A MESSAGE-PASSING SYSTEM FOR THE LARGE-GRAIN DATA-FLOW MACHINE

JEFFREY THIERET
DR. T. G. LEWIS
DEPARTMENT OF COMPUTER SCIENCE
OREGON STATE UNIVERSITY
APRIL 28, 1986

ABSTRACT

The Large-Grain Data-Flow (LGDF) Machine is a tightly-coupled multi-processor consisting of several independent nodes, upon each of which may run one or more tasks. These tasks must be able to communicate and synchronize with each other. This document describes a message-passing system developed to meet these requirements.

The message-passing system allows any task on an arbitrary node to send information to any other task on an arbitrary, possibly the same, node. The system is based on the use of shared memory. Messages may be of any length, subject only to the constraints of memory size. Except for a header of control information, the system places no interpretation upon the contents of the message. Using performance data automatically provided, the user can easily tune the system to enhance performance.

The first section of this paper presents a brief introduction to the concept of large-grain data-flow. This is followed by a description of the message-passing system. All components of the system from the hardware to the high level system calls are discussed. The third section is a user's manual. This is a guide to the low level system calls, to system tunable parameters, and to deadlock conditions. The paper is concluded with a discussion of possible problems with and extensions to the system.

TABLE OF CONTENTS

1. Introduction 2
2. The Message-Passing System 2
   - 2.1 Design Criteria 3
   - 2.2 The Hardware 5
   - 2.3 Shared Memory 8
   - 2.4 PIO_MP_POSTMAN 13
   - 2.5 The VRTX Kernel 14
   - 2.6 Low Level System Calls 15
     - 2.6.1 PC_SM_ALLOCATE/PC_SM_DEALLOCATE 15
     - 2.6.2 PC_MP_POST 15
     - 2.6.3 PC_MP_PEND 16
   - 2.7 High Level System Calls 17
     - 2.7.1 PC_MP_SEND 17
     - 2.7.2 PC_MP_RECEIVE 18
   - 2.8 An Example 18
3. User's Manual 23
   - 3.1 Low Level System Calls 23
   - 3.2 System Tunable Parameters 28
   - 3.3 Deadlock 29
4. Conclusions 30
5. References 32
6. Appendices 34

1. INTRODUCTION

In the traditional control-flow model of computation, a grain, which is a unit of work to be performed, is processed only when the control is present at that grain. In other words, a grain is "fired" when the program counter points to it. In the data-flow model, however, a grain may fire as soon as it has all its inputs. There is no notion of centralized control, as in a control-flow machine. The order in which the grains are fired is determined solely by the availability of inputs.

Data-flow machines typically have several units for processing grains in parallel. The architectural structure of these processing units is a function of grain size. In a small-grain data-flow machine, a grain may consist of a single machine instruction such as an add or a multiply. The processing units are designed accordingly, perhaps being simple alu's. In contrast, large-grain data-flow machines have typical grain sizes of at least 50 machine instructions. These grains may correspond to program modules or procedures in a high-level language. In this paper, these grains are called tasks. The processing units in these machines are traditional control-flow processors, each of which executes each "firing" task in a sequential manner. However, just as in small-grain computation, the "firing" of the tasks is a function of data availability and not of control "availability". Readers are referred to [2] [3] [9] [12] for a more detailed discussion of both large and small-grain data flow computation.

The processing unit of the LGDF Machine is a node consisting of two central processors. The machine will eventually have 5 nodes. Each node may have up to 64 tasks resident on it at any point in time. As soon as a task has all its inputs, it executes. If it produces any outputs, these may be used as the inputs for some other task. This "other" task may or may not reside on the same node as the outputting task. The message-passing system discussed in this paper is a means of transferring the outputs of one task (the source) to the inputs of another (the destination) as well as a means of controlling the firing of the tasks; i.e., no task should execute before it has all of its inputs.

2. THE MESSAGE-PASSING SYSTEM

In the following discussion, the reader is guided through the

message-passing system from the hardware level to the level of the highest system calls.

Briefly, the message-passing system works as follows: when a task wants to send a message to another task, it places a system call to copy the message from its local memory to the shared memory on its node. Periodically, each node i checks shared memory on each of nodes j (j != i) for messages to tasks residing on node i. These messages are then copied from shared memory on node j to shared memory on node i. A task may receive a message by placing a system call that will copy the message from shared memory on its node to its local memory.

The above discussion illustrates the fact that the message-passing system makes the LGDF Machine something of a hybrid between traditional shared memory multi-processors and distributed multi-processors. In traditional shared-memory machines, such as those discussed in [1] [10] [12] [16] [19] [20], a message is placed in shared memory where the destination task can access it immediately. In a distributed system, the message must be transported from the sender to the receiver by kernel routines accessing a network. Only after the message has traversed the network can the destination task access it. References relating to distributed computing abound, including [4] [5] [6] [7] [8] [15] [17] [18].

The message-passing system utilized in the LGDF Machine more closely resembles a distributed communication system than a traditional shared memory communication system. In particular, the Accent kernel described in [18] has many interesting similarities with the LGDF message-passing system, although it is an order of magnitude more complex.

Section 2.6 gives an example of the behavior of the message-passing system: the reader may wish to read that section first before proceeding with the more detailed discussion below.

2.1 DESIGN CRITERIA

The message-passing system has been designed to meet the following criteria:

1) The application task should not have to know the details of the system. It should only know what the message is and

the identification number (id) of the destination task.

2) The message-passing system should only minimally impact the task processing, either in terms of cpu time or in terms of memory space.
3) A task should be able to send/receive messages to/from any other task in the system regardless of the node on which the destination/source task resides.
4) The user's interface to the message-passing system should be consistent regardless of the location of or type of task sending/receiving.
5) A task should be able to receive messages both in FIFO order based on arrival time over all messages and in FIFO order based on arrival time for only those messages from a desired source task.

Criteria 1, 3, and 4 above serve to hide details of the system from the user. To borrow a term from the field of distributed processing, the message-passing system is "network transparent". The benefits of such transparency include ease of programming, ease of software modification, and facilitated process (task) migration [5] [17]. Although the LGDF Machine can not be classified as a distributed system, it can benefit from such transparency.

Criterium 2 serves to speed the processing of tasks. Communication overhead can have a significant effect on task processing speed. Multi-processing systems strive to minimize this overhead because, as the communication overhead increases, the benefits, in terms of execution speed, of parallelism decrease [12]. The LGDF Machine attempts to resolve this conflict by having two processors on each node, one for task computation and one for task communication.

Criterium 5 gives the user greater flexibility as to which messages he may receive. He is no longer restricted to receiving only the message at the head of the queue; he may also receive the first message sent from a particular task, whether or not it is at the head.

There are certain criteria that the system was not designed to meet:

1) The system will not be fault tolerant: the failure of a node

or a task after system boot may cause a system deadlock.

2) The system will have no facility for automatic message re-transmission: messages are assumed to arrive intact.
3) The system will not automatically send a message receipt acknowledgment to the source task.
4) The system will not provide broadcasting, multicasting, or global streaming services: these can, however, be built up from the basic service provided.

2.2 THE HARDWARE

Figure 1 illustrates the overall structure of the LGDF system. There will be 6 nodes, numbered 0-5. These nodes are connected by a VME bus. Each node may serve as bus master. Nodes 1-5 run identical kernels and are processing units for application tasks.

