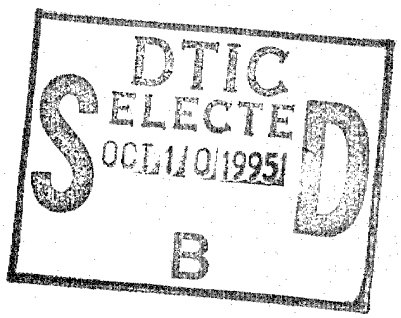
MicroCIM

An Architectural and Owner's Manual

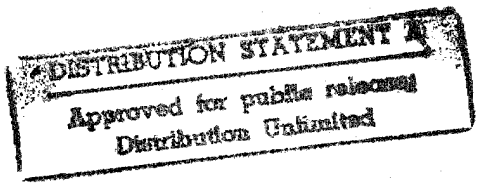
DLA900-87-D-0017
Delivery Order 0002



Dr. J. M. Westall
Dr. A. W. Madison

Bapiraju Buddhavarapu
Sudhir Mookly

Department of Computer Science
Clemson University
Clemson, SC 29634-1906



REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 29, 1992	Final 09/88 to 05/92

4. TITLE AND SUBTITLE

MicroCIM: An architecture Manual; MicroCIM: An Operator's Manual

5. FUNDING NUMBERS

DLA900-87-D-0017 DO 0002

6. AUTHOR(S)

Dr. J.M. Westall, Dr. A.W, Madison, B. Buddhavarapu, S. Moolky

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Clemson Apparel Research
500 Lebanon Road
Pendleton, SC 29670

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Logistics Agency
DLA-PRM
Room 4B195 Cameron Station
Alexandria, VA 22304-6100

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Computer Integrated Manufacturing (CIM) uses computer and network technology to facilitate the interaction of machines and personnel involved in the manufacturing process. The goal of CIM is to provide effective real-time monitoring and management of the entire manufacturing process. CIM is widely viewed as the missing link in providing quick response within the apparel manufacturing industry.

This project supported the development of a distributed computer operating system to be used as a platform for CIM. The MicroCIM research has demonstrated that a true distributed system with multitasking and Local Area Network capability can be built upon computer systems costing significantly less than one thousand dollars per network node.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Advanced Apparel Technology, MicroCIM, Distribution System	80
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL

MicroCIM

An Architecture Manual

Dr. J. M. Westall
Dr. A. W. Madison

Bapiraju Buddhavarapu
Sudhir Moolky

Department of Computer Science
Clemson University
Clemson SC 29634-1906

Table of Contents

1. Introduction to Computer Integrated Manufacturing 4

2. Architecture of microCIM 7

2.1 The Kernel 10

2.1.1 Process Management 11

2.1.1.1 Process Scheduling 12

2.1.1.2 Dispatching 13

2.1.1.3 Process Termination 13

2.1.2 Memory Management 14

2.1.3 Local Inter Process Communication 15

2.1.3.1 Message Passing 16

2.1.3.2 Issues in Message Implementation 16

2.2 The Network Management System 18

2.2.1 The Input and Output Packet Managers 19

2.2.2 Addressing and Naming 22

2.2.3 The Connection Manager 23

2.2.4 Network Application Program Interface 24

St#A, Attn: DLA/MMPRT (Ms Kerlin-767-1415) Telecon, 10 Oct 95 CB

2.3 Dynamic Network Program Loader 26

2.3.1 Sender Process 28

2.3.2 Receiver Process 28

2.3.3 Dynamic Linking 29

2.4 Input/Output Subsystem 30

2.4.1 The Window System 30

2.4.1.1 The Window Server 31

2.4.2 The Keyboard System 32

2.4.2.1 The Keyboard Server 32

3. Performance Evaluation 34

3.1 Performance Measures 34

3.2 The Test Workload 35

3.3 Results 36

3.3.1 Local IPC 36

3.3.2 Network IPC 37

4. Conclusions 44

5. References 45

1. Introduction to Computer Integrated Manufacturing

The manufacturing environment has changed dramatically in the last few years. Two major driving forces behind the changes are the stiff worldwide competition among manufacturing companies and the development and utilization of new technology which includes microprocessors, robots, databases, local area networks, artificial intelligence and others. The manufacturing environment has evolved from manual operation to semiautomatic operation to a high degree of automation making extensive use of computers and other automated equipment.

A *Computer Integrated Manufacturing* (CIM) system is an interconnected system of material processing stations capable of automatically processing a wide variety of part types simultaneously and under computer control. The system is interconnected by a *material transport system* and a *communication network* for integrating all aspects of manufacturing. The communication network not only transfers information, such as programs, between processing stations, but also supports the coordination, monitoring, control, and management of the entire system.

A high degree of automation, integration and flexibility are essential characteristics of an automated system. Flexibility can take a number of forms, including, volume flexibility – the ability to handle changes in the volume of production, routing flexibility – the ability to route parts through the system in a dynamic fashion, and product flexibility – the ability to handle requests for a wide variety of products, including the ability to reconfigure the system.

CIM is a manufacturing strategy whose objectives are improved productivity and reduced production costs. These benefits are obtained through the integration of all computer systems involved in the production process. Components of a CIM system typically include CAD workstations, real–time production monitoring and control systems, order and inventory control systems, and data terminals used by equipment operators. Real–time production monitoring and control systems include sensors, actuators, industrial robots, computerized numerical control, etc. These components function as an integrated unit after they are physically connected by a high speed local area network (LAN) and equipped with software systems designed for distributed operation.

The concept of a *manufacturing cell* is important in automated manufacturing. Manufacturing operations are broken down into cells, with each cell responsible for the manufacturing of a specific part family. The cells are interconnected by a transport system for materials and finished products. Each cell has one or more computerized numerical

control machines or robots that process the parts. The worked pieces are then routed to other manufacturing cells for subsequent processing.

An illustration of a CIM system is shown in figure 1.1. A simple communications network called the *Cell Network* is used by a computer called the *Cell Controller* to monitor and reconfigure individual manufacturing machines. The cell controllers and other computer systems (PCs or mainframes) involved in the manufacturing process are interconnected by a local area network called the *Factory Network.*

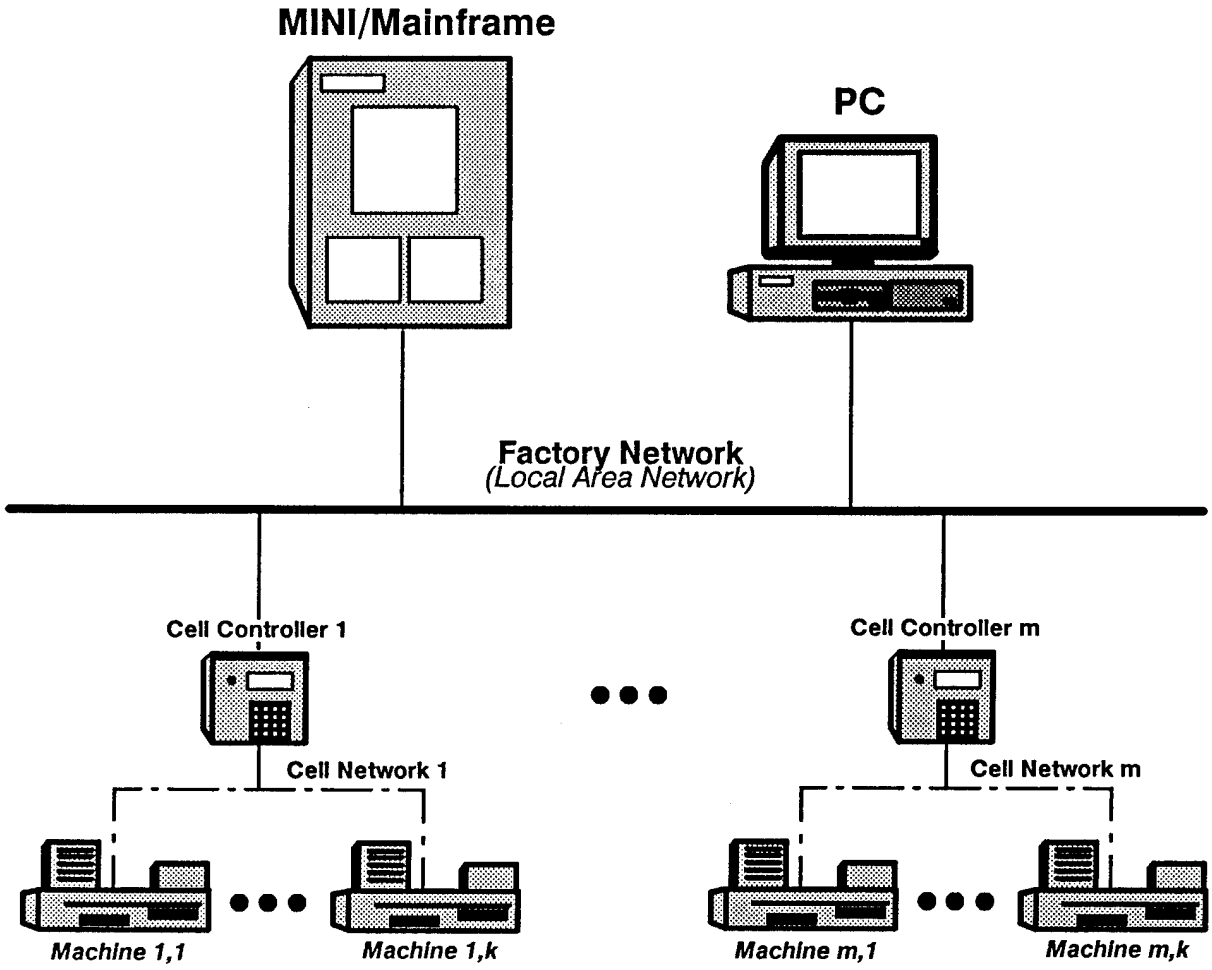


Figure 1.1 A typical CIM system

The shop level control system, consisting of a host computer and the equipment cell controllers, manages the coordination of resources and jobs on the shop floor. The cell controllers are low memory, intelligent systems that require that programs or data for cell devices be transferred from shop level computers.

As an example of the kind of interaction between the cell controllers and the host computer consider the flow of bundled apparel parts in a plant. When a bundle arrives at a workstation, a bar coded ticket attached to the bundle could be scanned by a cell controller computer. The cell controller would in turn transmit a message across the local area network to alert the host computer that work is about to begin on this bundle. The host computer would then access a file of data and extract information concerning the size and style of the problem. The size and style information would then be used to access a computer program stored in a database accessible to the host computer. This program, which would be tailored to the needs of an advanced technology sewing device, would be transmitted across the LAN to the cell controller. This program would then be placed into execution on the cell controller and would direct the movement of the needle across the fabric.

The benefits of integrated automated manufacturing include:

- Increased productivity and reduction of design to production cycle time.
- Reduction in reconfiguration time when product requirements change.
- Elimination of human error in machine setup and reduction of machine setup time.
- Reduced raw material and finished good inventories.
- Improved responsiveness in order–driven production, and
- Effective usage of equipment.

Despite these potential benefits, the development of an open architecture for CIM has been a slow process. Obstacles have included both a proliferation of competing standards for computer networks and a lack of standard interfaces and protocols for machine control functions. In the absence of accepted standards computer manufacturers have developed a variety of proprietary, competing networked systems described as CIM platforms.

A demonstration of the viability of large–scale, open CIM architecture required a multi–year commitment by corporate giants GM and Boeing. Their effort resulted in the development of the MAP (Manufacturing Automation Protocol) and TOP (Technical and Office Protocol) architectures. MAP/TOP based systems have now established their value in automobile and aircraft manufacturing and are presently making inroads into other heavy industries.

2. Architecture of MicroCIM

The proven benefits of CIM are directly applicable to apparel manufacturing and other light industries. Unfortunately, the cost of components is a formidable barrier to the adoption of the MAP/TOP architecture. Cell controller nodes in heavy industry CIM networks are computers capable of running complex operating systems such as UNIX or OS/2. Memory requirements for such systems are now in the range of six to eight megabytes. The MAP protocol requires the IEEE 802.4 token bus LAN. IEEE 802.4 attachment boards have historically been several times as costly as their more common ethernet or token ring counterparts. When the cost of system software is included, each reasonably configured MAP node can cost significantly more than $10,000.

MicroCIM is a distributed computer operating system, designed and developed at Clemson University. The objective is to provide CIM functionality to small apparel manufacturers in a cost effective package in a way that provides a reasonable migration path to a full MAP/TOP system.

The operating system is designed to be small, portable, and support multitasking and distributed processing. It is written in the C programming language and is developed on Intel 80x86 based personal computer systems.

A typical CIM system software configuration is illustrated in the figure below.

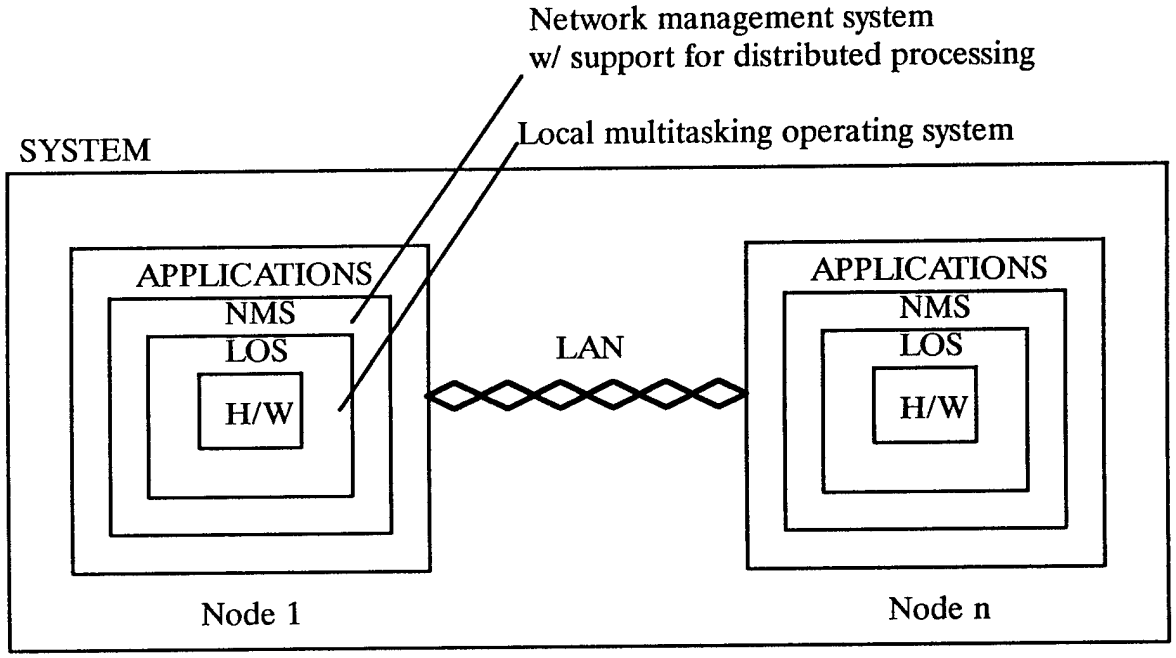


Figure 2.1. A CIM system software configuration

Each node in the network can either be a PC or a cell controller. The operating system on each node functions independently of the operating system running on any other node. Local resources, such as memory and CPU, are owned and managed by the local operating system. Global resources, such as secondary storage and service names, are obtained by request from a remote site with the intervention of the network control and management components.

MicroCIM is a real-time operating system. Unlike general purpose operating systems, it is not a development system. Program development is separated from the operational system itself. Applications have to be first developed on a development system before they can be used on the operational system.

The operating system is designed to be portable across different hardware platforms with minimal changes to the system software. Its design is based on a modular, layered concept. The layered architecture of MicroCIM is illustrated in figure 2.2. The kernel layer provides a well defined set of services to all the layers above it. The network management layer makes use of the kernel services to provide communication facilities to the application layer. The internal structures, mechanisms and algorithms used in one layer are not visible to the layers above it. Thus, changes effected in one layer have minimal impact on the layers above it so long as the interface between the layers is not changed.

The kernel of the operating system provides facilities for process management, inter process communication and memory management within a site. A preemptive, multi-tasking scheduler permits multiple application programs to run concurrently on each system in the network. The use of multi-tasking greatly simplifies the design of CIM application programs since each of the application programs can concern itself with performing only a single task. Operating system and network management software use a technique known as message passing to communicate and synchronize their activities within a given system.

An ARCNET LAN is currently used at the physical layer. ARCNET provides data rates of 2.5 Mbits/second, an order of magnitude greater than RS-422 and RS-485 networks, for about $50 per network node. Migration to faster, more expensive LANs such as Ethernet and Token Ring will require additional device drivers.

The network management system provides the interface between application programs and the ARCNET LAN. Development of distributed applications is facilitated by a high level application program interface (API) to network services. Network services include both connectionless(datagram) and connection oriented management.

