk = p_\Delta\left[p_\Delta(1-p_\ell)\{2p_\ell + p_\Delta(1-p_\ell)\} + p_\ell{}^2\right]$$

The probability of failure, pk, of the k^{th} function is not affected by any links or machines beyond the complex comprised of the $(k-1)^{st}$, k^{th}, and $(k+1)^{st}$ machine and the $(k-1)^{st}$ and k^{th} data link. This can be demonstrated by including the $(k+1)^{st}$ data link and the $(k-2)^{nd}$ data link as follows:

All failure states for the complex shown in Figure 2 exist as the only failure states for this complex even when the $(k+1)^{st}$ and $(k-2)^{nd}$ data link are added regardless if the states of these links. Since the $(k+2)^{nd}$ and $(k-2)^{nd}$ machines cannot by definition of the system take over the function of the k^{th} machine. So the don't care condition is then pk'

$$\begin{aligned} pk' &= pk(p_\ell{}^2) + 2pk\,(1-p_\ell)p_\ell + pk(1-p_\ell)^2 \\ &= pk(p_\ell{}^2 + 2p_\ell - 2p_\ell{}^2 + 1 - 2p_\ell + p_\ell{}^2) \\ &= pk. \end{aligned}$$

This can be extended link by link and machine by machine.

The probability, ps, that the complex in Figure A.1 will fail to perform its function is $p_s = 3pk$ since the complex fails if any combination of the function k, k-1 or k + 1 cannot be performed. Similarly if there are m machines in the system (Figure 8.1c) the failure probability $p_s = m_{pk}$.

Now what of the system comprised of m machines where the k^{th} function can be performed only by the k^{th} machine? Then the probability of failure of the k^{th} function is $pk = p_\Delta$ and the probability of failure of a complex of 3 adjacent machines (no data lines) is

$$\begin{aligned} ps_o &= p_\Delta^3 + 3p_\Delta(1 - p_\Delta)^2 + 3p_\Delta^2(1-p_\Delta) \\ &= p_\Delta^3 + 3p_\Delta - 6p_\Delta^2 + 3p_\Delta^3 + 3p_\Delta^2 - 3p_\Delta^3 \\ &= p_\Delta^3 - 3p_\Delta^2 + 3p_\Delta = p_\Delta(p_\Delta^2 - 3p_\Delta + 3) \\ &= p_\Delta(p_\Delta^2 + 3(1-p_\Delta)) \end{aligned}$$

Now for the reliability of the system of Figure A.1 to be greater than that of the above system

$$\frac{ps}{ps_o} < 1$$

$$\frac{ps}{ps_o} = \frac{3p\Delta[p\Delta(1-p\ell)\{2p\ell + p\Delta(1-p\ell)\} + p\ell^2]}{p_\Delta[p_\Delta^2 + 3(1 - p_\Delta)]} < 1$$

The quadratic in p_ℓ must be solved to meet inequality. If one remembers that values for p_ℓ or $p_\Delta \leq 1$ are the only admissible ones, then the solution shows inequality satisfied for $p_\ell < 1$.