Node 0 is the SCB (System Control Board). This node acts as a conduit for data flowing to and from the front-end processor, which will eventually be an Apple Macintosh micro-computer. It does not act as a processing unit for application tasks. It coordinates the activities of the other nodes during system boot and acts as multiple servers, providing file access, clock support, and error handling services to application tasks on the other nodes. The message-passing system does not distinguish between the SCB and the other 5 nodes. Server tasks are activated by messages in the same manner as application tasks. This is, again, the paradigm of network transparency at work. More information on the concept of servers may be found in [4] [7] [8]. In the discussion to follow, only nodes 1-5 will be called "nodes", the SCB node will simply be called the SCB.

Figure 2 illustrates the structure of each node. On each node reside two Motorola 68010 processors. One of these, the Pc, performs the computations of the tasks themselves. The other, the Pio, performs all the work involved in the inter-node message-passing between tasks executing on the Pc's. Note, however, that intra-node message passing does not involve the Pio. It is hoped that by having a separate processor for inter-node message-passing, the Pc will be able to process tasks at a rate only minimally affected by the inter-task communication overhead.

The LGDF Machine employs a hierarchical memory scheme. The Pc has

its own local memory (512K bytes RAM). It is inaccessible to the Pio and is where the task code and data segments will reside. The Pio also has its own local memory (16K bytes RAM). This is where the data segment for the PIO_MP_POSTMAN process, discussed in Section 2.4, will reside.

Also on each node is 128K of shared RAM. This RAM may be accessed from the local Pc, the local Pio, and any remote Pio through the VME bus. Arbitration logic handles simultaneous requests for access. That portion of the total shared memory address space that is physically located on node i is said to be node i's local shared memory (LSM). That portion of the total shared memory address space that is physically located on node j (j!=i) is said to be node i's remote shared memory (RSM).

The bus interrupt structure allows 7 priority levels of bus interrupts. Any node may interrupt any other node or the SCB. Each node (and the SCB) may be programmed to respond to interrupts of only certain priority levels. Each node (and the SCB) is currently programmed to respond to interrupt levels i+1, where i is the node id, and 7. For example, node 3 will respond to interrupts of priorities 4 and 7.

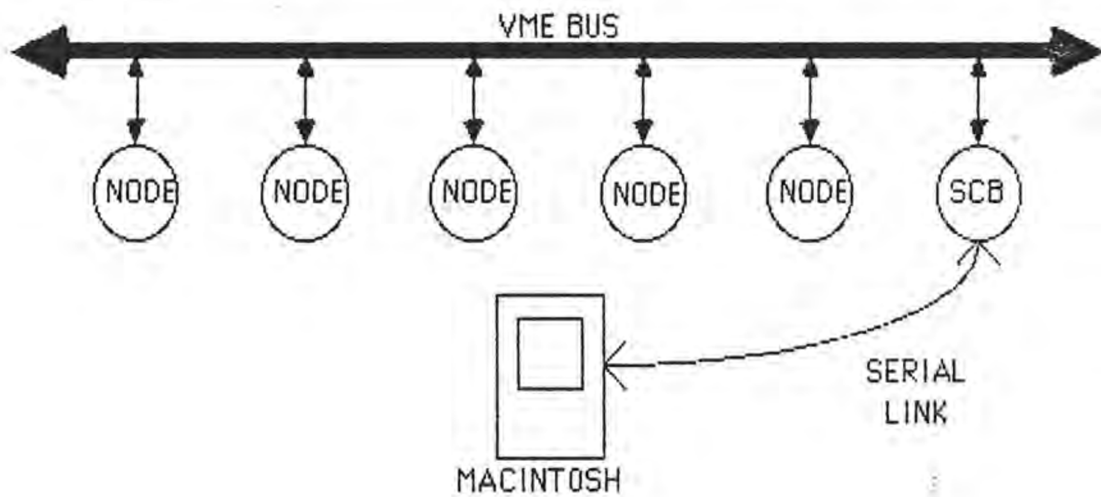

FIGURE 1. THE LGDF SYSTEM

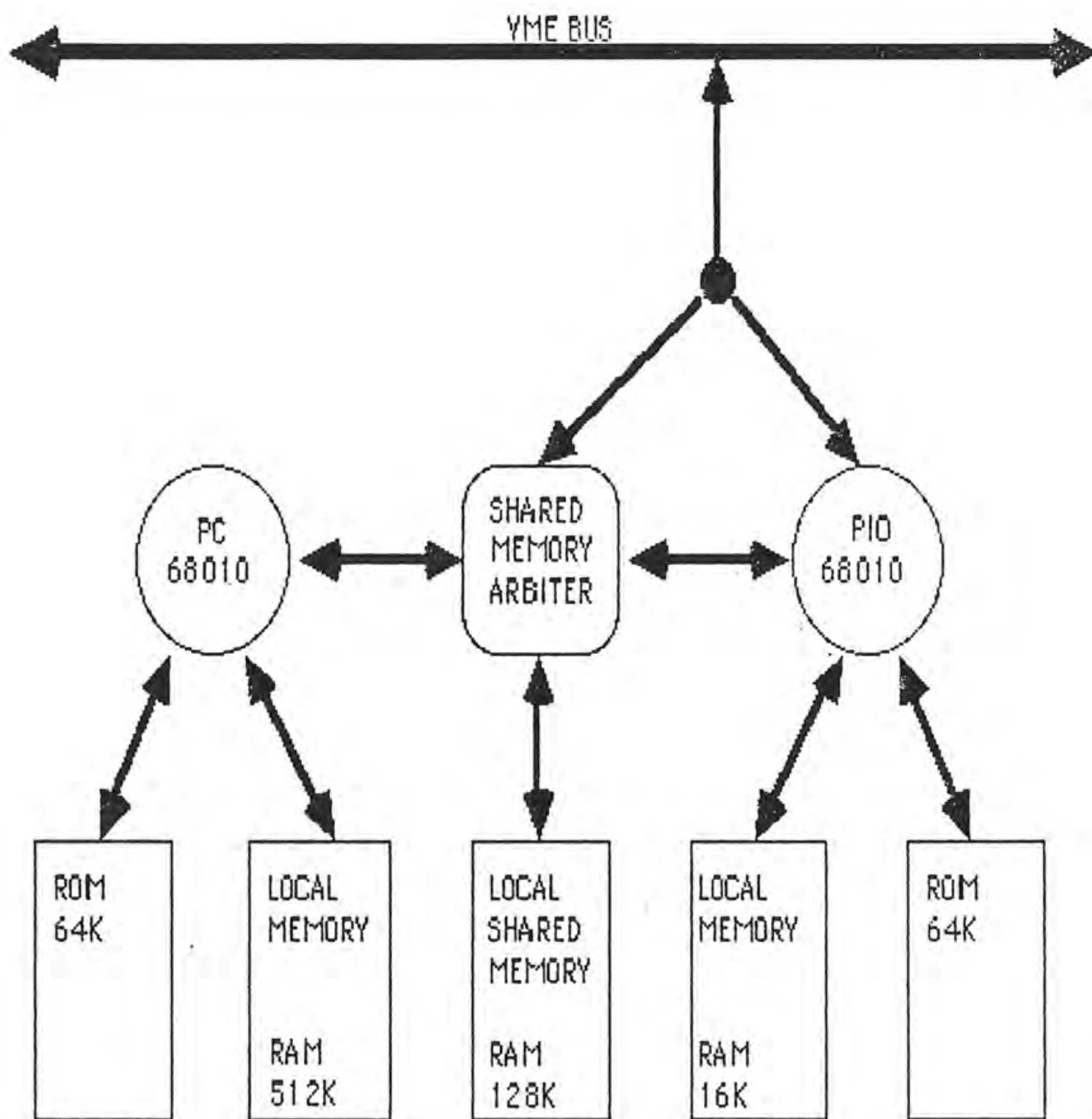

FIGURE 2. LGDF NODE STRUCTURE

2.3 SHARED MEMORY

Shared memory structure is illustrated in figure 3. Shared memory on each of nodes 1-5 has an identical structure. It is partitioned into 5 areas: a configuration table, a task table, an area for system variables, a queue area, and a message area.