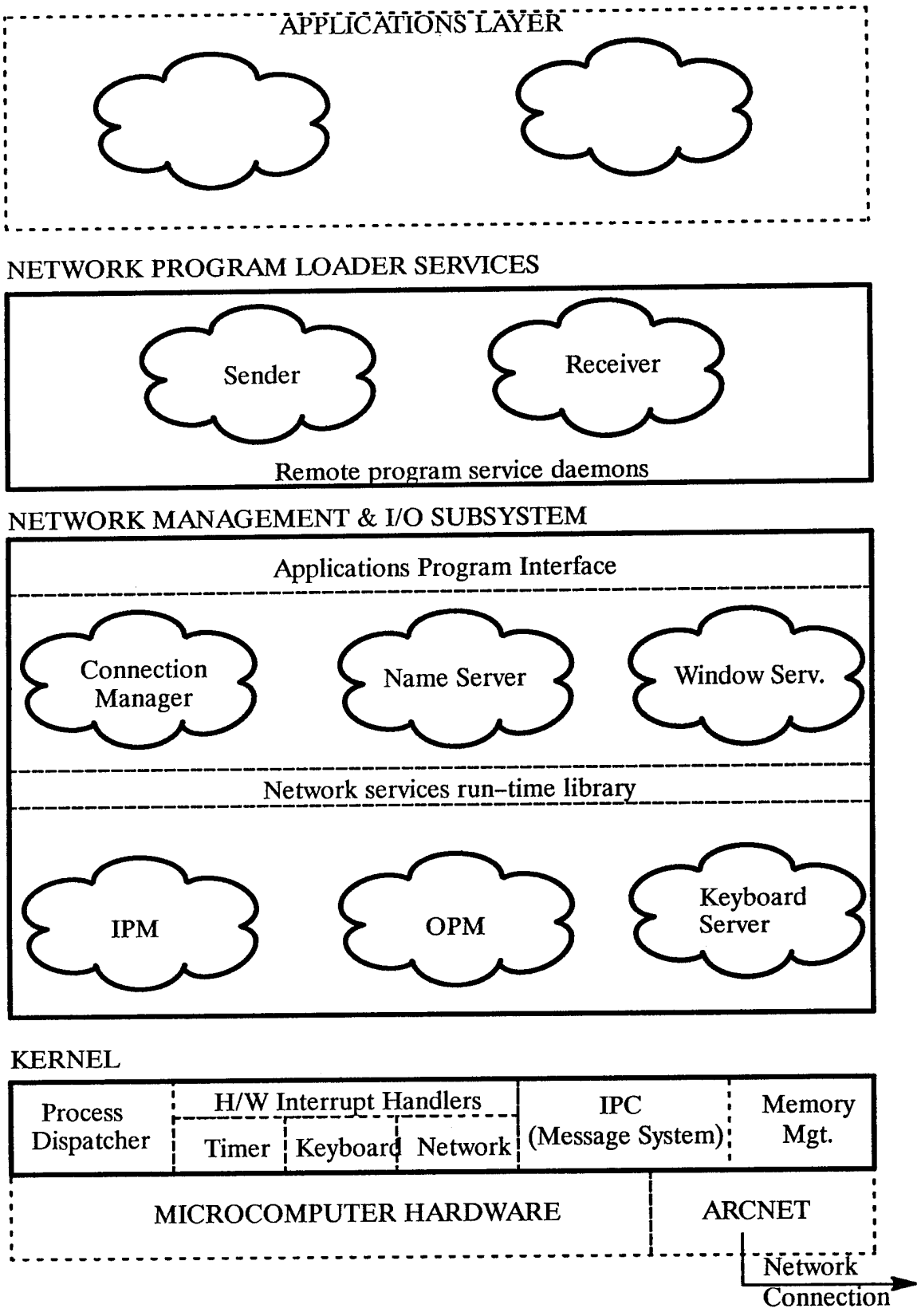


Figure 2.2 Architecture of MicroCIM

An application can associate logical names with the network addresses it uses in conducting communications. A nameserver program transparently provides services analogous to a telephone directory assistance. This permits one application to send data to another using the target application's "name" rather than address. When network messages are received, the recipient can likewise determine the "name" of the sender when required.

Network management is simplified through remote program loading facilities. Diskless workstations and cell–controllers can be used throughout the manufacturing facility. The operating system and all application programs can reside on one or more host personal computers. When the network is started, the operating system is down loaded to all diskless units. After the operating system is running, applications can be started. Any application can request that another application be started as a new task on any node in the network. The requestor need only specify the name of the application to be started as a new task on any node in the network. the requestor need only specify the name of the application to be started, the system its code resides on, and the system it is to be run on. Parameters passed to the application to be started enable it to send data or receive data from the program that initiated it.

The ultimate goal of this effort is to demonstrate a cost effective computer system that will simultaneously support real–time shop–floor data collection and analysis, automatic machine configuration, and factory control at a reasonable cost.

A detailed design of the various components of the operating system are described in the following sections.

2.1 THE KERNEL

The kernel is the lowest layer of the operating system directly above the hardware. It provides facilities for process management, inter process communication and synchronization, memory management and interrupt management.

Operating system services are provided through a *system call* mechanism. Since system calls access kernel data structures they are implemented as critical sections. This is accomplished by executing all kernel functions within a software interrupt handler. The software interrupt handler assures that hardware interrupts are disabled while it is servicing requests. A SYSTEM_INTERRUPT vector is reserved for kernel system calls. When a system call is made, parameters to the call including indication of the type of call are placed in registers and a software interrupt to the system interrupt is generated. The system interrupt service routine performs a switch on the type of the system call and calls an internal function with the parameters that were passed to it.

The details of the various components of the kernel are presented in the following sections.

2.1.1 Process Management

Process management deals with the operating systems policies and mechanisms for sharing the CPU resources among user and system processes. A process is typically viewed as a program in execution. Process control and management is performed through the use of Process Control Blocks (PCB). The PCB contains descriptive information about the process such as: process identification, process type (user/system), process priority, state information, stack size and location, resource requirements and state, linked list pointers, etc.

A process can exist in any one of four states: running, ready, blocked or free. A process is considered to be in free state before it is brought into the system and after it has terminated. All active processes in the system are in one of running, ready or blocked states. A process is said to be running when it is using the CPU and executing instructions. The ready state refers to the state in which a process is prepared to execute if given control of the processor. A process is in this state when it has all the resources it requires except the CPU. A process is said to be blocked when it is active, but is waiting to acquire some resource. Figure 2.3 shows the state transitions that a process can pass through.

Processes are of two types: system and user. A system process is one that is supplied by the operating system to aid in the controlled sharing of system resources. For example, the window server is a system process that controls access to the video screen. All system processes in MicroCIM are created at system initialization time. All application programs run as user processes.

The creation of a process involves the construction of a PCB with appropriate initializations. The information required to create a process is the process' priority, stack size, number of mailbox handles required, type (system/user), and the starting address of the code that the process is required to execute. A request for process creation with this information will initiate the following sequence of events:

- Acquire a process control block. (This is acquired from the array of free PCBs. The index of this PCB forms the process id to this process and uniquely defines the process in the system.)
- The process type and process priority are recorded in the PCB.
- The requested number of mailbox handles are allocated.

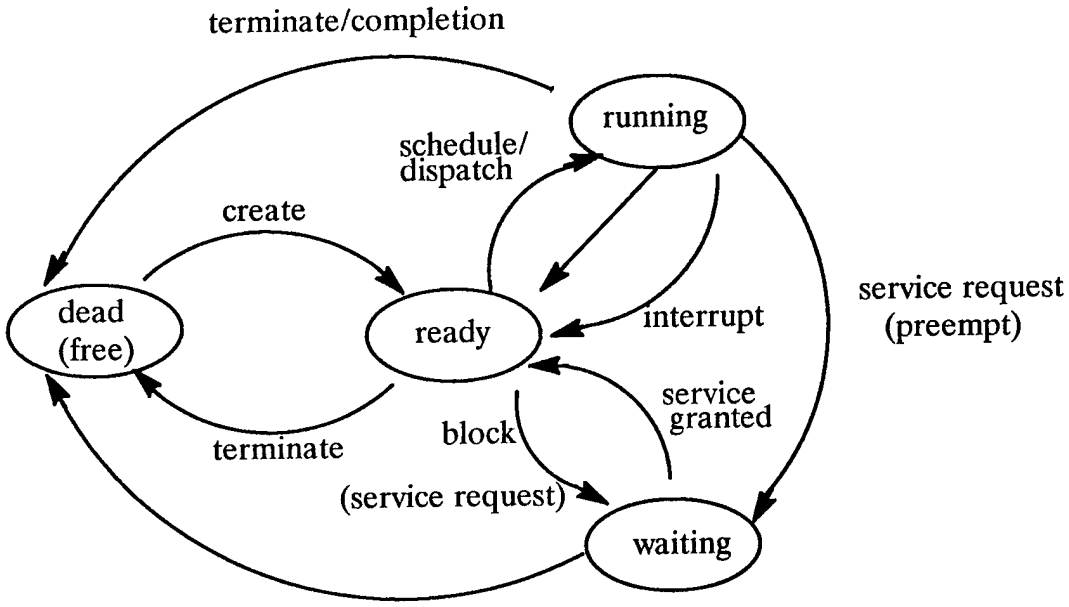


Figure 2.3 Process state transition diagram

- The requested stack area is allocated.
- The process state is initialized and recorded in the PCB. This reflects the initial values of all registers and the address at which the process is to start executing.

To introduce a process into the system, its PCB is inserted in the ready queue of processes. This queue represents all processes that are ready to execute. All processes in the ready queue are linked together to form a chain of active ready processes.

A process may get blocked waiting for an event to occur (such as receiving a message or waiting for a quantum of time to elapse). In such a case, its PCB is removed from the read queue and inserted in the appropriate suspend queue.

2.1.1.1 Process Scheduling

The process of selecting a time in the future for initiating the execution of a process is called scheduling. MicroCIM employs a **preemptive priority based time sliced** scheduling technique.

The ready queue is maintained in order of decreasing priority. The first of the highest priority processes will be selected next for execution. All ready processes with the same priority are serviced in FIFO order. Thus a low priority process cannot execute as long there is another ready process with a higher priority.

Each process may run uninterrupted for at most one time slice. The length of the time slice, measured in number of clock ticks, is a fixed at system initialization. A process can be preempted before its time slice expires if another process with a higher

priority is made ready. Preemptive scheduling is well suited to a real-time environment because it is characterized by a predictable response to high priority processes. This may however cause starvation of low priority processes. Hence choosing the priority of a process is an important decision.

2.1.1.2 Dispatching

Dispatching causes a scheduled process to gain access to the CPU. This is accomplished through the use of a dispatch interrupt service routine. A dispatch interrupt can occur either if a process exhausts its timeslice or if a preemption takes place.The dispatcher accesses the PCB of the process that is interrupted, saves the state of the process in the PCB and inserts it in the ready queue according to the scheduling policy. It then accesses the PCB of the process scheduled to run next, and loads the physical registers with its saved values to effect a start-up of the process.

Timer interrupt management is a critical part of process dispatching. When a process is dispatched it is given a full quantum of timeslice. Every time the timer interrupt occurs, the timeslice is decremented. A dispatch interrupt is generated when the timeslice is reduced to zero.

The timer interrupt service routine also services the delay queue. The delay queue is a list of process that are suspended for a specified period of time. It is maintained in increasing order of relative delay. Relative delay of a process is the number of clock-ticks to wait after the delay of that process's predecessor expires. Thus the absolute delay of a process is the sum of the relative delays of all processes preceding it in the delay queue. For the first process in the delay queue, its relative delay is the same as its absolute delay. Each time the clock interrupt occurs, the timer isr decrements the relative delay of the first process in the delay queue, and schedules it if its delay is reduced to zero.

2.1.1.3 Process Termination

A terminating process generates a terminate interrupt to relinquish all the resources it has acquired. The kernel dequeues the process's pcb from all the system queues, destroys all its mailboxes, deallocates the stack space, frees any network resources that are held by the process and invalidates its process id.

2.1.2 Memory Management

Memory management in MicroCIM is designed to be fast and simple. All processes reside completely in main memory itself; thereby making swapping unnecessary. The kernel allocates memory for essential data structures for each process when it is created and brought into the system. This memory is deallocated by the kernel when the process terminates. Contiguous chunks of memory are allocated dynamically as and when requested by the processes. A process is responsible for freeing all memory it requests while it is an active process. Otherwise, that memory will remain unused.

The memory management system uses a *first-fit* algorithm to manage memory. A contiguous pool of memory is allocated when the system is first initialized. Blocks are allocated from this pool upon request, and free blocks are maintained in a linked list. Each free block contains two fields at its head: the size of the free block in bytes and a pointer to the next free block in the list. The size field is adjusted to reflect the bytes consumed by the two fields at the head of the block. The list is maintained in ascending order by address.

When a memory request is made, the kernel searches the list for the first block that contains enough space to accommodate the request. If the request consumes all the memory of the block, the block is removed from the list, and its preceding block is linked to its succeeding block. Otherwise, a new block is created at the end of the requested chunk of memory and its size field updated to show the remaining number of bytes from the parent block.

When a block is returned to the free pool, the system checks to see if the block immediately preceding it or following it is free, and combines the blocks if possible. Three cases are possible when freeing a block:

1. If it can be combined with the block immediately preceding it, the change in the size of the block is reflected in the size field of the preceding block. In this case the list pointers remain the same.

2. If it can be combined with the block following it, the change in the size field is reflected in the freed block. The list pointer of the freed block now points to the block that was previously pointed to by its following block.

3. If it can be combined with both the preceding and following blocks the change is reflected in the size field of the preceding block. The list pointer of the preceding block now points to the block previously pointed to by the following block.

The system makes no attempt to recover from external fragmentation if a memory request fails.

2.1.3 Local Inter Process Communication

Interprocess synchronization and communication (IPC) are necessary to support concurrent process execution. IPC mechanisms allow arbitrary processes to exchange data and synchronize execution. Communication is required to allow cooperating processes to exchange data. Synchronization is necessary to preserve system integrity and to prevent timing problems resulting from concurrent access to shared resources by multiple processes.

All processes executing in a multitasking environment compete with other processes for system resources, such as CPU and memory. However, the handling of competition among process in a multitasking system depends largely on the nature of the specific process. System–defined processes tend to be self sufficient and separate from other processes, while programmer–defined processes that collectively constitute a single logical application often cooperate and share common resources. The operating system manages the use of system resources on behalf of such user processes in a manner that is often transparent to the related processes, and no explicit synchronization statements need to be provided in their source code for that purpose.

There are two facets to interprocess communications in a distributed multitasking environment:

- local IPC – between processes on the same node, and
- network IPC – between processes across two different nodes.

Local IPC is effected through a set of communication primitives that are provided by the kernel of the operating system. Network IPC requires the support of a LAN, a set of daemon processes and interrupt handlers in addition to the communication primitives of the operating system. Application programs communicate by the use of an application program interface (API) that keeps the details of the network transparent to the user.

A variety of choices were available in selecting a method for interprocess communications and synchronization. Message passing was selected as the most appropriate method for this project. Messages are a simple mechanism suitable for IPC in both centralized as well as distributed environments. Sending and receiving messages is a standard form of inter–site communication in computer networks, making it very attractive to augment this facility in the operating system. For this reason, messages are the most popular IPC mechanism in distributed operating systems.

2.1.3.1 Message Passing

A message is a collection of information that may be exchanged between a sending and a receiving process. Messages may contain control/status information or data. Messages represent requests or responses for service in the system. All actions are controlled via this message-passing scheme.

2.1.3.2 Issues in Message Implementation

Message passing in MicroCIM is an indirect message communication method, where messages are sent to and received from special repositories called *mailboxes.* A mailbox is a named data structure that can be thought of as a queue of messages. Processes can either *send* a message to a mailbox (enqueue a message) or *receive* a message from a mailbox (dequeue a message). This form of message communication requires additional primitives for *creating* and *deleting* mailboxes.

Message operations can be of a conditional or unconditional class. In a unconditional send/receive, the process will block until the operation can be carried out – regardless of how long the suspension may last. A conditional send/receive may be used to insure that the process blocks for no longer than a given amount of time. The maximum time for which the process may be suspended is specified in the send/receive call. If a maximum time of 0 is specified, then the send/receive will not block at all and will either be carried out immediately or a failure code will be returned. The time units for block time is based on the frequency of occurrence of the timer interrupt on a given hardware.

Message exchange between two processes is effected by copying the whole message from the sender's address space into the receiver's space.

A mailbox must be created by the *mcreate()* system call before it can be used. The name of the mailbox, the number of messages that it can buffer and the maximum length of a message it can accept are specified at mailbox creation time. This permits greater flexibility in the use of the mailbox and minimizes memory requirements. Upon successful completion, the create system call returns a mailbox *handle* which is a pointer to the mailbox object. This handle is then used in all subsequent calls to access the mailbox. The process that creates a mailbox becomes the "owner" of the mailbox and is the only process that can later destroy the mailbox.

A mailbox may be created in any one of three modes: M_SEND, M_RECEIVE or M_SENDRECEIVE. The mode indicates the operations that other processes can use on the mailbox. The owner process can always send/receive to/from the mailbox. A receive operation on a mailbox that has been created in the M_SEND mode or a send operation

on a mailbox that has been created in the M_RECEIVE mode, by a non-owner process will fail.

Buffer space is pre-allocated to a mailbox when it is created, instead of allocating a message buffer when a message is sent. The latter technique is a more general implementation of buffer space management and does make better utilization of memory. However, the overhead involved in allocating and de-allocating buffer space on every message send/receive is unacceptable in this environment.

A process can attach itself (gain access) to a mailbox by using the *mopen()* system call, specifying the name of the mailbox. A mailbox must be created before it can be opened. A non-owner process that tries to open a mailbox before the owner process has created it will receive a failure code on the open. The process must then retry until the owner process has had a chance to create the mailbox.

A process that opens a mailbox can detach itself from it by the *mclose()* system call. A process that creates a mailbox can destroy the mailbox by the *mdestroy()* system call. mdestroy() de-allocates the mailbox's buffer space. If any process(es) is blocked on the mailbox waiting to receive a message or send a message, the send/receive operation is terminated and a failure condition is returned to the sending/receiving process via the return code of the send/receive. Any messages left in the mailbox when it is destroyed are lost. A process that tries to send/receive to/from a mailbox after it is destroyed will receive a failure code from the operation.

Mailbox based communication is very versatile in that it can provide one-to-one, one-to-many, and many-to-many mappings between sending and receiving processes. One-to-one mappings provide a private communication channel between two processes. One-to-many mappings provide for a single sender with multiple receivers. Many-to-one mapping is important for server processes.

In order to use a mailbox for interprocess synchronization a mailbox must be allocated for each type of resource whose use must be regulated. A number of messages equal to the total number of resources available will be sent to the mailbox initially. Each process wanting to use a resource of this type must first execute a blocking receive on the mailbox corresponding to the resource type. When the process is finished with the resource, it will send a message back to the mailbox from which it received it. Critical regions of a code can be protected by conceptualizing them as types of resources in which there is only one resource of a type.

2.2 THE NETWORK MANAGEMENT SYSTEM

The job of network management, as a part of a distributed operating system, is to provide intra- and intersite communications among consenting processes. Network IPC in MicroCIM is implemented by a set of interrupt service routines and daemon processes that provide low level software interface with the physical network.

Fig 2. gives an overview of the network architecture developed for MicroCIM. The network management system consists of a device driver (interrupt service routines), an input packet manager, an output packet manager, a name server, a communications manager and an application run-time library.

An ARCNET LAN is currently used as the physical layer. ARCNET provides data rates of 2.5 Mbits/second. ARCNET uses a contention-free token passing access scheme. A token is passed through every station on the network, whether it needs to transmit or not, giving each station an equivalent share of the network time. A station can only transmit a message when it has the token. ARCNET provides direct acknowledgement capability by which stations can indicated immediately if they can accept data and acknowledge when they have received data. This ensures efficient and successful first time transmissions. ARCNET can be wired in a star or bus topology and can use a combination of different types of cabling.

The interrupt service routine (ISR) provides the interface between the ARCNET hardware buffers and the incoming packet manager (IPC) and outgoing packet manager (OPC). The device driver services interrupts generated by the ARCNET card. It signals either the IPC or the OPC of the occurrence of an interrupt. The IPC receives incoming messages and queues them for receipt by an application process. The OPC receives requests from application processes for the transmission of messages to other nodes, and carries out the physical interaction with the ARCNET adapter to accomplish transmission.

The name server, network program loader daemons and application processes interact with the network via the IPC and OPC. The function of the name server is to allow a node to obtain access to a resource on the same or a different node. The connection manager allows processes on different nodes to establish virtual communication circuits for the exchange of transactions and information. The interface to these systems is provided by a run-time library.

Logical/physical interprocess linking is accomplished through the use of *ports*. The port identifier, an integer, is an index into a port table maintained by the system. Net-

work resources that are allocated to processes are associated with specific ports owned by the requesting processes.

A detailed description of each of the components of the network management system is given below.

2.2.1 The Input and Output Packet Managers

The IPM is responsible for transferring incoming packets from the ARCNET adapter to the application processes.

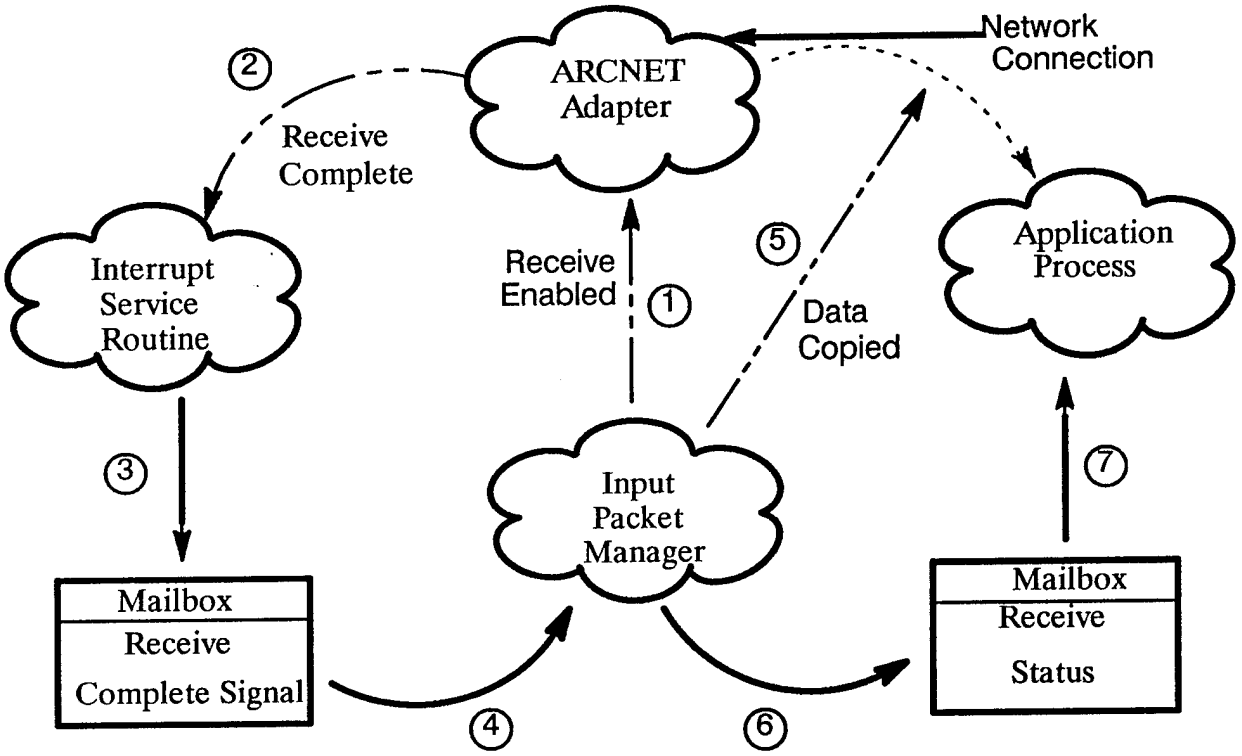


Figure 2.4 Network Input Message Processing

The sequence of operations involved in receiving a message (as illustrated in figure 2.4) are:

1. The IPC ensures that there is a buffer available on the adapter memory to receive a packet and enables a receive operation into a specified page of adapter memory. It then performs a blocking receive on a mailbox and waits for a receive-complete signal from the LAN interrupt handler.

2. The ARCNET adapter generates an interrupt when a packet is successfully received.

2. The OPM reads the request to transmit from the output-request mailbox and constructs an internal ARCNET packet on an adapter buffer page. A sequence number is tagged on to the message. This is done to ensure that the IPM at the receiving end can identify duplicate messages. As mentioned earlier, a race condition exists in which a message can get transmitted more than once(this occurs if a sender times out and retransmits even though the original transmission was transmitted correctly). The head of the packet contains the destination node and port addresses, followed by the message buffer.
3. The message to be transmitted is copied from the application process's address space onto the adapter buffer page.
4. The OPM enables data transmission that will initiate sending the data from the specified buffer page.

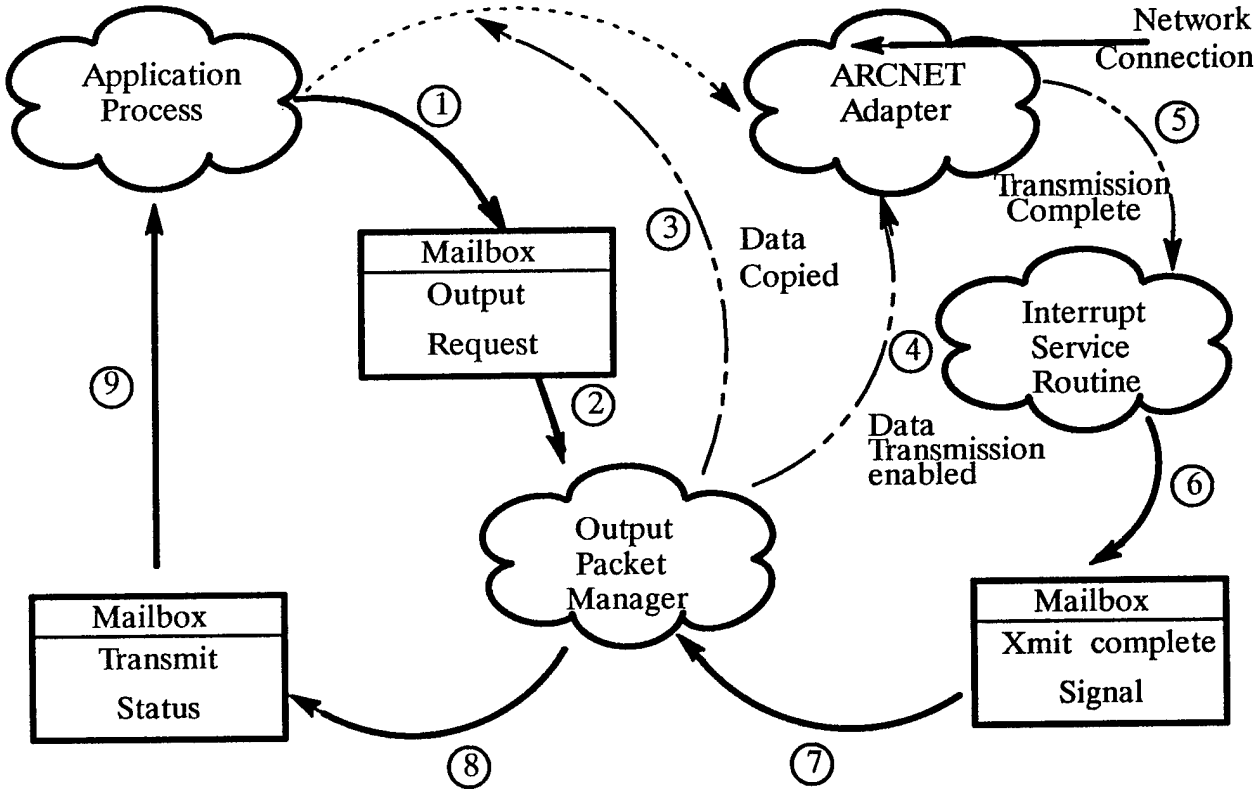


Figure 2.5 Network Output Message Processing

5. The ARCNET adapter generates an interrupt when transmission is complete.
6. The LAN interrupt service routine sends a signal to the transmit-complete mailbox.

7. The OPM receives indication that the transmission is complete by reading the transmit-complete mailbox.
8. The status of the transmit operation is returned to the application process on a transmit-status mailbox associated with the port that requested the transmit operation.
9. The application process receives the status of the transmit operation from the transmit-status mailbox.

2.2.2 Addressing and Naming

Source and destination addressing is specified via a (logical port, physical node address) ordered pair. An address of this type is equivalent to a Service Access Point (SAP) in ISO terminology or a Socket in UNIX networking terminology.

Network communication between two processes requires knowledge of the source and destination SAP addresses. For example, consider a situation where a user process wants to communicate with a server process. To establish communication, the server process must have a fixed SAP address which must be known to the application process in advance. While fixed SAP addresses might work for a few key server processes that might never change, in general, application processes often want to talk to other application processes that only exist for a short time and cannot have a SAP address that is known in advance. Hence, application programs do not typically use actual SAP addresses. Instead, logical service names are defined by the applications themselves and resolved by session layer software. This translation of logical to physical addresses is hidden from the application processes. In this way application programs are kept free of physical addressing dependencies.

The dynamic association of logical names to SAP addresses and the function of resolving logical names to SAP addresses is handled by the *name server* process. The name server maintains a database of network-wide logical names and their allocated SAP addresses. It listens for requests on a fixed known SAP address and services three types of requests: *register*, *resolve* and *cancel*.

When a new service is created, the owner process must register the service with the name server, by sending a *register* request, giving both its service name and the address of its SAP. The name server records this information in its database, so that when queries come in later, it will know the answers.To find the SAP address corresponding to a given service name, an application process sends a *resolve* request message to the name server specifying the service name, and the names server sends back the corresponding

SAP address. When an application process decides to terminate its service it must invalidate its logical service name. This is done by sending a *cancel* request message to the name server specifying the name of the service to be cancelled.

2.2.3 The Connection Manager

The connection manager(CM) is a system process which handles requests to create and terminate connections. Application programs make these requests indirectly through the network application program interface. While processing such requests the CM manipulates connection table entries as well as communicates with the CM on the target node.

In MicroCIM connections are simplex. Therefore the functions performed at the two ends are distinct. The process requesting a connection is called the SENDING PROCESS and it is at the SENDING END of the connection. The process to whom the connection is to be made is called the TARGET PROCESS and it is at the RECEIVING END of the connection. We also use SENDING NODE, SENDING PORT, TARGET NODE and TARGET PORT to identify the two machines involved in the connection.

OPEN request : First the CM allots an entry from the connection table. Then the CM identifies the target node.

If the target node is a remote node, the CM sends an OPEN request to the CM on the target node. The target CM makes sure that the target port is indeed in use. If not it sends an error message back to the sending CM. It then allots an entry in the connection table and enters control information. Finally it sends an OK back to the sender CM. The sender CM then returns the connection id to the process which made the OPEN request.

If the sender and target reside on the same node, then the CM makes sure that the target port is in use. It allocates another entry from the connection table, enters control information and returns the connection id to the requesting process.

CLOSE request : First the CM closes the connection table entry associated with the connection. Then it identifies the target node.

If the target node is remote, the CM sends a CLOSE request to the CM on the target node. The target CM closes the connection table entry and returns OK to the sender CM. The sender CM finally returns an OK to the requesting process.

If the connection is local, then the CM closes both the connection table entries and returns OK to the requesting process.

Note that the CM is not involved in actual virtual circuit message passing

from the sender process to the target process. However the connection table entries created by the CM are consulted at strategic points to ensure reliable message transfers.

2.2.4 Network Application Program Interface

The network layer provides facilities by which packets of information can be routed through the network. Two kinds of services are provided by the network API.

Independent packets of connectionless organization are called *datagrams* in networking terminology. A datagram facility is characterized by the fact that each packet sent is routed independently of its predecessors. Successive packets may follow different routes. Routing details are handled by the ARCNET adapter.

A virtual circuit is a service which, unlike the datagram facility, guarantees delivery of the messages. It also guarantees the correct ordering of packets sent. In other words, packets sent in a particular order will be received in exactly the same order. The datagram, virtual circuit and naming services provided by the network management system are accessed by application programs through the application program interface (API). The API provides facilities for opening and closing a SAP, sending and receiving datagrams, and finding SAP addresses associated with logical service names. The API interface remains the same independent of whether the communication is between local or remote processes.A detailed description of these services is given below.

Datagram Services

1. dg_open: dg_open must be called specifying a logical name for the service before any datagram based services are requested. Its function is to associate a SAP with a process. It opens a port, which combined with the physical node address comprise the SAP address. The (logical name, SAP address) pair is registered with the name server. It also creates a dynamic buffer pool which can be used to buffer incoming datagrams. A process that does not define a buffer pool but attempts to repeatedly receive messages suffers an exposure to lost datagrams since a datagram that arrives between the completion of a receive and the issue of the succeeding receive will be lost. Use of a buffer pool greatly reduces this exposure. No messages will be lost unless the entire buffer pool is full. The maximum size of the message to buffer and the maximum number of messages to buffer are specified in the dg_open call. Calling dg_open indicates a willingness to receive datagrams from any other SAP in the network. Before a datagram may be successfully sent, the recipient must have executed dg_open.