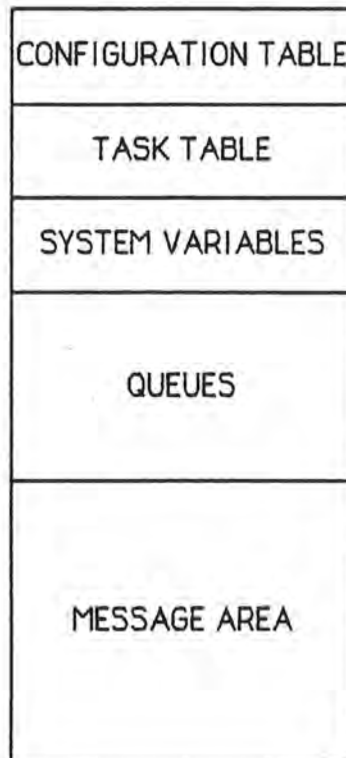

FIGURE 3. SHARED MEMORY STRUCTURE

The Pc multi-tasker and the message-passing system are dynamically reconfigurable at system boot. At system boot, the SCB will copy values

for the current configuration into the configuration table. The message-passing system and the multi-tasker are then configured accordingly by the boot routine on the Pc. Each node is identically configured. Those configuration parameters relevant to the message-passing system are discussed further in Section 3.2, System Tunable Parameters.

Each task in the system is assigned a unique task id. Id 0 is reserved for system use. The maximum number of user tasks that may be resident in the system is currently restricted to 255. The maximum number of tasks resident on any single node is currently restricted to 64. The task table has an entry, indexed by task id, for each task in the system. Included in this entry are the id of the node on which the task resides and the id of its corresponding task queue. All tasks must be created and bound to processors and task queues before any task is allowed to execute. Because of this static binding, it is most convenient to place identical copies of the task table in shared memory of each node, thus reducing bus traffic. Task table structure is illustrated in figure 4. Task queue numbers are distinct on each node, but not between nodes. For example, every node has a task queue with id 17, but there is at most one task queue with that id on any one node.

System variables consist of "tickets" to be used for mutual exclusion, heads and tails of queues, pointers to the various areas of shared memory, such as a pointer to the first free block in the message area, and compaction flags (discussed below) for each task queue. Also included are various counters used to provide feedback to the user, enabling him to tune the system configuration to enhance performance. These counters keep track of the number of times the various queues are found to be full or to be empty and the number of times LSM is found to be full.

There are three classes of queues in the queue area of shared memory: send queues, acknowledge queues, and task queues. These queues closely resemble the "kernel ports" and "data ports" of the Accent kernel [18]. Each queue has a head and a tail pointer and is filled/emptied in a cyclic manner. All queues within a given class have the same length but queue lengths between classes may vary. These lengths are system tunable parameters. All queue elements are pointers to messages.

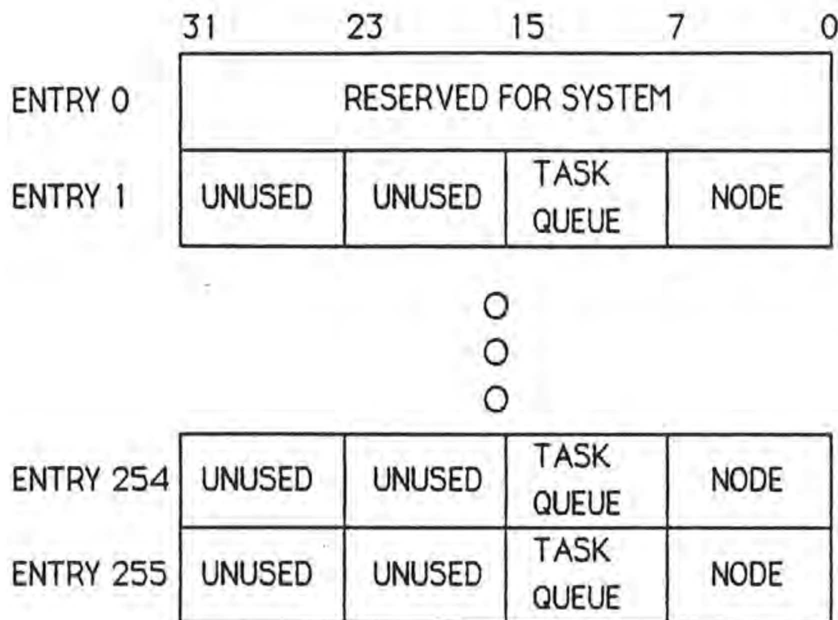

FIGURE 4. THE TASK TABLE

There are 6 send queues; 1 for each node and one for the SCB. These hold pointers to outgoing messages in LSM that are yet to be copied to RSM on the destination node. Mutual exclusion must be maintained on the tail pointer of each send queue because several Pc tasks may be attempting to append to the queue simultaneously. This is accomplished by disabling the VRTX (see Section 2.5) task scheduler.

As in the case of send queues, there are 6 acknowledge queues, one for each node and one for the SCB. These hold pointers to the original messages in shared memory of the source node that have been copied into LSM of the destination node. Mutual exclusion mechanisms are not required for this class of queue.

Task queues hold pointers to incoming messages in LSM of the destination node that are yet to be received by the destination task. There is one task queue for each task on the Pc. Task queues are not strict FIFO queues. Pointers are appended to a task queue only at its tail, but they may be removed from any location in the queue, not just the head. Items

removed from any queue location other than the head are zeroed out. These zeroed items are removed by compacting the queue. Associated with each queue is a compaction flag, indicating whether or not the queue needs to be compacted. When a system task attempts to append to a task queue and finds it full, this flag is checked. If the flag indicates that compaction is necessary, the queue is compacted and the attempt to append is repeated. The "random" access of task queues is similar in concept, if not in implementation, with the prioritized port access of Accent [18].

Mutual exclusion must be maintained on the entirety of each task queue. At any instant, several Pc tasks and PIO_MP_POSTMAN (see Section 2.4) on the Pio may be simultaneously attempting to access the same task queue. To insure Pc-Pc mutual exclusion, the VRTX task scheduler is disabled. A version of Lamport's bakery algorithm [13] [14] is then used to insure Pc-Pio mutual exclusion. Each task queue has two corresponding tickets; i.e., locks are on individual queues, not the group of task queues as a whole.