2. dg_findid: dg_findid is used to obtain the SAP address associated with a particular logical service name. It is used by a process that desires to send one or more datagrams to a remote service. This function communicates with the name server to resolve the logical service name into a SAP address.
3. dg_send: dg_send is used to send a datagram to a remote (or local) SAP. It builds a standard network layer header and communicates with the output packet manager to transmit the datagram.
4. dg_recv: dg_recv is used to express willingness to receive a datagram. If a datagram has already arrived and been placed in the buffer pool, it is immediately returned. Otherwise, the calling process is suspended until a datagram arrives.
5. dg_recvc: dg_recvc is a conditional receive. The maximum time that the calling process can suspend waiting for the receipt of a datagram is specified in the dg_recvc call. If no datagram has been received after the interval specified has expired, control is returned to the caller. dg_recv should not be used to poll for the arrival of a datagram as this will unnecessarily degrade the performance of the entire system. The multitasking facilities of the operating system can be used to create a process dedicated to handling the receipt of datagrams. the dedicated process can then communicate with a control process using the message passing facility.
6. dg_close: dg_close is used to close a SAP and free the associated resources. All SAPs that are not closed by a process are closed upon its termination. If the task has an associated dynamic buffer pool, it is freed. A message is sent to the name server requesting that the entry associated with this SAP be cancelled. The corresponding port table entry is freed and associated mailboxes are closed.

Virtual Circuit Services

7. vc_open: vc_open is used when a process wants to open a virtual circuit connection to a target process. This call is built on top of dg_open and so the parameters are the same as those required in dg_open. The extra parameter needed is the SAP address of the target process. This is obtained by calling dg_findid. Vc_open has to be called first before any process attempts to send messages on a connection. Since the connection is simplex, the target process is unaware of the whole procedure. On each node there is a system process called the connection manager. The connection manager(CM) is in charge of making and breaking connections. Vc_open first calls dg_open to open a port. It then sends a request to the

CM to open a connection to the target process. Finally it creates an acknowledgement mailbox in the port table entry. The IPM writes any incoming acknowledgements into this mailbox.

8. vc_send: vc_send is used to send messages on a virtual circuit which was opened by vc_open. When the CM opens a connection to the target process it creates a connection table entry. In this connection table entry is a sequence number to be applied to the current message on the connection. Note that this is a higher level sequence strategy than the sequencing done by OPM and that it is not applied to datagram messages. The sequence number is applied to the current message along with other control information and the message is sent to the target process. A field in the control information is used to identify the fact that this is a virtual circuit message as opposed to a datagram message. Virtual circuit messages can be either data or acknowledgements and this is identified using the same field. Then vc_send waits for an acknowledgement on the acknowledgement mailbox with a finite timeout. If an ack is received then it's status is checked otherwise the current message is retransmitted(stop and wait protocol).

   If the status is good then the sequence number of the ack is checked with the sequence number of the message sent. Since the target process acks the sequence number received(as opposed to sequence number expected next) both these numbers should match. If the ack number matches the current message number then the sequence number in the connection table entry is updated for the next message and success is returned. If the ack number matches the previous message then the current message is retransmitted again. Otherwise an error status is returned to the caller.

   If the status is bad then there is a fatal problem and the connection cannot be used anymore. An error status is returned to the caller and it is the caller's responsibility to close the connection immediately using vc_close.

9. vc_close: vc_close is used to close a connection opened previously by vc_open. Vc_open first sends a request to the connection manager to close the connection. Then it destroys the acknowledgement mailbox following which it closes the port entry using dg_close.

2.3 DYNAMIC NETWORK PROGRAM LOADER

A CIM network consists of a number of nodes that run MicroCIM and are inter-connected on a communications network. One of the nodes functions as a controlling

host and the remaining nodes take on the functions of a cell controller. The design of a low-cost cell controller that can support a multitasking operating system, a LAN, an efficient machine interface and also provide for human operator interaction is presently under consideration. These cell controllers would be diskless workstations with about one megabyte of memory.

Remote program loading facilities permit the use of diskless manufacturing cell controllers and simplify network management. This facility is provided by a *network program loader* incorporated in MicroCIM. The network program loader enables a process on one node to establish a session with a process on another node for the purpose of interchanging data. When the network is started, the operating system is started on each node in the network. When the OS is up and running, multiple applications can be started on any node. Any application program can request that another application be started as a new task on any node in the network. Parameters passed to the new task enable it to send data to or receive data from the program that initiated it. Figure 2.6 illustrates the use of the network program loader.

The network loader is implemented by two daemon processes on each node: a *sender* and a *receiver*. The datagram services provided by the network layer are used for communication between the user processes, senders and receivers.

The network loader facility is accessible to application programs through a load function call provided in the run-time library. When a user process executes a load function call, a load request packet which contains the filename, source node, target node and the requesting process's process id is formed. The function determines the appropriate SAP address of the sender process on the source node, sends the load request to it, and

waits for the return of an error code. On receipt of the load request, the sender process sends a message to the receiver process on the target node specifying the name and size of the file to be loaded. If there is enough memory on the target node to accept the file, the receiver process sends an acknowledgement to the sender process on the source node. This initiates the file transfer which is accomplished by a simple hand-shaking protocol. When the file is completely transferred the receiver process performs relocation and spawns an independent process on the loaded program. Command line arguments to the program can be specified along with the filename in the load request by the user process. The length of the filename, along with its command line arguments should not exceed 80 characters. A detailed description of the sender and receiver processes is given below.

2.3.1 Sender Process

This process opens two SAPs, one for receiving load requests from user processes and another for communicating with receiver processes. When a load request is received, it determines the size of the file and the appropriate SAP address of the receiver process on the target node. A load-command message, containing the name of the file and its size is sent to the receiver. Command line arguments are also sent along with the file name. The sender process then conditionally waits for an acknowledgement from the receiver. An ack indicates willingness on the part of the receiver to accept the file. A nak indicates that the receiver does not have sufficient memory to accept the file. If the sender times out before receiving a message (which could happen if the target node crashed), an error code is returned to the user process.

File transfer between the sender and receiver processes employs a simple hand-shaking protocol. The sender process sends the number of bytes requested by the receiver process. The maximum number of bytes that can be sent on each request is limited by the maximum packet size supported by the system. The sender process optimizes the number of file read operations required by reading more bytes than are requested by the receiver. The size of the buffer used by the sender is a system parameter. The receiver indicates the success of the file transfer and the subsequent process creation by sending an error code to the sender. The sender returns this error code to the user process that made the load request.

2.3.2 Receiver Process

The receiver process opens two SAPs, one for receiving load commands from sender processes and another for data transfer operations with senders. On receipt of a load command from a sender process, it sends an acknowledgement indicating its willingness to receive the file provided it has adequate memory.

DOS executable files are made up of two parts: a header portion and the load module. The receiver process first requests only the first 36 bytes of the file which contains descriptive information about the file. This is used to determine the size of the header and the size of the load module. The relocation table contained in the header is received into a dynamically allocated block of memory. Another block of memory is allocated for receiving the load module. The load module is preceded by a 256 byte long program segment prefix (psp) which is used, among other things, as a temporary storage for command line parameters. The segment value of the starting address of the load module is the *start segment*.

The relocation table is an unordered list of relocation items each of which is a segment:offset pair. These two fields represent a displacement into the load module of a word which requires modification before the module is given control. Typically this contains the address of a variable used in the program which is initially assigned with respect to address zero by the compiler. When the file is loaded into memory all variable addresses have to be changed with respect to the starting address into which the program is loaded. This process is called *relocation* and is performed as follows: each relocation table item segment value is added to the start segment value. This calculated segment, in conjunction with the relocation item offset value, points to a word in the load module to which is added the start segment value. The result is placed back into the word in the load module.

Command line arguments are copied into psp segment at offset 0x82 and the number of bytes contained in the arguments is placed at offset ox80. These values are now accessible in the program through argc and argv variables. The entry point into the loaded program is extracted from the header block and used to create a process on the load module. An error code indicating the success of the process creation operation is sent to the sender.

2.3.3 Dynamic Linking

Loaded programs typically perform network communications after they begin execution. Therefore, a number of procedures used in network communications must be linked with each application.

Static linking produces large application programs. Each time a statically linked program is run, the main program and all procedures are loaded into memory as a single unit. This makes very inefficient use of memory as there can be multiple copies of the same procedure in memory at the same time.

Dynamic linking reduces the size of executable programs. Some common mechanisms of dynamic linking are: the system call mechanism, load time dynamic linking, execution time dynamic linking, etc.

The system call mechanism is the simplest form of dynamic linking. Unprivileged applications make system calls to request operating system services. An application uses a numeric identifier to specify the precise service being requested, and then generates a software interrupt in order to invoke the O/S service procedure. All O/S procedures are commonly linked with the O/S kernel and kept resident in memory in order to optimize performance. MicroCIM uses the system call mechanism. Load time and execution time

linking would be useful enhancements to the system. However, any implementation based on these methods would be difficult in the absence of global, distributed file system.

2.4 INPUT/OUTPUT SUBSYSTEM

Interactive input and output are provided by two processes: *window server* and *keyboard server*. The window server manages a set of virtual windows. Output from the application processes is directed to the virtual windows. The keyboard server along with a second level keyboard interrupt handler retrieves data from the keyboard device and forwards it to the application processes.

The window and keyboard servers run as system processes at a priority higher than that of the application processes to ensure timely service of input/output.

At system start-up window 0 is displayed. Pressing the F1 key will swap the current window with the next active window (if there is one). All keyboard input is directed to the process whose window is displayed. If the process running in the displayed window terminates, no further keyboard input is accepted until the F1 key is pressed to display the next active window. The keyboard server can buffer up to 10 keystrokes. When the F1 key is pressed all buffered characters (if any) are flushed.

2.4.1 The Window System

The window system manages a set of virtual windows each of which is the size of the display screen. A window can be accessed by only one process although a process can own more than one window. The window server provides facilities for opening a window (*wopen*), closing a window (*wclose*), formatted output to a window (*dprintf*), and changing the displayed window (by the F1 key).

A process must open a window before it can use it and close that window before terminating.

Application processes access window server facilities through functions provided in the application run-time library. *wopen* and *wclose* are blocking operations. They create a mailbox on which the requesting process waits for a reply from the window server, after sending an appropriate request message. The request message contains indication of the type of request (wopen or wclose) and the handle of the mailbox on which the reply is expected. The reply mailbox is destroyed after receiving a reply from the window server.

The *dprintf* function is the window system's version of *printf* in C language. It copies the contents of the variables according to the format specification onto a message buffer. The message buffer is appended to a dprintf function indication and sent to the

window server. The calling process does not wait for the window server to output this message.

2.4.1.1 The Window Server

The window server is the only process that can modify the virtual window data structure. Each virtual window is made up of a display buffer, a mailbox for keyboard input through the window, an attribute field, and cursor position coordinates. The display buffer of each virtual window maps on to the video memory of the computer. Each character on the display screen is represented by two bytes in the video memory – one byte for the ascii code of the character and the next byte for its display attributes.

The window server listens for requests on a known mailbox called WINDOW_BOX. When a request is received it identifies the service requested from the first field of the request message. The following actions are taken depending on the kind of service requested:

1. WOPEN: It finds a window that is not in use, allocates a block of memory required for the display buffer, and creates a mailbox for communication with the keyboard server. The index of the window in the virtual window array is returned to the user process via the mailbox specified in the wopen request message.
2. WCLOSE: The handle of the window to be closed is extracted from the request message. The display buffer of the window is freed and the mailbox associated with it is destroyed. An error code is returned to the user process via the mailbox specified in the wclose request message.
3. DPRINTF: The window handle is extracted from the request message. The string to be displayed is written on to the display buffer of the virtual window one character at a time. If this window is currently displayed on the screen then each character with its attributes is also copied on to the video memory. The cursor positions are accordingly updated in the virtual window.
4. CHANGE_WINDOW: This message is received from the keyboard interrupt handler when the F1 key is pressed. The window server refreshes the display with the contents of the active virtual window that follows the currently displayed window in the virtual window array.

All window operations are executed serially by the window server in FIFO order. This preserves the integrity of the window data structures and obviates the need for any critical sections in the window system.

2.4.2 The Keyboard System

Keyboard input is processed by a second level interrupt handler and the keyboard server process. All keystrokes are directed to the window currently displayed. The microcomputer hardware generates an interrupt every time a key is pressed or released. Each interrupt is accompanied by a single byte scan code on port 0x60 and a single byte status code on port 0x61. The scan code uniquely identifies the keystroke. The interrupt handler must clear the status line after processing a keystroke. Otherwise, the hardware will not be able to generate an interrupt on the next keystroke.

The interrupt handler and the keyboard server process communicate through a mailbox called the KEYBOARD_BOX. This mailbox is created by the keyboard server process at system initialization. On the occurrence of an interrupt the interrupt handler reads the scan code and status code. All key release interrupts except the shift key release interrupt are ignored. All other scan codes are conditionally sent to the window server with a timeout of zero timer ticks.

2.4.2.1 The Keyboard Server

The keyboard server listens to the KEYBOARD_BOX mailbox and processes the scan codes it receives from the interrupt handler . If the scan code is for the Esc key the system is halted. If it is for the F1 key, a request to change the displayed window with the next active virtual window is sent to the window server. Pressing or releasing the shift key toggles a shift_key_flag that is used to control the case of the characters. All other scan codes are used to look up a translation table for the corresponding ascii characters which are then sent to the keyboard mailbox associated with the currently displayed window. The server can buffer up to 10 characters in the mailbox associated with the window after which the characters are discarded. Keyboard input with echo processing is illustrated in figure 2.7.

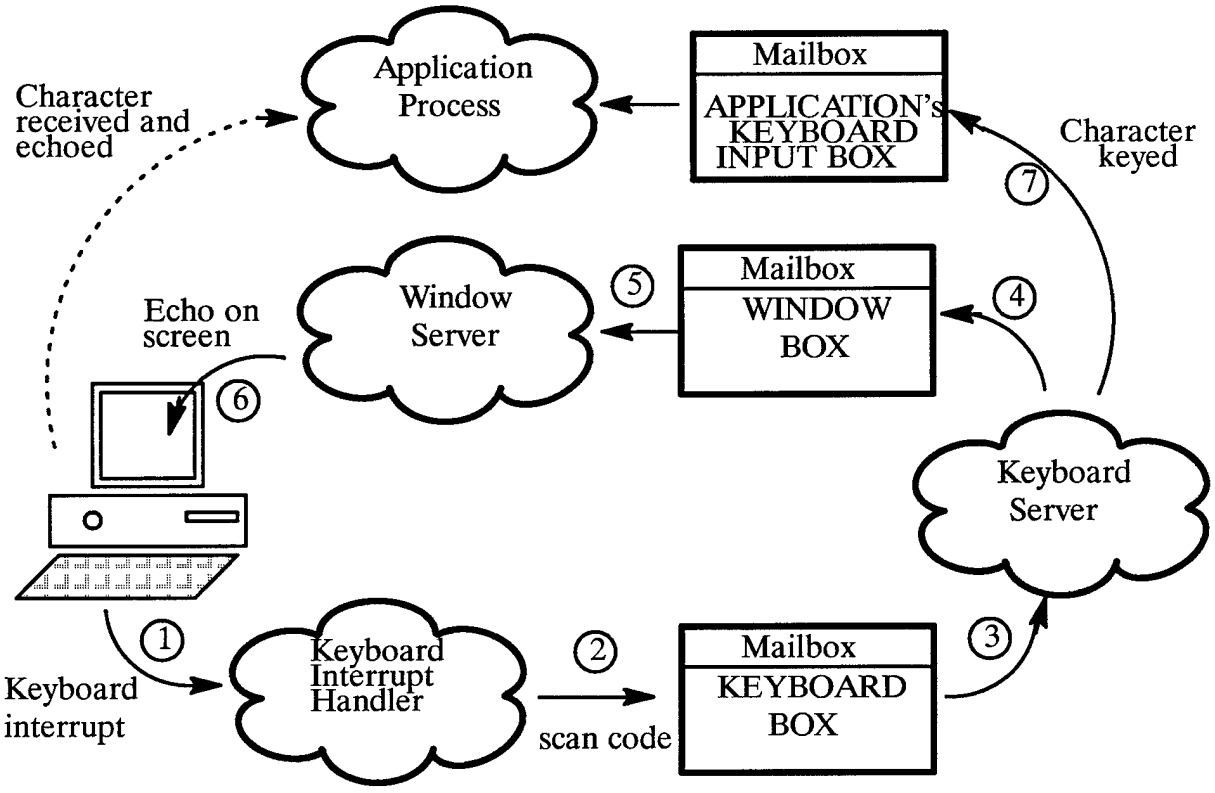


Figure 2.7 Keyboard input with echo processing

The application run–time library includes functions for providing keyboard access to application processes. *keyread* reads a character but does not echo it on the display window. *getch* reads a character and echos it on the window. *dscanf* is similar in function to standard C language *scanf*. However, not all format specifications are currently supported.

3. Performance Evaluation

The evaluation of a computer system's performance is of prime importance to system designers. The performance of various system components, which may be influenced by acting on such variables as hardware speed, is useful in problems concerning the choice of hardware for the system. It also aids in the prediction of when in the future the capacity of an existing system will become sufficient to process the installation's workload with a given level of performance. Performance evaluation is also of importance to installation managers who are concerned with a balanced and cost–effective usage of system components, while providing satisfactory user response time.