The message area is divided into fixed-length blocks and structured as a linked-list. The block size is a system tunable parameter. The last 32 bits of each block is a pointer to the start of the next block. When a message is allocated, the required number of blocks are removed from the free list. When it is deallocated, the blocks are returned to the free list. The pointer to the head of the free list, the first free block, is kept in the system variables area of shared memory. At any point in time, several tasks on the Pc and PIO_MP_POSTMAN on the Pio may be simultaneously attempting to allocate/deallocate message space. Safety is maintained by controlling access to the first free block pointer. This is accomplished by disabling the VRTX task scheduler to resolve Pc-Pc competition and using Lamport's Bakery Algorithm to resolve competition between the active Pc task and PIO_MP_POSTMAN.

Messages may be of any length and may occupy several, possibly non-contiguous, blocks in the message area. The link field in the last block of a message will be NULL. Every message has a header. Included in this header are the source task id, the source node id, the destination task id, the destination node id, and the message length. Message structure is illustrated in figure 5. This example illustrates a message whose length requires three message blocks.

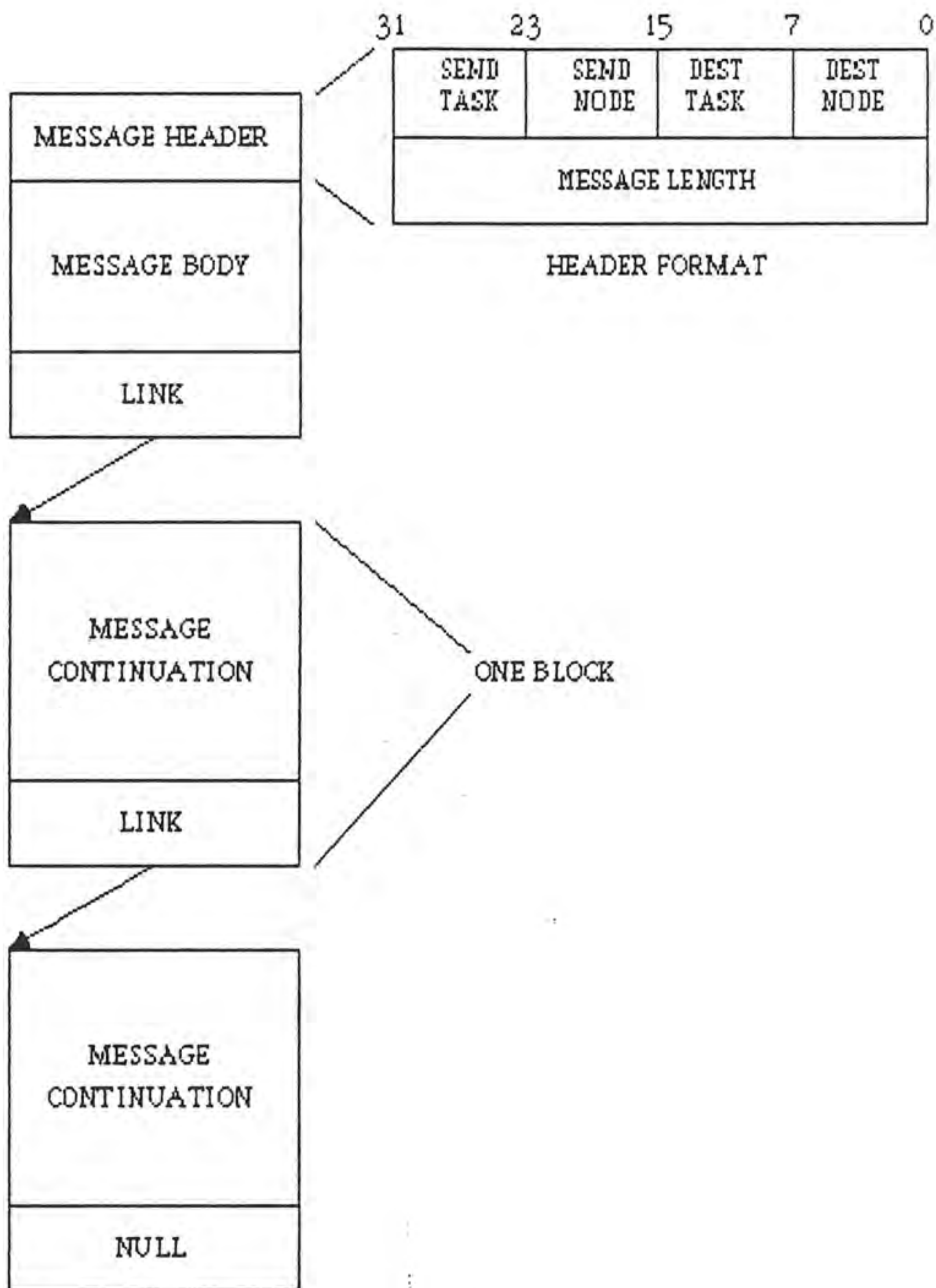

FIGURE 5. MESSAGE STRUCTURE

2.4 PIO_MP_POSTMAN

Each Pio runs a single process continuously. This process is called PIO_MP_POSTMAN. Each PIO_MP_POSTMAN executes in parallel with every other PIO_MP_POSTMAN in the system; however, to reduce traffic on the VME bus, it is allowed access to the bus only when it is its turn. Turns are decided in round-robin fashion. As soon as one PIO_MP_POSTMAN is done, it interrupts the next PIO_MP_POSTMAN in the cycle, telling it to proceed, and then performs a busy-wait until it is again its turn. During its turn, each PIO_MP_POSTMAN performs the following in the order given:

1) Looks on each node, except its own, at its corresponding acknowledge queues. For example, the PIO_MP_POSTMAN running on the Pio of node 3 will examine acknowledge queue 3 on each of nodes 0, 1, 2, 4, and 5. Each of these queues contain pointers to messages in LSM of node 3 that have already been copied to LSM of the destination node. PIO_MP_POSTMAN deallocates the LSM space used by these messages, returning the message blocks to the linked-list of free blocks.
2) Looks on each node, except its own, at its corresponding send queues in a manner analogous to that used for acknowledge queues. Each of these queues contain pointers to messages in LSM of the source node that PIO_MP_POSTMAN must copy into its LSM. For each pointer found in these send queues, PIO_MP_POSTMAN performs the following in the order given:

    a) Determines, by examining the message header, the length of the corresponding message.
    b) Attempts to allocate space in its LSM for the message. If the attempt fails (due to lack of space), it increments the LSM full counter and relinquishes its turn. When it is again its turn, it performs step 1 above in hopes of freeing up LSM space and then tries to allocate the space again. It repeats this process until the allocation is successful.

c) Copies the message from RSM to LSM. There are now two copies of the message, a remote copy on the source node and a local copy on the destination node.
d) Determines, by examining the message header, the destination task id and examines the task table to find its corresponding task queue.
e) Attempts to append a pointer to the local message to the appropriate task queue. If the attempt fails (because the queue is full), it checks the compaction flag associated with the queue. If the flag indicates that compaction is necessary, it compacts the queue and repeats the attempt to append. If the flag indicates that compaction is not necessary, it increments the full counter for that task queue and relinquishes its turn. When it is again its turn, it tries again. This process is repeated until the attempt succeeds.
f) Attempts to append a pointer to the remote message to the appropriate acknowledge queue. If the attempt fails (because the queue is full), it relinquishes its turn. When it is again its turn, it tries again. This process is repeated until the attempt succeeds.

3) Interrupts the next PIO_MP_POSTMAN and then performs a busy-wait until it is interrupted.

2.5 THE VRTX KERNEL

VRTX (Virtual Real-Time Executive) is a set of prepackaged kernel routines. These routines provide many different services, such as multi-tasking, semaphore operations, and memory management [11]. The operating system running on the Pc incorporates these VRTX routines. All routines are available, but the message-passing system uses only those routines related to multi-tasking.

Tasks running on the Pc will be scheduled on a priority basis and by round-robin within a priority class. As the system is currently configured, all tasks have the same priority, thus the scheduling is strictly time-sliced. VRTX calls used by the message-passing system

serve to enable/disable the scheduler, thus enforcing mutual exclusion for certain critical operations, and to suspend execution of a task for a given time interval.

2.6 LOW LEVEL SYSTEM CALLS

The low level system calls are: PC_MP_POST, PC_MP_PEND, PC_SM_ALLOCATE, and PC_SM_DEALLOCATE. These calls allocate /deallocate space for a message in LSM (PC_SM_ALLOCATE /PC_SM_DEALLOCATE), append a message pointer to the appropriate queue (PC_MP_POST), and remove a message pointer from the appropriate queue (PC_MP_PEND). Readers are referred to Appendix A for a brief discussion of the naming convention used for operating system programs.

2.6.1 PC_SM_ALLOCATE/PC_SM_DEALLOCATE

PC_SM_ALLOCATE is a function that attempts to allocate the appropriate number of message blocks for the message size specified. It is passed the message size and a reference parameter into which will be copied (if the allocation was successful) a pointer to the start of the allocated message list. If sufficient free blocks are available, PC_SM_ALLOCATE removes them from the free list of blocks, places NULL in the link of the last block in the allocated list, and copies a pointer to the first block of the allocated list into the specified reference parameter. If sufficient space is not available, it increments the LSM full counter. The return value of this function indicates whether or not the allocation was successful.

PC_SM_DEALLOCATE returns a list of message blocks, whose starting address is passed by the caller as a parameter, to the free list. It has no return value.

Both of these calls go through the mutual exclusion mechanisms described in Section 2.3 for access to the first free block pointer of the message area.

2.6.2 PC_MP_POST

PC_MP_POST appends a message pointer to either the destination node

send queue, if the destination task is remote, or the destination task queue, if the destination task is local. It is passed two parameters: the id of the node on which the destination task resides and a pointer to the message (in LSM) to be sent. It performs as follows:

1) Compare the source and destination node id's.
2) If they are equal, determine, by examining the message header, the destination task id. Look up the corresponding entry in the task table to get the task queue id for that task. Attempt to append the pointer to the task queue. If the attempt fails (due to a queue full condition), check the compaction flag associated with the queue. If it indicates that compaction is necessary, compact it and then repeat the attempt to append. If the flag indicates that no compaction is necessary, increment the full counter for that task queue, suspend for NAPTIME clock ticks (see Section 3.2), and then try again. When the attempt succeeds, return to the caller.
3) If they are not equal, attempt to append the message pointer to the appropriate send queue. For example, if the message is going to node 4, the pointer gets appended to send queue 4. If the attempt fails (due to a queue full condition), increment the full counter for that send queue, suspend for NAPTIME clock ticks, and then try again. When the attempt succeeds, return to the caller.

Note that in neither case 2 nor 3 above will PC_MP_POST return to the caller until it is successful.

2.6.3 PC_MP_PEND

PC_MP_PEND removes a message pointer from the calling task's task queue and returns it to the calling task. It is passed two parameters: the destination task id and the desired source task id. It performs as follows:

1) Examine the source task id.
2) If the source task id is zero, retrieve the pointer at the

head of the destination task queue and return it to the caller. If the queue is empty, increment the empty counter for that task queue, suspend for NAPTIME clock ticks, and then try again.

3) If the source task id is non-zero, retrieve the first (in FIFO order) pointer to a message from the specified source task that is encountered in the destination task queue, whether or not that pointer is at the head of the queue. Return this pointer to the caller. If the queue contains no pointer to a message from the desired task, increment the empty counter for that task queue, suspend for NAPTIME clock ticks, and then try again.

Note that in neither case 2 nor 3 above will PC_MP_PEND return to the caller until it has found the pointer for which it is seeking.

2.7 HIGH LEVEL SYSTEM CALLS

These calls, PC_MP_SEND and PC_MP_RECEIVE, are the application interface to the communication system. These calls have not been implemented by the author. The following two sub-sections discuss these calls in a general sense, omitting application-specific details.

2.7.1 PC_MP_SEND

PC_MP_SEND initiates the message-passing process by formatting a message in LSM and calling PC_MP_POST. When an application task calls PC_MP_SEND, it passes the destination task id and a pointer to the information (in Pc local RAM) to be sent. PC_MP_SEND performs as follows:

1) Examine the task table entry for the destination task. If either the node id or the task id are invalid, return an error code.
2) Attempt to allocate space in the message area of LSM for the message by calling PC_SM_ALLOCATE. If the attempt fails, PC_MP_SEND will suspend for NAPTIME clock ticks

and then try again. The attempt is repeated until it is successful.

3) Format the message header.
4) Copy the information to be sent into the message blocks.
5) Call PC_MP_POST, passing it the destination node id and a pointer to the message in LSM.
6) Return a successful code as soon as PC_MP_POST returns.

2.7.2 PC_MP_RECEIVE

PC_MP_RECEIVE calls PC_MP_PEND to get a pointer to an incoming message and then dis-assembles the message, copying the message body into the desired location of Pc local memory. It is passed the task id of the desired source task (as a reference parameter) and a pointer to where the message is to be copied. It performs as follows:

1) Determine the calling task's task id.
2) Call PC_MP_PEND, passing it the source and destination task id's.
3) When PC_MP_PEND returns a pointer to the message, copy the message contents, without the header, into the area of Pc local memory specified. Place the source task id in the task id reference parameter.
4) Deallocate (by a call to PC_SM_DEALLOCATE) the LSM space used by the message.
5) Return to the caller.

2.8 AN EXAMPLE

Figures 6 and 7 illustrate the message-passing system. Message acknowledgements are illustrated separately because of lack of space on the first figure. These figures, along with the following example, should help to clarify the message-passing process. Suppose that task 27 wants to send a message to task 52. It has no idea on which node task 52 resides; it knows only what the message is and that it is to be sent to task 52. Further suppose that the task table entries for these tasks are as illustrated in figure 8.