Performance of a system can be discussed only in the context of what the system is required to do. User's applications, once translated into programs and commands, can be characterized by the type and the amount of resources the system will have to allocate to execute these programs and commands. The total of resource demands generated by the user applications represent the *system workload.*

3.1 Performance Measures

Performance is characterized by a set of quantitative parameters called performance measures. *Response* and *throughput* are two metrics commonly used in characterizing performance. Response is a measure of the elapsed time required to complete a task. Throughput is the measure of how many units of work per unit time the system is completing. In general it is not possible to provide the shortest possible response time and the highest possible throughput. This follows from the fact that the fastest response will occur when the system is as lightly loaded as possible. But under conditions of light load throughput will also be low.

Throughput may be expressed in many ways: as the number of programs processed per unit time, the amount of data processed per unit time, the number of requests processed per unit time, and so on.

Throughput and response are influenced by many factors, among which are the characteristics of the workload with which it is evaluated, the system's hardware characteristics, the degree of concurrency among participating processes, and the algorithms used for assigning system resources to the programs being executed.

Aspects of a CIM environment require both short response time and high throughput capability. Fast response is an absolute necessity in performing machine control functions in near real–time. Conversely, the large volumes of data generated by machine

monitoring applications place high throughput demands on a system. In order to function correctly, a CIM system must possess sufficient capacity to sustain the throughput and response demands placed upon it. However, significant excess processing capacity should be avoided since it drives up system price but provides no better performance than a properly configured system.

The response and throughput requirements of a CIM system are determined by the collection of manufacturing machinery attached to the CIM network. The throughput and response requirements of each machine are provided by the machine's manufacturer. The throughput and response requirements of the entire distributed CIM system can be obtained by combining the requirements of each component.

The objective of this study on performance of MicroCIM is to characterize sustainable loads on Intel's 80286 and 80386 based control PCs. For each control PC, sustainable network and local interprocess communication throughput and response times are examined. This characterization will enable a person configuring an apparel CIM network to determine how many manufacturing machines can be attached to a given cell controller and how many cell controllers can be supported by a given type of control PC.

3.2 The Test Workload

The test workloads used in the study are designed to stress the network and local interprocess communication mechanisms. They are characterized by minimum I/O operations and maximum CPU utilization for communication operations. The exclusion of I/O operations from the test workload is justified because the I/O interface that may be used in the actual CIM environment may be very different from the I/O interface incorporated in the test system.

A unit of workload is defined to be the computation required by a set of two process: one that only sends messages and another that receives these messages. These two processes are run on the same node when measuring local IPC and across different nodes when measuring network IPC. Several experiments with exponential interarrival times between successive requests generated by the test programs were conducted. It was observed that the objectives of this study were best satisfied when the workload was one in which requests were generated continuously, that is, with zero interarrival time. This observation is based on the following reasons:

- The test workload is not designed to simulate the resource demands of an actual CIM network. It is designed to determine the throughput and response of the system under conditions of peak load.

- Experiments have shown that the control PC is capable of handling workloads of up to three units at peak throughput. Extending the workload beyond three units does not result in any improved utilization of system resources. However, to achieve the same amount of load on the system by using a probabilistic model, a workload of greater than six units must be used. Memory limitations in the test system preclude the possibility of running more than six multitasking processes.
- A non–probabilistic workload model is easily reproducible on different machines. Hence, the performance measures show a direct reflection of the change in the experimental parameters, namely, the hardware speed, the load on the system and the length of the messages.

Throughput is measured in bytes per second and messages per second. The response time is a measure of the time taken to complete one send/receive operation. It is expressed in milli seconds.

3.3 Results

The tests were conducted with three machines as the control PCs: an 80386 PS/2 Model P70 running at 20 MHz, an 80286 PS/2 Model 30 running at 10 MHz and a PC AT running at 8 MHz. The test configuration contained two PCs – one functioning as the control PC and the other as a cell controller. They were connected by a coax cable.

The operating system was configured with a time slice of 100 timer ticks. System processes run at a higher priority than application processes. All the benchmark programs, which run as application processes run at the same priority.

Time measurement facilities were provided by a software tool that could sample the 8253 timer in the PC hardware. It provides a resolution of 840 nanoseconds.

For each control PC, response and throughput are measured under conditions of varying load and message length. In all the experimental cases the load was increased from one unit to three units. At three units of load the inter process communication mechanism is stressed to its maximum. Hence increasing the load beyond three units does not result in any significant change in the response or throughput. The maximum message length (248) is fixed by the maximum packet size that the LAN can handle.

3.3.1 Local IPC

Figures 3.1(a) and 3.1(b) show the maximum throughput for local inter process communication in kilo bytes per second and messages per second for each of the three control PCs. The difference in maximum throughput between the three machines reflects

the differences in the speed of the hardware. The number of messages exchanged per second does not vary significantly with the length of the messages. This shows that the overhead incurred by increasing the length of messages is not significant in local IPC. Response time is defined to be the amount of time taken to complete one send/receive operation. Figure 3.2 shows the response times for each of the control PCs. Response times do not vary much with the message length. Therefore the graph shows the mean response time for various message lengths against increasing load. The marginal increase in response times with increasing load is because there can be no overlap in the execution of local IPC operations as they are executed as critical sections.

3.3.2 Network IPC

Figure 3.3 shows the network throughput for the various control PCs. The maximum throughput (in kilo bytes/sec) was 25.5 on the 386 PS/2, 17 on the 286 PS/2 and 13.5 on the PC AT. The corresponding maximum throughputs in messages/second as shown in figure 3.5 were 99, 69 and 55. In this case the remote PCs were also running the operating system. It is interesting to note that the maximum throughput in messages/sec does not vary much with message length on the 386 PS/2. However, on the 286 PS/2 the throughput varies by as much as 37% between the maximum and minimum message lengths and by 35% on the PC AT. This shows that at peak loads the 386 PS/2 has adequate processing power to handle the network traffic.

The performance of the control PC may be affected by the processing power of the remote PC. This is of particular importance in the case where the control PC is faster than the remote PC. For example, as shown in the performance evaluation of the local IPC, the 386 PS/2 is about 2.5 times faster than the 286 PS/2. To identify the effects of the remote PC on the control PC's performance, the benchmark programs were executed without the operating system on the remote nodes. That is, the remote node contained only that part of the network management system necessary to communicate with the ARCNET adapter. All the packets that were received on the remote node were discarded.

Figures 3.4 and 3.6 show the throughput without the operating system on the remote nodes. With the 386 PS/2 as the control PC, the maximum throughput (figure 3.4) increased by 51% from 25.5 Kilo bytes/sec to 38.6 kilo bytes/sec. There is also a proportionate increase in the throughput (figure 3.6) measured in messages/sec. However, there is no significant change in the throughput with the 286 PS/2 and the PC AT, both of which had a faster machine (386 PS/2) as the remote node. This can be attributed to the fact that data packets can be put on the LAN only as fast as it can be taken out of it. Thus

it is desirable to have adequate processing power on the cell controller if it is expected to exchange large amounts of data with the control PC.

Response time is defined to be the amount of time taken to complete one send/receive operation across the network. The response times with and without the operating system on the remote node are shown in figures 3.7 and 3.8 respectively. In all of the six graphs in these two figures, it can be observed that the response times decrease with increasing load. This is a result of better utilization of the network IPC mechanism with an increase in the degree of concurrency among the benchmark processes. The average response times are about 10 ms on the 386 PS/2, 14 ms on the 286 PS/2 and 17.5 ms on the PC AT.