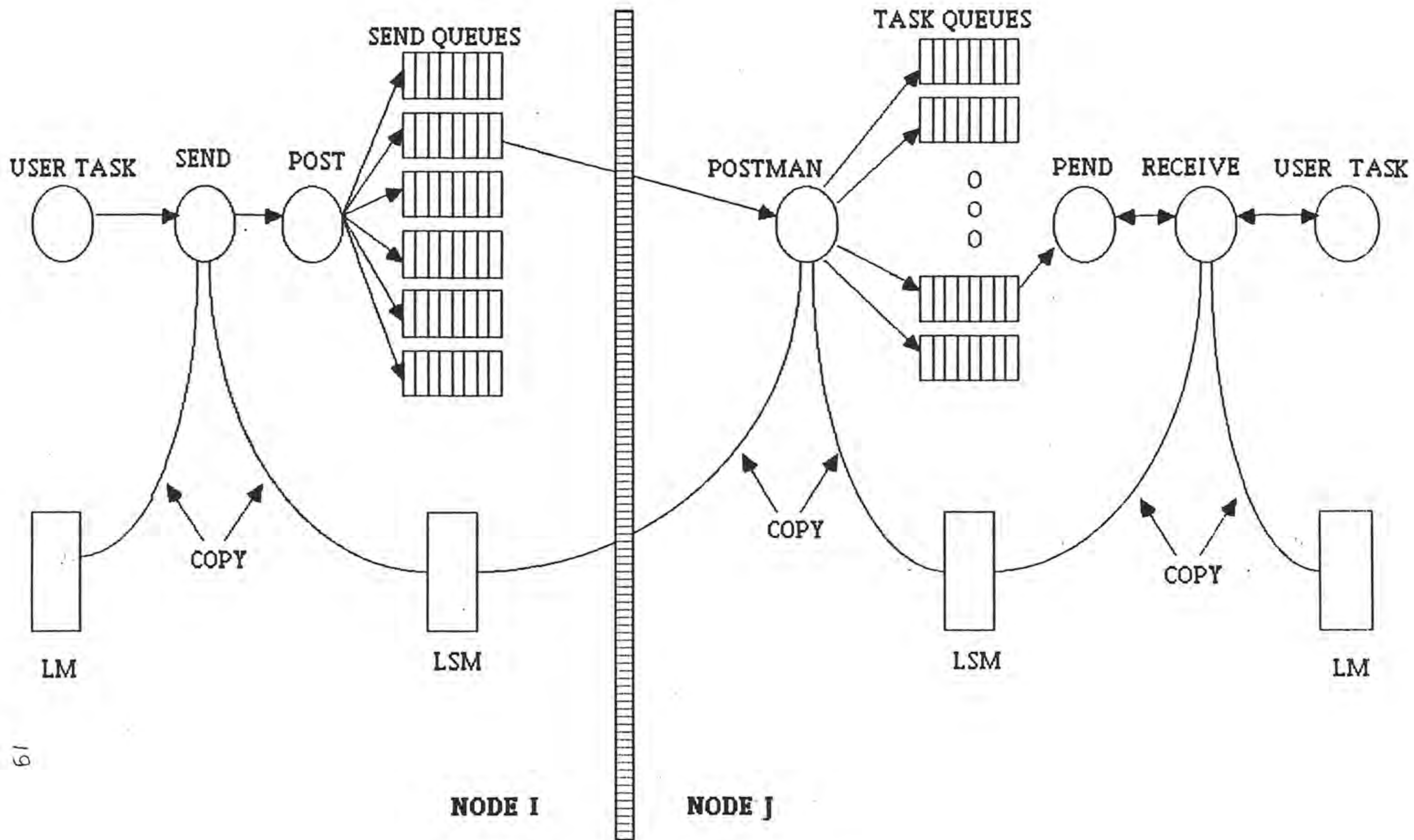

FIGURE 6. THE MESSAGE-PASSING SYSTEM (PART 1)

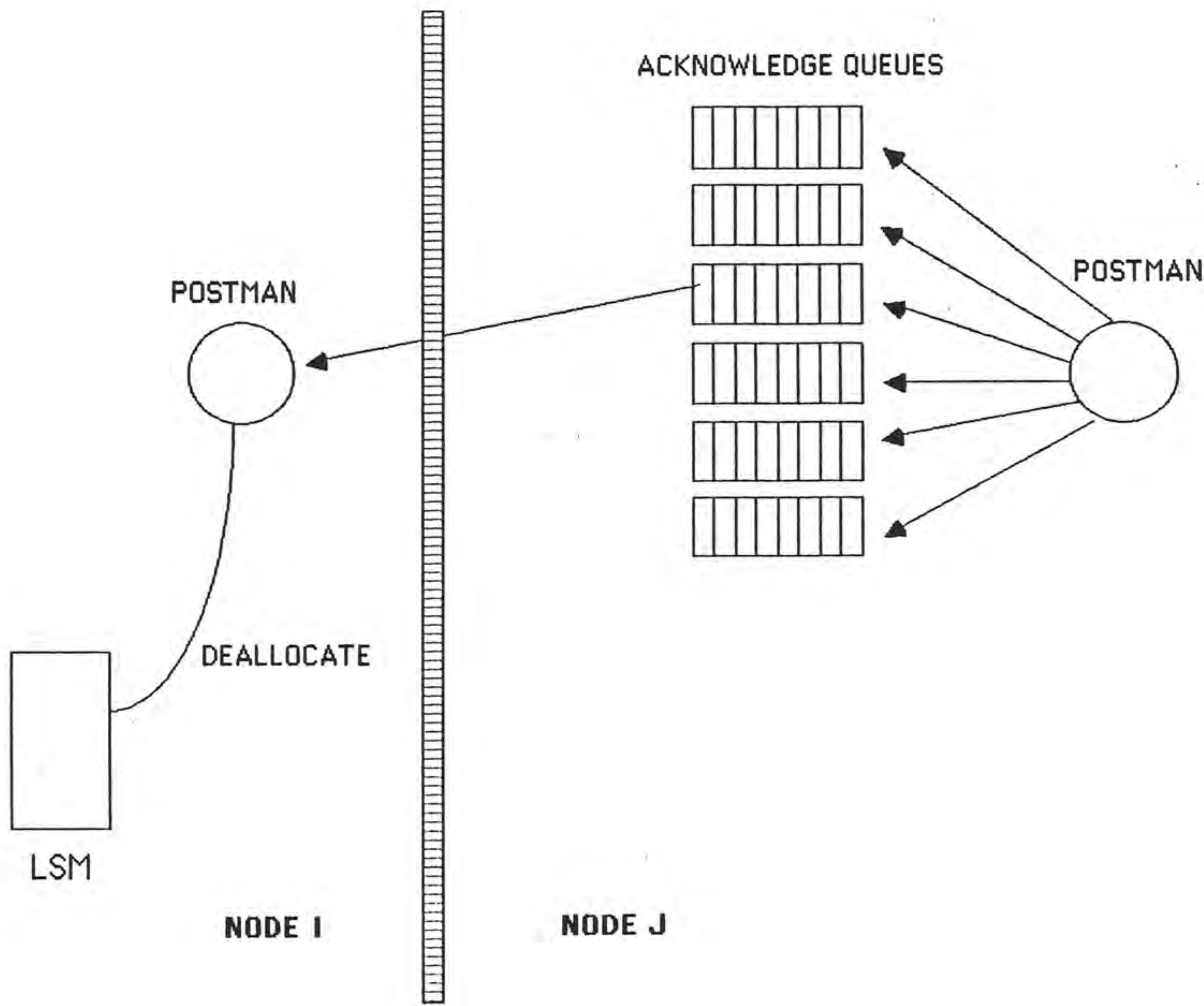

FIGURE 7. THE MESSAGE-PASSING SYSTEM (PART 2)

Task 27 calls PC_MP_SEND, passing it a pointer to the information to be sent and the task id 52. PC_MP_SEND checks to task table entry for task 52 to insure that its sub-entries are valid. PC_MP_SEND looks in the task table for the calling task's node id, 2. It also retrieves the destination task's node id, 3. An appropriate size list of message blocks is

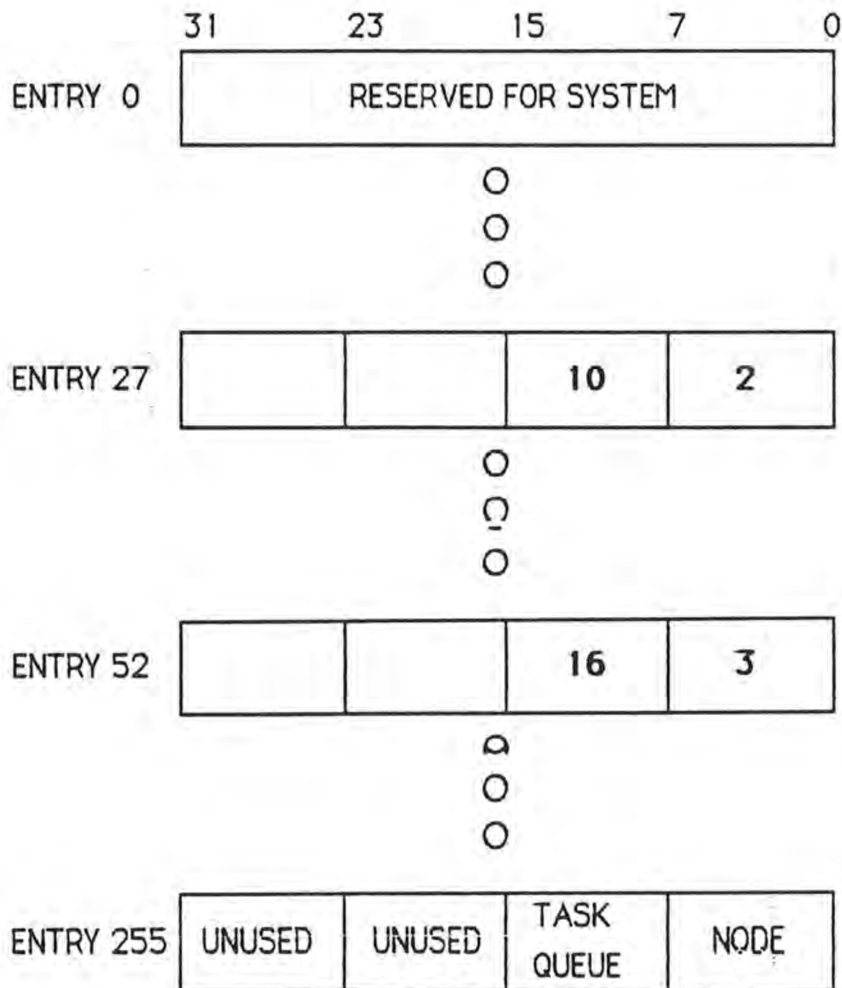

FIGURE 8. THE TASK TABLE FOR THE EXAMPLE

allocated by a call to PC_SM_ALLOCATE and the message and header information is copied into this list. PC_MP_SEND will then call PC_MP_POST, passing a pointer to the start of the message list and the destination task's node id.

PC_MP_POST compares the destination task's node id, which is 3, with its own node id, 2. Because they are different, the pointer is placed in send queue 3. (Had they been the same, the pointer would have been placed in task queue 16). PC_MP_POST then returns to PC_MP_SEND, which then returns to task 27. Task 27 proceeds without waiting for the message to be received.

When PIO_MP_POSTMAN on node 3 gets its turn to use the VME bus, it will, among other things, examine send queue 3 on node 2. It will find the pointer to the message and will copy the message into its own LSM. When the copy is complete, it will place a pointer to the local copy of the message into task queue 16, the task queue corresponding to task 52. It also places the remote message pointer in acknowledge queue 2.

When PIO_MP_POSTMAN on node 2 gets its turn, it will, among other things, examine acknowledge queue 2 on node 3 and find the message pointer. It will then deallocate the LSM space used by the message.

In the meantime, task 52 has executed a call to PC_MP_RECEIVE, specifying that it wants to receive a message from task number 27 and that the message is to be copied into address A of Pc local memory. PC_MP_RECEIVE calls PC_MP_PEND, passing it the source and destination task id's.

PC_MP_PEND examines the task table and finds that the corresponding task queue for task 52 is task queue 16. It then examines this queue for a pointer to a message from task 27. In this example, it is lucky; it finds such a pointer on its first scan through the queue. (Had it not found it on the first scan, it would have continued scanning until it had done so). This pointer is returned to PC_MP_RECEIVE.

PC_MP_RECEIVE copies the message, without the header, from LSM to address A in Pc local memory. It then deallocates the LSM space used by the message by placing a call to PC_SM_DEALLOCATE. Following the deallocation, it returns to task 52. Task 52 now continues processing.

3. USER'S MANUAL

The following three sub-sections discuss the details needed to utilize the low level system calls, to tune the system parameters to enhance performance, and to recognize and avert deadlock.

3.1 LOW LEVEL SYSTEM CALLS

PC_SM_ALLOCATE()

```
#include SYS_DEFINES:LGDF

int pc_sm_allocate(msg_length,msg_ptr)
    long msg_length;
    long **msg_ptr;
```

msg_length - the message length expressed as long words (4 bytes). This length includes the message body and message header but excludes the link field (1 long word) in the message blocks.

msg_ptr - on return, if the allocation was successful, this reference parameter will contain a pointer to the start of the first block of the allocated message list. If the allocation was unsuccessful, the value is indeterminate and should not be used.

PC_SM_ALLOCATE will attempt to allocate the required number of blocks from the message area of local shared memory. Upon successful allocation, the address of the start of the first allocated block will be placed in the reference parameter msg_ptr and a value CAN_ALLOCATE will be returned. CAN_ALLOCATE is defined in the file SYS_DEFINES:LGDF, a copy of which is included in the program listings folder.

If the allocation failed (due to lack of space), msg_ptr will be indeterminate and a value CANT_ALLOCATE (also defined in SYS_DEFINES:LGDF) will be returned.

Programs copying to/from a list of message blocks must be aware that the list may occupy non-contiguous blocks. The program must know the blocksize (BLKSIZE) in order to know when to examine the link. The link field is the last long word in each block. Because BLKSIZE is expressed in long words, the link field is at offset BLKSIZE - 1 in each block.

PC_SM_DEALLOCATE()

```
void pc_sm_deallocate(first_msg_block_ptr)
    long *first_msg_block_ptr;
```

first_msg_block_ptr - a pointer to the first message block in the message list in LSM to be deallocated.

PC_SM_DEALLOCATE will return the list of message blocks to the list of free blocks in the message area of LSM. It has no return value.

PC_MP_POST()

```
void pc_mp_post(msg_ptr,dest_node)
     long *msg_ptr;
     char dest_node;
```

msg_ptr - a pointer to the first block of a message list in LSM.
dest_node - the destination task node id.

PC_MP_POST will initiate the message-passing process by placing the message pointer in either:

a) the send queue appropriate for the destination node if dest_node is not equal this node or
b) the task queue appropriate for the destination task if dest_node is equal this node.

In neither case a nor b above will PC_MP_POST return to the caller until the pointer has been successfully appended to the appropriate queue. PC_MP_POST has no return value.