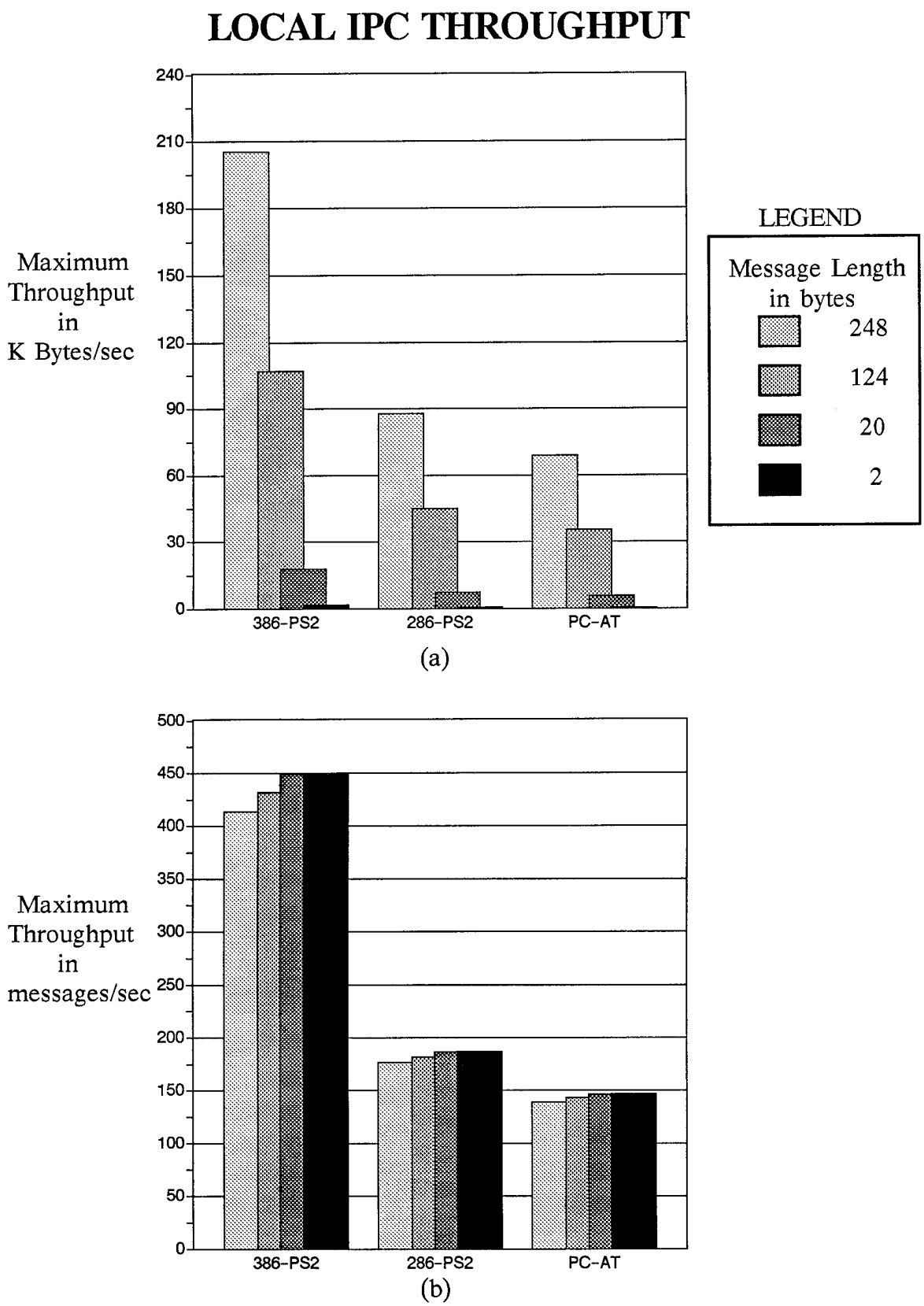


Figure 3.1 Local throughput on various control PCs

LOCAL IPC RESPONSE

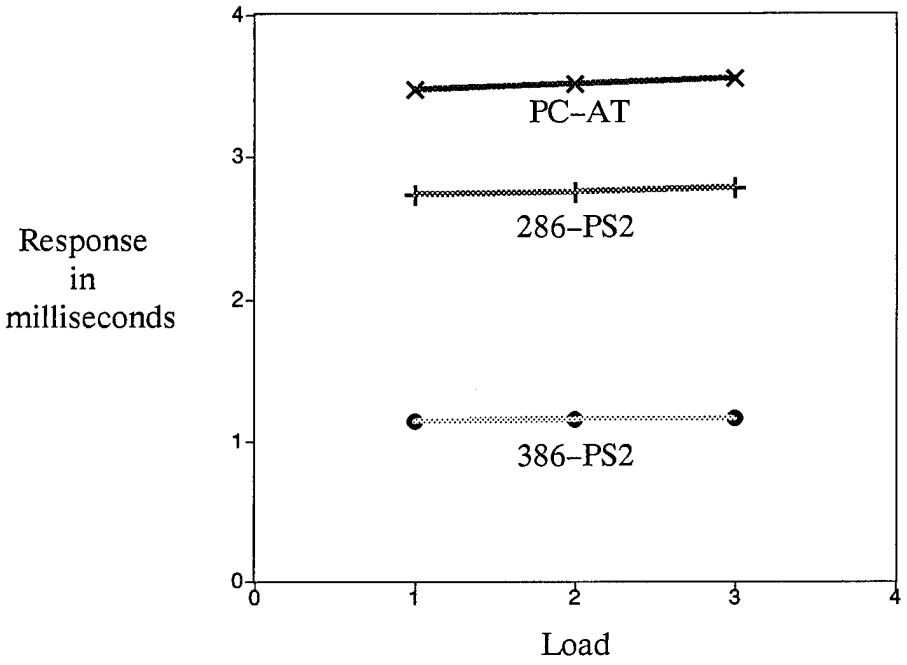


Figure 3.2 Local Response on various control PCs

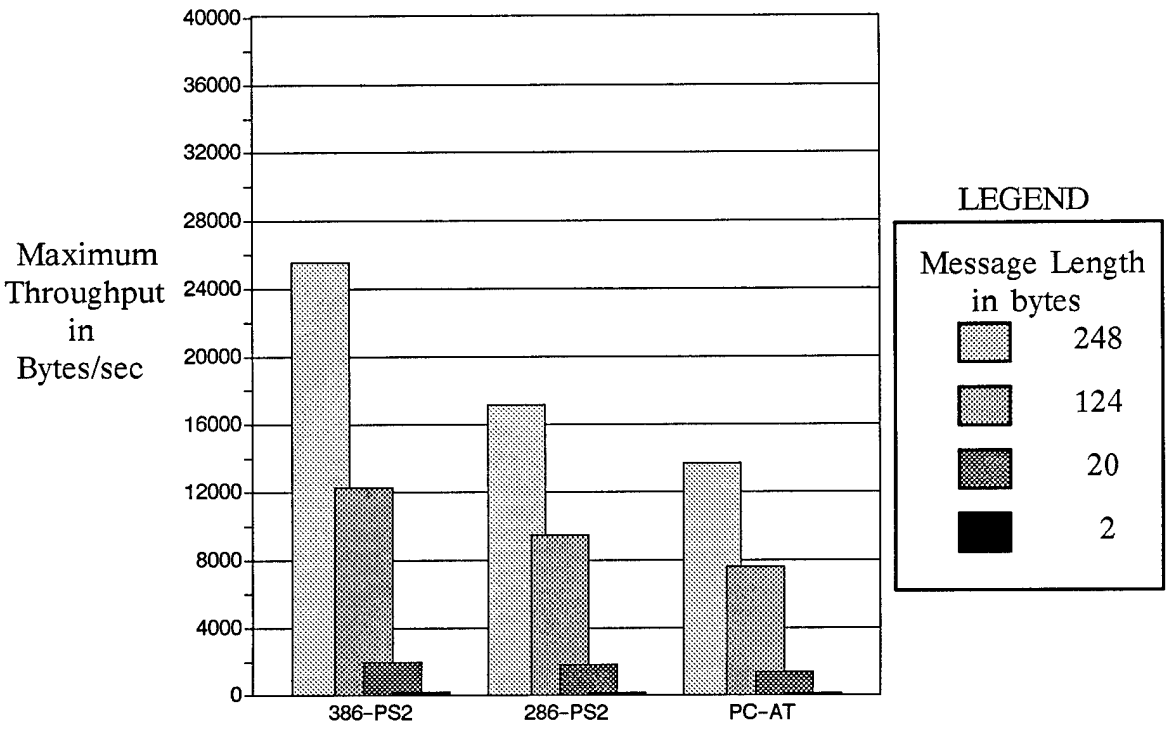


Figure 3.3 Throughput on various control PCs with OS on remote PC

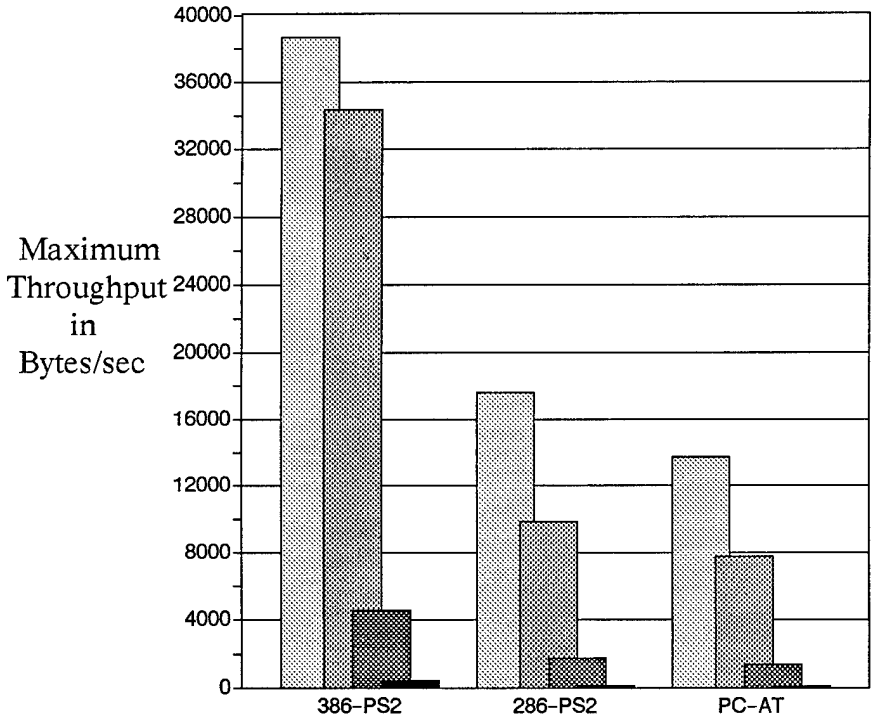


Figure 3.4 Throughput on various control PCs without OS on remote PC

NETWORK IPC THROUGHPUT

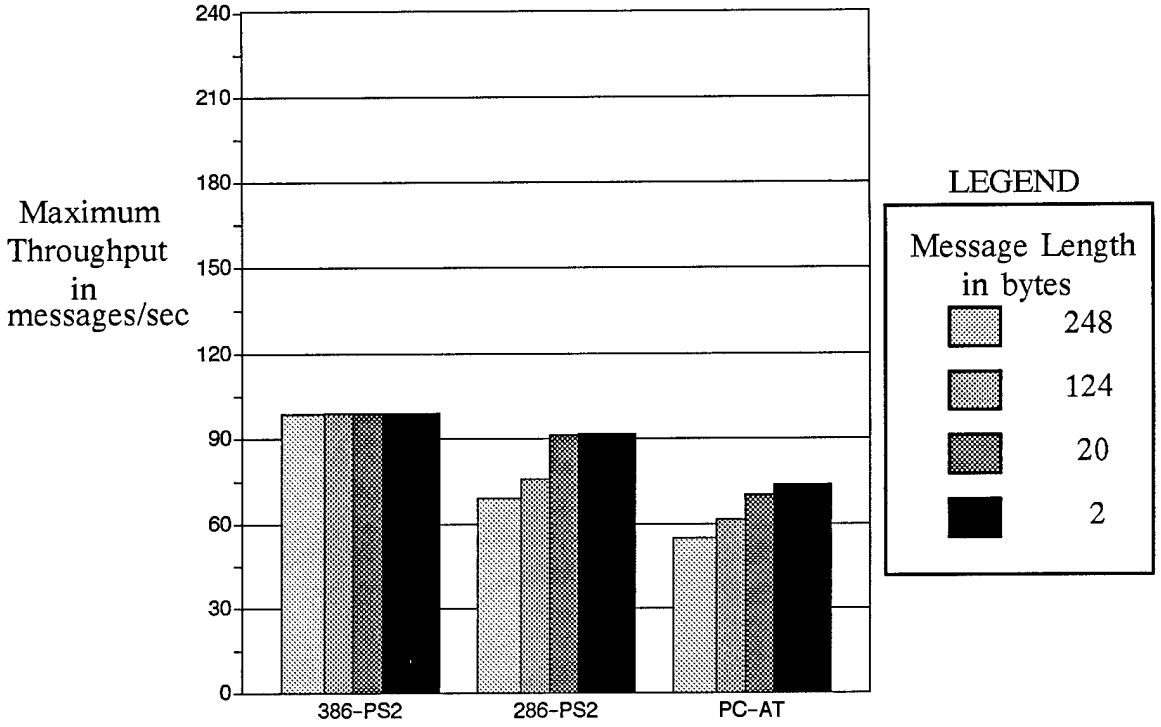


Figure 3.5 Network throughput on various control PCs with OS on remote PC

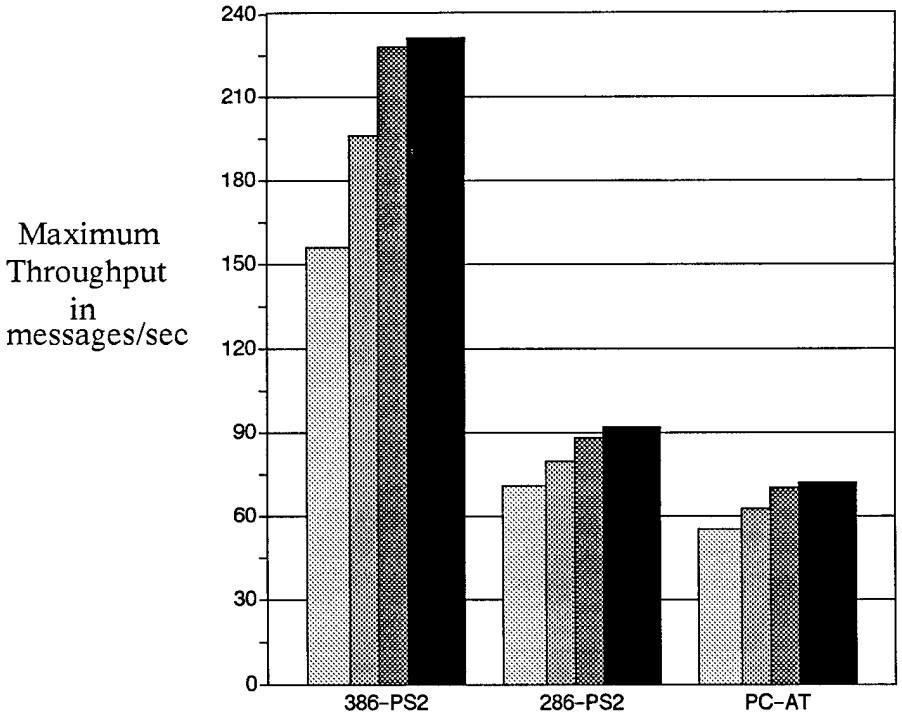


Figure 3.6 Network throughput on various control PCs without OS on remote PC

NETWORK IPC RESPONSE

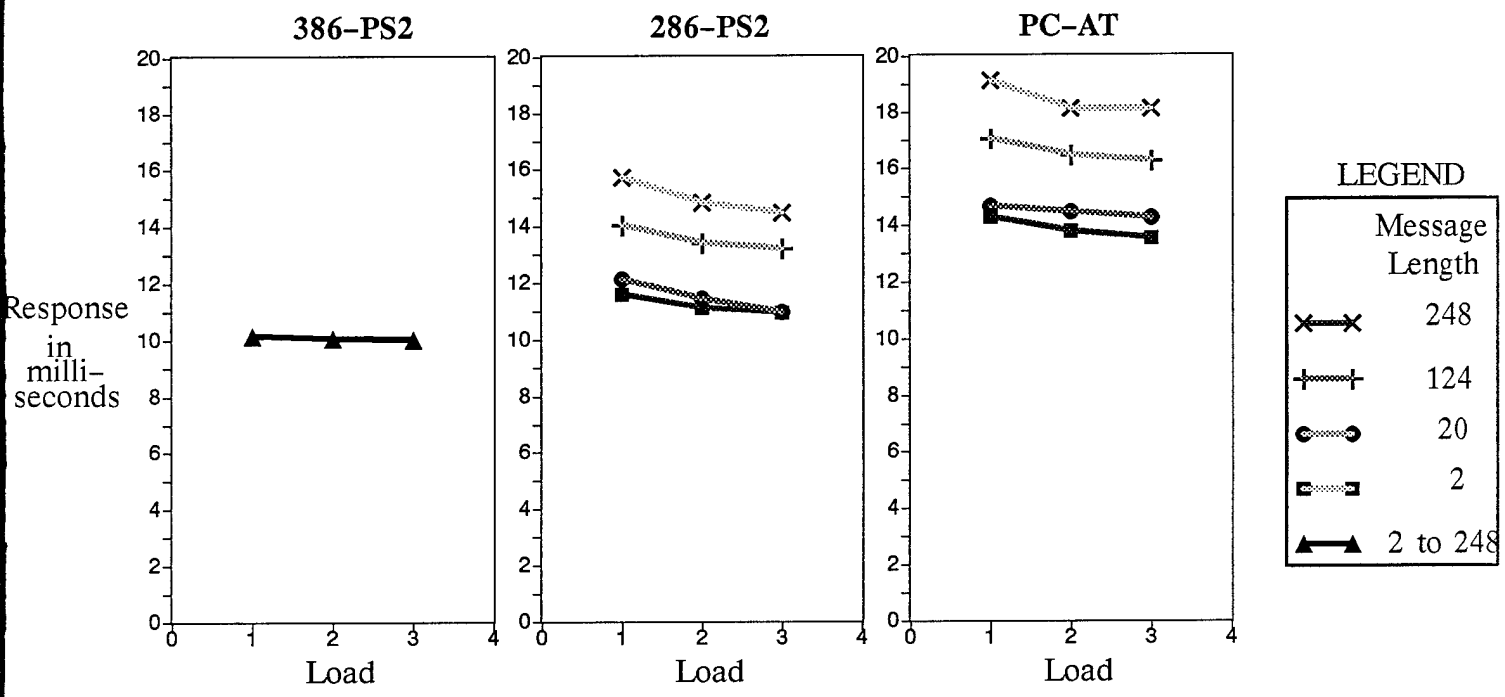


Figure 3.7 Network response on various control PCs with OS running on remote node

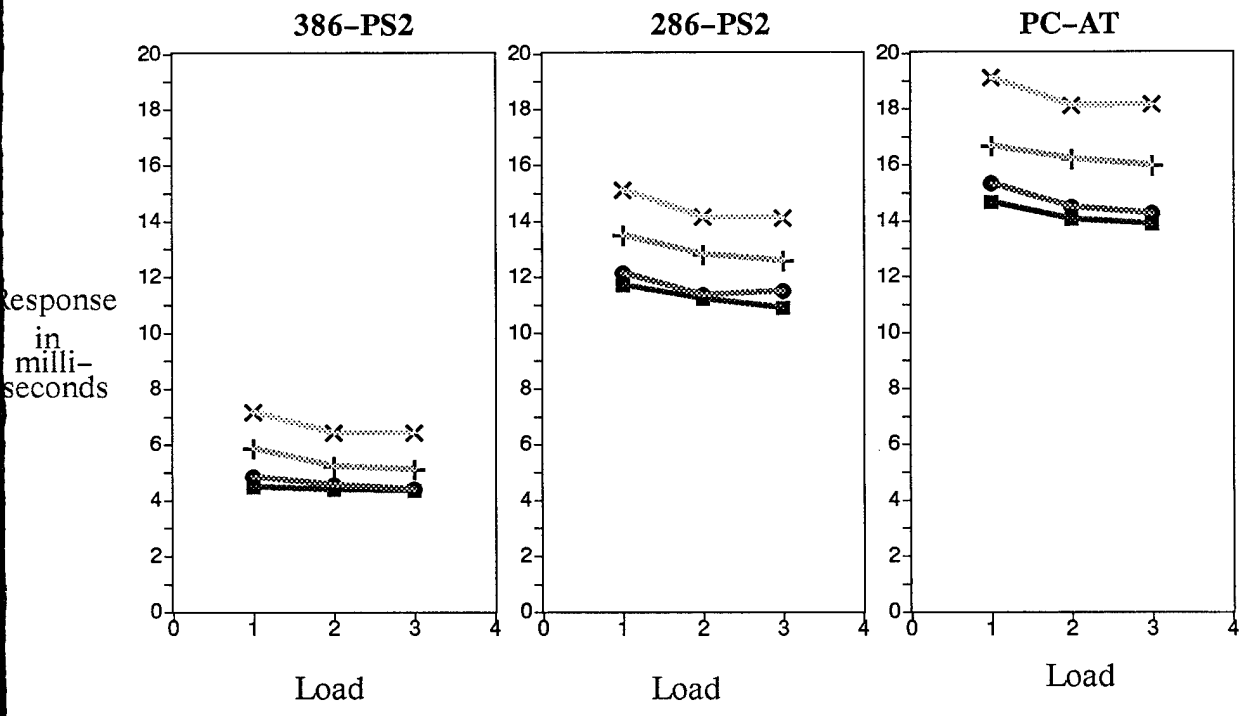


Figure 3.8 Network response on various control PCs without OS running on remote node

4. Conclusions

The development of a small, portable, multitasking distributed operating system called MicroCIM has been completed. The distributed system is designed to be a platform on which a complete Computer Integrated Manufacturing environment for the apparel industry can be based. Currently, the operating system can run on industry standard personal computers.

The operating system is designed to be portable across different hardware platforms. AN ARCNET LAN is currently used as the physical layer. A multitasking kernel provides facilities for process management, interprocess communication and synchronization and memory management. The network management system, which uses the services of the kernel, provides facilities for intra- and inter-site process communications. The services provided by the network management system are accesses by application programs through the application program interface. A network loader provides remote program loading facilities. A window management system and a keyboard management system provide interactive input/output facilities to all processes.

An evaluation of the performance of the local and network inter process communication mechanisms for various control PCs, in terms of sustainable throughput and response, was also presented. The values obtained indicate that MicroCIM will provide suitable performance in the environment for which it was designed.

5. References

Cheriton, D. A., The V distributed system. *Comm. of the ACM* 31, 3(Mar. 1988), 314–333.

Domenico, F., Serrazi, G., Zeigner, A. *Measurement and Tuning of Computer Systems.* Prentice Hall, Inc., 1983.

Fortier, J. P., *Design of Distributed Operating Systems – concepts and technology.* Intertext Publications. Inc., 1986.

Lampson, B. W., et. al., *Distributed Systems – Architecture and Implementation.* Lecture Notes in Computer Science, 105, Springer–Verlag, 1981.

Lampson, B. W., Designing a global name service. In *Proceedings of the 5th ACM Conference on Principles of Distributed Computing* (Calgary, Canada, Aug.), ACM, New York, 1986. pp. 1–10.

Liskov, B., Distributed programming in Argus. *Comm. of the ACM* 31, 3(Mar. 1988), pp. 300–313.

Milenkovic, M., *Operating Systems – concepts and design.* McGraw–Hill Book Company, 1987.

Pimentel, R. Juan., *Communication Networks for Manufacturing.* Prentice–Hall, Inc., 1990.

Svobodova, L., *Computer Performance Measurement and Evaluation Methods: Analysis and Applications.* American Elsevier Publishing Company, Inc., 1976.

MicroCIM

An Operator's Manual

Dr. J. M. Westall
Dr. A. W. Madison

Bapiraju Buddhavarapu
Sudhir Moolky

Department of Computer Science
Clemson University
Clemson SC 29634-1906

1.1 Introduction

This manual describes the following procedures :

a) Starting microCIM
b) Working with the console window and commands
c) Other system/user windows
d) Shutting down microCIM
e) Modifiying and recompiling microCIM

1.2 Installing microCIM

The microCIM distribution is organized as follows. The directory \source has the source files of microCIM. It also has three system object files which are needed when compiling microCIM application programs. The directory \bin contains the microcim executable and a definition file *"arcnet.def"*. Finally the \apps directory contains *"c0l.obj"* and *"cc.bat"*. These are to be used when compiling application programs. Create a directory called \microcim on your machine and copy(with *xcopy -s*) the directory structure as it is. When compiling application programs in the \apps directory keep in mind that the Turbo C linker looks for the system object files in the current directory(\apps). Modify *"cc.bat"* to reflect the location of these object files. Currently they are located in \source.

The definition file is used to set four system parameters. They are(in order) as follows :

1. <IRQ #> : This identifies the IRQ line used by the ARCNET adapter board. For example, if IRQ 3 is used by the adapter board then the value will be 3.

2. <I/O port> : This identifies the I/O port used in communicating with the LAN controller chip on the ARCNET adapter board. For example, if port number 2E0 is being used then this field will be 2E0.

3. <buffer address> : The ARCNET adapter has an on board memory which it uses to store messages during transmission/reception. This is mapped to the computer's main memory at <buffer address>. For example if the on board memory is mapped at main memory location 0D000 then this value will be 0D000.

4. <video buffer address> : This identifies the address in main memory where the video buffer is mapped. For monochrome monitors it is 0B000 whereas it is 0B800 for color monitors.

An example of all the fields in the definition file is

5 2E0 0C600 0B000

where IRQ line #5 is used, port 2E0 is used for communication with the LAN controller chip, the on board memory is mapped to main memory location 0C600 and a monochrome monitor is being used on the computer.

The first three parameters are obtained by using the reference diskette for MicroChannel

bus boards. The configuration viewing utility on the diskette will show the values for which the ARCNET adapter board has been set.

1.3 Starting microCIM

Two files are required to start up microCIM. They are "*microcim.exe*" and "*arcnet.def*". MicroCIM will read the definition file when starting up and set internal parameters according to these field values. To start up microCIM type "microcim".

1.4 Working with the console window and commands

Once microCIM has been started up, the console window comes into focus. There may be many windows active at a given time but only one will be displayed on the screen. The F1 key shuffles and displays the active windows in a round robin manner.

There are a limited number of commands one can use in the console window. They are listed below :

a) **load** <file.exe> <src_netid> <dest_netid> : This command invokes the network loader. Three arguments are required, the name of the executable file(file.exe), the network id of the machine on which the executable is located(src_netid) and the network id of the machine on which the executable is to be executed(dest_netid). For example the command "load server.exe 255 126" copies the executable file from the machine whose network id is 255 and executes it on the machine whose network id is 126. The network id of a machine is the same as the id of its ARCNET adapter card.

b) **nodeid** : This command will display the network id of the machine on which it is invoked.

c) **active** : This command lists all the active processes in the system. For each process listed, its type(system or user), priority, process id, status(running or ready) and number of mailbox handles are also displayed.

d) **blocked** : This command lists all the processes which are blocked on a mailbox. For each process listed, its type(system or user), priority, process id, name of mailbox blocked on etc are also displayed.

e) **checkstack** <pid> : This command checks if the process whose id is <pid> has suffered a stack overflow. It also checks stack integrity by searching for the stack signature which is put on the stack at process creation time.

f) **mail** : This command lists all the mailboxes in the system. For each mailbox listed, its name, owner process id, number of links to the mailbox, number of message slots, size of message buffer, id's of processes blocked on a send or receive etc are also displayed.

g) **memleft** : This command will display the amount of free memory in the system. It will also display the size of the largest available free block.

1.5 Other system/user windows

Besides the console, there may be other windows active. These will be opened by other system or user processes. Only one window is displayed on the screen at any given time, so use the F1 key to shuffle to the next window. If a process is expecting keyboard input you should first display it's window on the screen before typing anything on the keyboard.

1.6 Shutting down microCIM

Pressing the ESC key will shut down microCIM and return you to the DOS prompt. If the system has hung up then pressing the ESC key may not always work. If it doesn't, press the CTL–ALT–DEL keys simultaneously to reboot DOS. Sometimes that will not work either. In that case reset the computer by either pressing the reset button(if provided) or by powering off and on.

MicroCIM

A Programmer's Manual

Dr. J. M. Westall
Dr. A. W. Madison

Bapiraju Buddhavarapu
Sudhir Moolky

Department of Computer Science
Clemson University
Clemson SC 29634-1906

Table of Contents

1. Introduction 3

2. Modifying and recompiling microCIM 4

3. Compiling application programs 5

4. Library routines 6

5. Hello world 7

6. Local interprocess communication and synchronization 8

7. Remote interprocess communication and synchronization 12

8. Connection oriented message passing 16

9. MicroCIM function reference 20

    9.1 Mailbox routines 20

    9.2 Process manipulation routines 24

    9.3 Memory routines 27

    9.4 Window and I/O routines 29

    9.5 Network datagram service routines 31

    9.6 Network virtual circuit routines 34

1. Introduction

This manual is a guide to writing application programs for microCIM. Example program segments are shown which illustrate the use of various functions. Finally the syntax of all functions is given.

2. Modifying and recompiling microCIM

If any modifications are made to microCIM it needs to be recompiled. For recompilation you will need Borland's Turbo C editor/compiler(at least version 2.01) and also Borland's Turbo Assembler(at least version 1.0). The recompilation process is facilitated by the use of a makefile. You will need the two files "*makefile*" and "*microcim.rsp*" to be able to recompile microCIM. The "*makefile*" lists all the files which are part of microCIM. "*Microcim.rsp*" lists the same files along with the Turbo C library modules for linking using Tlink. Note that "*makefile*" lists only the .c files as dependencies and not the .h files. If you modify .h files then you will have to delete the corresponding .obj file and then remake. For example if you modified "*netcmgr.h*" then you have to delete "*netcmgr.obj*" and then recompile the system. However if you modified "*netcmgr.c*" then you do not need to delete "*netcmgr.obj*". You can straightaway recompile the system.

Once the system files have been modified just type "make" at the DOS prompt. This will automatically recompile and link the modified system files. Listed below are some of the files which represent important system functions.

File(s)	**Contain**
microcim.c	console process
stack.c, syscalls.c	process manipulation
dalloc.c	memory routines
syscalls.c, oslib.c	mailbox routines
iocalls.c	window routines
iocalls.c	keyboard routines
api.c, netapi.c	datagram network facilities
api.c, netapi.c, netcmgr.c	virtual circuit network facilities
loader.c	network loader (sender and receiver)

3. Compiling application programs

All application programs need to be compiled and linked in a way which is different from compiling and linking microCIM. A batch file "*cc.bat*" has been provided to facilitate this process. To compile an application program "*app.c*" just type "cc app". Do not give the .c extension, it is automatically appended.

Certain files are linked with each application program. The most important of them is "*c0l.obj*". This is Borland Turbo C Compiler's large model start up file which has been modified to work with microCIM. If "*c0l.obj*" is not present then compile "*c0l.asm*" with Tasm, the Turbo Assembler. The command is simply "tasm c0l.asm". The batch file "*cc.bat*" will attempt to link the following microCIM libraries with the application program : *oslib.obj, api.obj* and *iocalls.obj*. Make sure that these libraries are present otherwise compilation will fail.

In addition "*cc.bat*" provides paths to Borland Turbo C's *include* and *lib* directories. Make sure that these are correct according to the system on which the application program is being compiled.

To run a compiled application program, start microCIM and load the executable from the console as described in the operators manual. If the application opens a window you have to press the F1 key to be able to see it.

4. Library routines

There are many Turbo C library routines which cannot be used. MicroCIM provides alternate routines which are shown below.

Do not use	**Use instead**
printf(), scanf(), getc(), getchar() etc. Other graphics and window routines.	wopen(), dprintf(), dscanf(), getscr() wclose() etc.
malloc(), free() and calloc()	dmalloc(), dcalloc(), dfree(), dmemleft()
strdup()	none
exit()	terminate()

String routines like strcpy(), strcmp(), sscanf(), strcat() etc. and assert() can be used. File routines like fopen() etc. can also be used.

5. Hello world

Shown below is a hello world program segment.

```
/*---------------------------------------------------------------->*/

#include <stdio.h>
#include <oslib.h>

int my_window;

        /* Open a window to display messages */

        my_window = wopen();

        if( NULL == my_window )
        {
                /* error */
                terminate();
        }

        /* Print a hello world message */

        counter = 1;
        dprintf( my_window, "%d. Hello world.\n", counter);

        /* sleep 5 seconds so that the message can be viewed */
        /* 18 clock ticks = 1 second */

        snooze(18*5);

        wclose( my_window );
/*<----------------------------------------------------------------*/
```

6. Local interprocess communication and synchronization

This can be achieved using mailboxes and message passing. Shown below is the code for a server which creates a mailbox and then reads messages from it. Also shown is a client which opens the server's mailbox and writes a message into it. Note that the message length and the server's "address" are well known.

```
/*------------------------------------------------------------->*/
#include <stdio.h>
#include "oslib.h"

#define MAX_BUF_LEN 50

/*---------->*/
void server()
/*<----------*/

{

int my_window;

MAILBOX    *talk_box;
int        mode;
int        num_buffers;
char       message_buffer[ MAX_BUF_LEN ];

int        return_value;
int        mbox_count;
char       mbox_name[20];

        /* Initialize some values */

        mode = M_SEND;
        num_buffers = 5;
        mbox_count = 0;

        /* Open a window */

        my_window = wopen();
        if( NULL == my_window )
        {
                terminate();
        }
```

```
        /* Create a mailbox to communicate with the client */
        /* This is the well known "address" */

        sprintf( mbox_name, "mailbox #%d", mbox_count);
        talk_box = mcreate( mbox_name, mode, num_buffers,
                            MAX_BUF_LEN );

        if ( NULL == talk_box )
        {
                dprintf(my_window, "Server Mailbox creation failed.\n");
                snooze(18*5);
                wclose(my_window);
                terminate();
        }

        /* Go into an infinite loop and read messages from clients */
        /* After reading the message, display it in the window */

        while(1)
        {
                return_value = mreceive( talk_box, message_buffer,
                                         MAX_BUF_LEN );

                if ( 0 > return_value )
                {
                        dprintf(my_window, "Server Bad receive on mailbox.\n");
                        mdestroy( talk_box );
                        snooze(18*5);
                        wclose(my_window);
                        terminate();
                }

                else
                {
                        message_buffer[return_value] = '\0';
                        dprintf(my_window, "Received : %s", message_buffer);
                }
        }

}

/*<-------------------------------------------------------------*/
```

```
#include <stdio.h>
#include "oslib.h"

/*---------->*/
void client()
/*<----------*/

{

int             my_window;

MAILBOX     *my_mbox;
char            client_message[ MAX_BUF_LEN ];
int             message_length;

int             return_value;

        /* Open a window */

        my_window = wopen();
        if( NULL == my_window )
        {
                terminate();
        }

        /* Open the server mailbox. It's name should be well known */

        my_mbox = mopen("mailbox #0");
        if ( NULL == my_mbox )
        {
                dprintf( my_window, "Client : Server mailbox open failed.\n");
                snooze(18*5);
                wclose(my_window);
                terminate();
        }

        /* Send a message to the server */

        sprintf( client_message, "Hello from client to server" );
        message_length = strlen( client_message );

        return_value = msend( my_mbox, client_message, message_length);
        if ( 0 > return_value )
        {
                dprintf( my_window, "Client send failed.\n");
        }
```

```
        else
        {
                dprintf( my_window, "Client send successful.\n");
        }



        /* Finish up */

        snooze(18*5);
        mclose( my_mbox );
        wclose( my_window );

}

/*-------------------------------------------------------------------*/
```

7. Remote interprocess communication and synchronization

Mailboxes are insufficient to handle this need. Therefore we use the network facilities. There are two types of services provided by the network layer, a potentially unreliable message passing mechanism(datagrams) and a totally reliable one(connection oriented management).

First we show the datagram approach. Like the previous example, we have a server which opens a network port and listens for messages in an infinite loop. Then we have a client who sends messages to the server using the network facilities. The client also attempts to load an executable program. The loaded application program will start running as soon as the loading is complete.

```
#include <stdio.h>
#include "oslib.h"
#include "network.h"

#define MAX_BUF_LEN 50

int             my_window;
int             num_buffers;
int             status;
int             mesg_count = 0;
int             net_handle;
struct sidtype      cl_add;
char            cl_mesg[MAX_BUF_LEN+1];
int             rcv_len;

/*---------------->*/
void net_server()
/*<----------------*/

{
        my_window = wopen();
        if( NULL == my_window )
        {
                terminate();
        }
        dprintf( my_window, "SERVER : It's alive It's alive\n" );

        /* Open a network port for communication */
        /* Set number of port buffers to 75 and size of each buffer to 50 */
        /* The port is named "SERVER" */
```

```
        num_buffers = 75;
        status = dg_open0( "SERVER", &net_handle, MAX_BUF_LEN,
                        num_buffers);

        if (status)
        {
                dprintf(my_window, "Server could not open port.\n");
        /* error handling */
        }

        while(1)
        {
                /* read a message from the port if any */
                /* if no message, then block till one becomes available */

                /* the sender's node and port id will be available in */
                /* "cl_add" when we return from this call */

                status = dg_recv0( net_handle, &cl_add, cl_mesg,
                                MAX_BUF_LEN, &rcv_len);
                if ( status )
                {
                        dprintf(my_window, "Server Bad receive on port.\n");
                /* error handling */

                        dg_close0( net_handle );
                        wclose(my_window);
                        terminate();
                }

                else
                {
                        cl_mesg[ rcv_len ] = '\0';
                        dprintf(my_window, "RECV from %d : %s\n",
                         cl_add.node, cl_mesg);
                        ++mesg_count;
                        dprintf(my_window, "Count : %d\n\n", mesg_count);

                }

        }  /* end of while(1) */

}

/*-------------------------------------------------------------*/
```

```
#include <stdio.h>
#include "oslib.h"
#include "network.h"

#define MAX_BUF_LEN 50

/*------------>*/
void net_client()
/*<-----------*/
{

char            cl_mesg[MAX_BUF_LEN+1];
char            port_name[20];
int             mesg_len;

int             status;
struct sidtype      serv_add;
int             net_handle;
int             node_id;
int             pid;
int             msgcount;

        /* Get the network id of our host machine */
        /* Then get our process id */

        node_id = getnodeid();
        pid = getpid();

        /* Create a message to send to the server */

        sprintf( cl_mesg, "Hello networking world.");
        mesg_len = strlen(cl_mesg);

        snooze(18*10);

        /* Depending on the node id, call the network loader to load a program */
        /* The loader loads the executable from node 126 onto node 255 or */
        /* does the reverse */

        if ( node_id == 126)
                loader("mclt.exe",126,255);
        else if ( node_id == 255)
                loader("mclt.exe",255,126);
        else
                 terminate();
```

```
        /* Open a port with name "CLIENT" to communicate with the server */

        sprintf( port_name, "CLIENT/%d/%d", nid, pid );
        status = dg_open0( port_name, &net_handle, 50, 5 );
        if (status)
                terminate();

        /* Find the server's network address and port id using it's well */
        /* known port name */

        status = dg_findid0( net_handle, "SERVER", &serv_add );
        if (status)
        {
                dg_close0( net_handle );
                terminate();
        }

        for( msgcount = 0; msgcount < 25; msgcount++ )
        {
                /* send 25 messages to the server */

                dg_send0( net_handle, &serv_add, cl_mesg, mesg_len );
        }

        /* close and return */

        dg_close0( net_handle);

}

/*-----------------------------------------------------------------*/
```

8. Connection oriented message passing

The previous section introduced you to the datagram network facilities. In some cases where reliability of the message passing network is of absolute necessity, the datagram approach has a likelihood of failing. The connection oriented message passing system guarantees delivery of messages from the sender to the receiver. It also guarantees their ordering. Below is shown the same server/client example from the previous subsection modified to use the connection oriented message passing facilities.

```
#include <stdio.h>
#include "oslib.h"
#include "netdefs.h"
#include "netcmgr.h"

#define MAX_BUF_LEN 50

/*---------------------->*/
void interconn_server()
/*<----------------------*/

{

int     handle;
char    clt_buf[50];
int     msg_len;
int     status;
int     my_window;
struct  sidtype clt_add;

        /* Open a window */

        my_window = wopen();
        if( NULL == my_window )
        {
                /* handle error */
        }

        dprintf(my_window, "Interconnect server is OK.\n");

        /* Open a network port for communication. Name it "INTERC_SERV". */
        /* This will be the well known port name. Allot 5 message buffers */
        /* with each being 50 bytes in length. */

        status = dg_open0( "INTERC_SERV", &handle, 50, 5);

        if ( 0 != status )
```

```
        {
                dprintf( my_window, "Server : Couldn't open port.\n");
                snooze(18*5);
                wclose(my_window);
                terminate();
        }


        /* Perform a blocking read on the port for messages from clients */
        /* Note that the connection management is only simplex and in this */
        /* case it is from client to server. Therefore the server is unaware */
        /* of the connection management and receives the messages as if it */
        /* were handling datagram messages */

        while(1)
        {
                dprintf( my_window, "Waiting for messages.....\n");
                status = dg_recv0( handle, &clt_add, &clt_buf, 50, &msg_len );
                if ( 0 != status )
                {
                        dprintf(my_window, "Bad recv by server.\n");
                        while(1);
                }

                clt_buf[msg_len] = '\0';
                dprintf( my_window, clt_buf );

                /* Sleep a long time so that client messages will not be */
                /* read. Hopefully this will force the client to timeout */
                /* This will also cause the server's port buffer to overflow since */
                /* the client will continue to send messages */

                dprintf( my_window, "\nSleeping a bit...\n");
                snooze(18*39);

        }

        dprintf( my_window, "Am finished.\n");
        dg_close0( handle );
        wclose( my_window );
}

/*-------------------------------------------------------------------*/
```

```
/*---------------------->*/
void interconn_client()
/*<----------------------*/

{

int     i;
int     handle;
int     handle_table[5];
struct  sidtype serv_add;

char    mesg_buf[75];
int     my_pid;
int     status;
int     my_window;
int     mesg_count = 0;

        my_window = wopen();
        if( NULL == my_window )
        {
                /* handle error */
        }

        dprintf(my_window, "Interconnect client on screen.\n");

        /* Typically the network address and port id of the server is */
        /* is found using its well known port name "INTERC_SERV" */

        status = dg_findid0( 5, "INTERC_SERV", &serv_add );
        if( status )
        {
                dprintf(my_window, "Couldn't locate server address.\n");
                while(1);
        }

        /* Open a virtual connection to the server using the server */
        /* address just obtained. Note that this opens up a port for */
        /* communication with the server. Therefore just like the datagram */
        /* approach we need to specify a port name, message length and */
        /* message buffer length */

        status = vc_open( "INTERC_CLT", &handle, &serv_add, 50, 10);
        if ( 0 != status )
        {
                snooze(18*5);
```

```
        wclose(my_window);
        terminate();
    }

    dprintf( my_window, "Did a vc_open for server address.\n");

    /* Send some messages to the server on the connection and then */
    /* terminate */

    while (1)
    {
        ++mesg_count;
        sprintf( mesg_buf, "Client at node %d sending message %d",
                 getnodeid(), mesg_count);
        status = vc_send( handle, mesg_buf, strlen(mesg_buf) );
        if ( 0 != status )
        {
            dprintf( my_window, "Bad send by client.\n");
            while(1);
        }
        else
        {
            dprintf(my_window, "Successful send by client.\n");
        }

        snooze(18*10);
        if ( 8 == mesg_count ) break;
    }

    dprintf( my_window, "Finished sending messages.\n");
    snooze(18*5);
    wclose(my_window);

    vc_close( handle );
}
```

9. MicroCIM function reference

9.1 Mailbox routines :

#include "oslib.h" /* for all mailbox routines */

Syntax

MAILBOX *mcreate(name, mode, nslots, buflen);
char *name;
int mode, nslots, buflen;

Remarks

mcreate creates a mailbox and names it *name*. The *mode* is M_SEND or M_RECEIVE or M_SENDRECEIVE and specifies whether messages can only be sent to the mailbox, read from the mailbox or both. The owner of the mailbox can read and send messages irrespective of the mode. The mailbox is created with *nslots* number of message buffers and each buffer is *buflen* bytes in size. *mcreate* returns a handle which is used for subsequent operations on this mailbox. The process which creates a mailbox becomes it's owner. When the mailbox has served it's purpose the owner should use *mdestroy* to destroy the mailbox.

Return value

NULL	Creation of mailbox failed.
non NULL	Mailbox handle is returned.

Syntax

int mdestroy(handle);
MAILBOX *handle;

Remarks

mdestroy destroys a mailbox which was created using *mcreate*. Only the owner of a mailbox may destroy it. If any processes are blocked on the mailbox then they receive a failure code. All messages remaining in the mailbox are lost.

Return value

0	Mailbox destroyed and buffer space deallocated.
-1	Error. Mailbox does not exist.
-2	Error. Calling process is not the mailbox owner.
> 0	# of processes which have not yet closed this mailbox. The mailbox is destroyed but the buffer space is not deallocated until the last *mclose*.

Syntax

```
MAILBOX   *mopen( name );
char *name;
```

Remarks

A process(non owner) gains access to a mailbox using *mopen*. The mailbox must have been created prior to executing this call. After opening a mailbox the process can send and/or read messages depending on the mode with which the mailbox was created. When the process finishes using the mailbox it must be closed using *mclose*.

Return value

NULL	Open failed.
non NULL	Mailbox handle is returned.

Syntax

```
int         mclose( handle );
MAILBOX *handle;
```

Remarks

A process(non owner) releases access to a mailbox using *mclose*. If the process owns the mailbox then it should use *mdestroy* instead of *mclose*.

Return value

0	Successful close.
-1	Invalid mailbox handle.
-2	Caller is the owner of the mailbox. Use *mdestroy* instead of *mclose*.

Syntax

```
int         msend( handle, msg, len );
MAILBOX *handle;
char *msg;
int len;
```

Remarks

msend is used to send a message *msg* to a mailbox which is specified by *handle*. *Handle* is obtained by calling *mopen*. *Len* denotes the length of *msg*. If *msg* is longer than the length of the mailbox buffer, it will be truncated. The process executing *msend* will block indefinitely until a free message buffer becomes available. The call will fail if the owner process did not create the mailbox with an M_SEND or M_SENDRECEIVE mode.

Return value

>= 0	Send was successful. This also indicates the number of bytes sent.

-1	Invalid mailbox handle.
-2	Mailbox mode is receive only.
-3	Mailbox is full.

Syntax

```
int       msendc( handle, msg, len, sleeptime );
MAILBOX *handle;
char *msg;
int len, sleeptime;
```

Remarks *msendc* is used to send a message *msg* to a mailbox which is specified by *handle*. *Handle* is obtained by calling *mopen*. *Len* denotes the length of *msg*. If *msg* is longer than the length of the mailbox buffer, it will be truncated. The process will block for a maximum of *sleeptime* tics while waiting for an available message buffer in the mailbox. The time unit is based on the timer interrupt frequency on an IBM PC. It is approximately 18 tics per second. The call will fail if the owner process did not create the mailbox with an M_SEND or M_SENDRECEIVE mode.

Return value

>= 0	Send was successful. This also indicates the number of bytes sent.
-1	Invalid mailbox handle.
-2	Mailbox mode is receive only.
-3	Mailbox is full and the send timed out.

Syntax

```
int       mreceive( handle, msg, maxlen );
MAILBOX *handle;
char *msg;
int maxlen;
```

Remarks *mreceive* is used to read a message from the mailbox into *msg*. *Handle* is obtained by calling *mopen*. *maxlen* specifies the maximum number of characters that will be copied into *msg*. The process executing *mreceive* will block indefinitely until a message becomes available in the mailbox. The call will fail if the owner process did not create the mailbox with an M_RECEIVE or M_SENDRECEIVE mode.

Return value

>= 0	Receive was successful. This also indicates the number of bytes

	received.
-1	Invalid mailbox handle.
-2	Mailbox mode is send only.
-3	Mailbox is empty.

Syntax

int *mreceivec*(handle, msg, maxlen, sleeptime);
MAILBOX *handle;
char *msg;
int maxlen, sleeptime;

Remarks

mreceivec is used to read a message from the mailbox into *msg*. *Handle* is obtained by calling *mopen*. *maxlen* specifies the maximum number of characters that will be copied into *msg*. The process will block for a maximum of *sleeptime* tics while waiting for an available message in the mailbox. The time unit is based on the timer interrupt frequency on an IBM PC. It is approximately 18 tics per second. The call will fail if the owner process did not create the mailbox with an M_RECEIVE or M_SENDRECEIVE mode.

Return value

>= 0	Receive was successful. This also indicates the number of bytes received.
-1	Invalid mailbox handle.
-2	Mailbox mode is send only.
-3	Mailbox is empty and the receive timed out.

9.2 Process manipulation routines :

#include "oslib.h" /* for all process manipulation routines */

Syntax void *snooze*(tics);
int tics;

Remarks *snooze* puts the caller to sleep for the specified number of *ticks*.The time unit is based on the timer interrupt frequency on an IBM PC. It is approximately 18 tics per second.

Syntax int *smunch*(pid);
int pid;

Remarks This will kill any process regardless of its type. Only a SYSTEM process is allowed to execute *smunch*.

Return value

0	Success.
-1	Process does not exist.
-2	Access violation.

Syntax int *kill*(pid);
int pid;

Remarks A process of type USER can be killed with *kill*. A SYSTEM process can be killed only if it is killing itself.

Return value

0	Success.
-1	Process does not exist.
-2	Access violation.

Syntax int *getpid*( );

Remarks *getpid* returns the caller's process id.

Syntax int *check_stack*( );

Remarks *check_stack* checks the caller's stack for overflow. This is done by checking the stack signature at the top of the stack. The stack signature is put there at process initiation time.

Return value

0	Stack has not overflowed.
-1	Stack has overflowed.
-2	Invalid pid.

Syntax int *get_stack_usage*( );

Remarks *get_stack_usage* returns the number of bytes used up on the caller's stack.

Syntax void *terminate*( );

Remarks A replacement for the Turbo C exit routine. Unlike exit *terminate* does not take any parameters.

Syntax int *createprocess*(process);
void (*process)( );

Remarks This is an internal system function and cannot be used by application programs. *Createprocess* creates a process from a function. System deamons can be written as functions and then converted to processes using this call. The process is assigned default values for priority, stacksize, number of mailbox handles and it's type(USER or SYSTEM).

Return value

-1	Error.
> 0	Pid of the created process.

Syntax int *pcreate*(process, priority, stacksize, nhandles, type);
void (*process)( );
int priority, stacksize, nhandles, type;

Remarks This is an internal system function and cannot be used by application programs. *Pcreate* creates a process from a function. System deamons can be

written as functions and then converted to processes using this call. The *priority, stacksize,* number of mailbox handles *nhandles,* and process type *type* can be set. *Priority* is either USER_PRI or SYSTEM_PRI and *type* can be USER or SYSTEM.

Return value

-1	Error.
> 0	Pid of the created process.

Syntax int *spawn*(progblk, priority, stacksize, handles, type);
struct program *progblk;
int priority, stacksize, nhandles, type;

Remarks *spawn* is used to create a process from a loaded program where *Progblk* represents the loaded program. This is an internal system function and is not available to application programs.

Return value

-1	Error.
> 0	Pid of the created process.

Syntax int *getnodeid*( );

Remarks Returns the network id of the host on which the caller is running.

9.3 Memory routines :

#include "oslib.h" /* for all memory routines */

Syntax void **dmalloc*(nbytes);
unsigned int nbytes;

Remarks *dmalloc* is used to dynamically allocate *nbytes* of uninitialized memory.

Return value

NULL	Requested amount of memory is not available.
non NULL	The address of the start of the block is returned.

Syntax void **dcalloc*(nbytes, nelements);
unsigned int nbytes, nelements;

Remarks *dcalloc* is used to dynamically allocate memory for *nelements* each of size *nbytes*. The allocated space is initialized to zeros.

Return value

NULL	Requested amount of memory is not available.
non NULL	The address of the start of the block is returned.

Syntax void *dfree*(memptr);
void *memptr;

Remarks Memory allocated with either *dmalloc* or *dcalloc* is freed using *dfree*.

Return value

None.

Syntax unsigned long *dmemleft*( );

Remarks *dmemleft* indicates the amount of memory remaining in the free pool. Because of external fragmentation, *dmemleft* is likely to return a value greater than the amount of memory than can actually be allocated in a contiguous block.

9.4 Window and I/O routines :

```
#include "oslib.h"          /* for all window and i/o routines */
```

Syntax int *wopen*( );

Remarks *wopen* is called to open a window. A handle is returned and this must be used in subsequent window operations.

Syntax int *wclose*(win);
int win;

Remarks This closes a window which was previously opened by *wopen*. The handle of the window is required.

Return value

0	Success.
-1	Error.

Syntax int *setattr*(win, attr);
int win;
char attr;

Remarks Set the attribute of the characters which are printed in the window *win*. Attribute values are as defined in the Turbo C manual.

Syntax int *getscr*(win);
int win;

Remarks Many windows are open in the system at any given time but only one window is displayed on the screen. *Getscr* forces *win* to be displayed on the screen when executed.

Syntax int *dprintf*(win, format, args);
int win;
char *format;
int args;

Remarks *dprintf* is a routine for outputting strings to a window *win*. *Format* corresponds to that in the C printf function. Only %d, %c and %s are allowed in the format. The window is scrolled when needed.

Syntax

```
int    dscanf( win, format, args );
int win;
char *format;
int args;
```

Remarks *dscanf* is a function for inputting data from the keyboard. The *format* corresponds exactly to the C scanf routine. Only %d, %c and %s are supported.

Syntax

```
int    loader( filename, srcnode, targetnode );
char *filename;
int srcnode, targetnode;
```

Remarks *loader* is used to load and run an executable file on any node on the network. The specified file *filename* is read from the source node *srcnode* and then transferred to *targetnode*. It is then executed as an independent process on *targetnode*. The length of *filename* which may include command line arguments should not exceed 80 characters.

Return value

0	Successful load.
-1	File does not exist on source node.
-2	Memory shortage on target node.
-3	Load error.

9.5 Network datagram service routines :

```
#include "network.h"          /* for all network routines */
```

Syntax

```
int    dg_open0( lservice_name, handle, msglen, msgcount );
char *lservice_name;
int *handle, msglen, msgcount;
```

Remarks *dg_open0* must be called before any other datagram services are called. *Dg_open0* opens a port for network communication and registers *lservice_name* with the name server. It also creates a buffer pool for the port with *msgcount* buffers, each of size *msglen* bytes. Finally the necessary mailboxes are allocated to the port. A *handle* is returned which is to be used when requesting other datagram services.

Return value

0	Success.
Non 0	Error.

Syntax

```
int    dg_findid0( handle, rservice_name, service_identifer );
int handle;
char *rservice_name;
struct sidtype *service_identifier;
```

Remarks *dg_findid0* is used to find out the SAP address associated with *rservice_name*. This is done by communicating with the name server since it maintains a table of service names and SAP addresses. To send datagrams to another process, the sender process should either know the receiver's SAP address or it's *rservice_name* which can then be used to get the SAP address.

Return value

0	Success.
Non 0	Error.

Syntax

```
int    dg_send0( handle, service_identifier, message, len );
int handle;
struct sidtype *service_identifier;
char *message;
int len;
```

Remarks *dg_send0* is used to send a datagram to another process. The SAP address of the other process(*service_identifier)* is obtained by calling *dg_findid0. Message* specifies the message to be transmitted and *len* indicates it's length.

Return value

0	Success.
Non 0	Error.

Syntax

```
int     dg_recv0( handle, service_identifier, buffer, len, msglen );
int handle;
struct sidtype *service_identifier;
char *buffer;
int len;
int *msglen;
```

Remarks *dg_recv0* is used to receive datagrams from other processes. The SAP address of the sending process is returned in *service_identifier. Buffer* is the location where the datagram is to be put and *len* indicates the buffer size. The datagram will be truncated if it's size is greater than *len.* The actual number of bytes stored in the buffer is returned in *msglen.* The process calling *dg_recv0* blocks indefinitely until a datagram is available.

Return value

0	Success.
Non 0	Error.

Syntax

```
int     dg_recvc0( handle, service_identifier, buffer, len, msglen,
    timeout );
int handle;
struct sidtype *service_identifier;
char *buffer;
int len;
int *msglen;
int timeout;
```

Remarks *dg_recvc0* is used to receive datagrams from other processes. The SAP address of the sending process is returned in *service_identifier. Buffer* is the location where the datagram is to be put and *len* indicates the buffer size. The datagram will be truncated if it's size is greater than *len.* The actual number of bytes stored in the buffer is returned in *msglen.* The caller will block only for *timeout* tics if a datagram is not available. Control will then return to the caller.

Return value

0	Success.
Non 0	Error.

Syntax

int *dg_pool0*(handle, msglen, msgcount);
int handle;
int msglen;
int msgcount;

Remarks

dg_pool is used to create a buffer pool for a port. The buffer pool is used to hold incoming datagrams if they arrive when a process is not blocked on a datagram read. However if the buffer pool itself becomes full then incoming datagrams will be discarded. An alternate way of creating a buffer pool is via *dg_open0*.

Return value

0	Success.
Non 0	Error.

Syntax

int *dg_close0*(handle);
int handle;

Remarks

A network port opened with *dg_open0* must be closed with *dg_close0* when it's purpose has been served. The buffer pool and mailboxes allocated to the port are freed and finally the logical service name which was registered with the name server during *dg_open0* is cancelled.

Return value

0	Success.
Non 0	Error.

9.6 Network virtual circuit routines :

#include "network.h" /* for all network routines */

Syntax

```
int   vc_open( lservice_name, handle, service_id, msglen,
          msgcount );
char *lservice_name;
int *handle;
struct sidtype *service_id;
int msglen, msgcount;
```

Remarks *vc_open* must be called before *vc_send* can be used. *Vc_open* first opens a port by calling *dg_open0*. The parameters passed to *dg_open* are *lservice_name, handle, msglen* and *msgcount*. *Service_id* is the SAP address of the target process to which a connection is to be made. The caller should call *dg_findid0* first to obtain the SAP address. *Vc_open* then communicates with the connection manager which opens a connection to the target process. A *handle* is returned which is required when requesting other virtual circuit services.

Return value

0	Success.
LOCAL_CONN_BAD	No space in local connection table.
TARG_PORT_BAD	Target port is not in use.
TARG_CONN_BAD	No space in target connection table.

Syntax

```
int   vc_send( handle, message, len );
int handle;
char *message;
int len;
```

Remarks Once a connection has been established by calling *vc_open*, a process sends messages on the connection using *vc_send*. *Vc_send* ensures that the messages reach the target process and also guarantees the ordering of messages sent.

Return value

0	Success.
LOCAL_CONN_BAD	Local connection entry is messed up. Do not use the connection anymore. Close it immediately if possible.
VC_BAD_SEQNO	The message sent was not the one the target expected. Do not use the

	connection anymore. Close it immediately if possible.
VC_TGTPORT_BAD	Target port is either closed or the owner of the port changed. In either case the connection is bad and is not to be used anymore. Close the connection immediately if possible.
VC_SEND_FAILED	The message could not be sent. This possibly indicates that the target process is fine but for some unknown reason it is not reading the messages sent.

Syntax

int *vc_close*(handle);
int handle;

Remarks

A connection opened by *vc_open* should be closed with *vc_close* when it's purpose has been served.

Return value

0	Connection has been closed.
CONN_NOT_FREED	Connection could not be closed.