PC_MP_PEND()

```
long *pc_mp_pend(dest_task_id,src_task_id)
     char dest_task_id;
     char src_task_id;
```

<u>dest_task_id</u> - the task id of the destination task.
<u>src_task_id</u> - the task id of the requested source task, or zero if the destination task merely wants the first message in the queue, regardless of who sent it.

PC_MP_PEND will return a pointer to a message in LSM to the calling task. There are two options available to the caller:

a) PC_MP_PEND may return the first message pointer found in the caller's task queue, regardless of the sender, or
b) PC_MP_PEND may return the first message pointer found in the caller's task queue to a message from a specific task.

In neither case a nor b above will PC_MP_PEND return until the requested pointer has been found.

3.2 SYSTEM TUNABLE PARAMETERS

This section discusses the various parameters that may be tuned to enhance the performance of the message-passing system. There are other system tunable parameters relating to the multi-tasker; those parameters are not discussed here.

A performance analysis has not been run on the system, thus, the current values for each of these parameters are nothing but "first guess" values. BLKSIZE has been picked to be 32 bytes because it is believed that most messages will have lengths below this value.

Queue lengths are critical parameters for system performance. Whenever a system routine attempts to append a pointer to a full queue, either the queue is compacted or the routine blocks. If queues are too small, this situation will occur frequently, thus degrading system performance. It is recommended that SEND_Q_SIZE and ACK_Q_SIZE be set to identical values because of the similar natures of these queues. TASK_Q_SIZE, however, should be calculated independently.

NAPTIME should be chosen to be significantly fewer clock ticks than the quantum size. To understand why this is recommended, consider the following scenario: a system routine suspends for NAPTIME clock ticks but there is no other task ready to execute. In this case, the routine should be unblocked as soon as possible.

BLKSIZE	definition:	size of message blocks (in long words) in LSM. Specified as an integer.
	min. value:	4.
	max. value:	limited only by LSM space.
	curr. value:	8.

SEND_Q_SIZE	definition:	length (max # of entries) of send queues. Specified as an integer.
	min. value:	2.
	max. value:	255.
	curr. value:	32.

ACK_Q_SIZE	definition:	length (max * of entries) of acknowledge queues. Specified as an integer.
	min. value:	2.
	max. value:	255.
	curr. value:	32.

TASK_Q_SIZE	definition:	length (max * of entries) of task queues. Specified as an integer.
	min. value:	2.
	max. value:	255.
	curr. value:	32.

NAPTIME	definition:	the number of clock ticks for which a process will suspend.
	min. value:	2.
	max. value:	2**32 - 1.
	curr. value:	??

3.3 DEADLOCK

There are several scenarios in which deadlock can occur in the message-passing system. It has already been mentioned that failure of any node or task may cause a system deadlock. If, for instance, task X on node i fails but other tasks continue sending messages to task X, the task queue for X will become full and node i's PIO_MP_POSTMAN will be unable to proceed. This deadlock may then propagate throughout the entire system as the send queues for node i become filled. Failure of any PIO_MP_POSTMAN or hardware failure of any node may result in a similar deadlock. In these scenarios, neither deadlock prevention nor deadlock detection is used: the system must be re-booted.

Another scenario is as follows: task Y asks to receive a message from

task Z but the task queue for task Y is filled with pointers to messages from task X. Task Y will be blocked forever, waiting for a message from Z, but, because its task queue is filled, a message from Z can never arrive. It is not known how often such a scenario will occur, but, when it does, deadlock will result. In this case, deadlock detection is used in preference to deadlock prevention. PC_MP_PEND will be able to detect the deadlock. It will send an error message to the SCB. It will not return to PC_MP_RECEIVE. This will "freeze" the state of the system, thus allowing for further diagnostics to be run, if desired. Eventually, the error message will make its way to the console. At this point, the programmer has two avenues to take to solve the problem. First, he could re-configure the message-passing system to increase the task queue length. Second, he could restructure his program to change the pattern of message transfers.

5. CONCLUSIONS

A message-passing system for the LGDF Machine has been presented. The system provides a basic mechanism for the transfer of data from one task to another and for the synchronization of tasks. Because the system places no interpretation on message contents, it can be used both for inter-(application)task communication and for (application)task-server communication. This provides a consistent interface for all system communication.

The author has attempted to keep the system simple, to facilitate future enhancements and modifications. At this stage in the development of the LGDF machine, the message-passing requirements are still somewhat undefined, thus, the message-passing system has been designed to provide a basic service which can be used to build up more complex services, such as multicasting and global streaming, should they be needed.

Perhaps the most serious problem with the message-passing system, aside from the possibility of deadlock, is the bandwidth. For each message sent, there are three copies that must be performed: from local memory to local shared memory, from local shared memory to remote shared memory, and from remote shared memory to remote local memory. Clearly, this is a time-consuming process, but it is mandated more by the hardware than the message-passing system itself. It should be pointed out, however, that

distibuted communication systems often require three copies for each message as well; from task memory to a kernel buffer, from kernel buffer to kernel buffer accross the network, and from the destination kernel buffer to the destination task memory [18].

Another possible problem relates to bottlenecks. Two bottlenecks exist in the system; the bus and the shared memory arbiter. Bus bottlenecks can be minimized by incorporating multiple busses in the design [12], by reducing bus traffic, or by reducing bus contention. The LGDF Project budget is such that the use of multiple busses was considered impractical. The message-passing system restricts all bus traffic to messages; therefore, all traffic is considered essential. However, where possible, the task loader should strive to place the heaviest communicators on the same node, thus eliminating some bus traffic. Bus contention has been minimized by giving access to the bus in round-robin fashion.

Solutions to the shared memory bottleneck also take several forms: hardware caches and software techniques to reduce the number of shared memory accesses. The Balance System [19] [20] minimizes the shared memory bottleneck by having local caches for shared variables. Cache hardware maintains consistency between caches. Due to budgetary constraints, such techniques could not be employed in the LGDF Machine: we have opted, instead, for the software solution. An attempt has been made to keep the number of accesses to variables in shared memory to a minimum, but whether the shared memory bottleneck will degrade system performance is not yet known.

A performance analysis of the system cannot be undertaken until the hardware is completed and more data is available on such factors as average message length and frequency. Until this analysis is performed, the author will venture no guesses as to system throughput.

REFERENCES

[1] *AT&T 3B2 Computer UNIX System V Release 2.0 Inter-Process Communication Utilities Software Information Bulletin*, AT&T Technologies, Inc., Issue 1, Oct. 1984.

[2] Babb II, R.G., "Data-Driven Implementation of Data Flow Diagrams," *Proc. Sixth Int'l Conf. Software Engineering*, Sept. 1982, pp. 309-318.

[3] Babb II, R.G., "Parallel Processing with Large-Grain Data Flow Techniques," *Computer*, Vol. 17, No. 7, July 1984, pp. 55-61.

[4] Birrell, Andrew D., et al., "Grapevine: An Exercise in Distributed Computing," *Comm. ACM*, Vol.25, No. 4, April 1972, pp. 260-274.

[5] Birrell, Andrew D., "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems*, Vol. 2, No. 1, Feb. 1984, pp. 39-59.

[6] Cheriton, David R., et al., "Thoth, a Portable Real-Time Operating System," *Comm. ACM*, Vol. 22, No. 2, Feb. 1979, pp. 105-114.

[7] Cheriton, David R., "The V Kernal: A Software Base for Distributed Systems," *IEEE Software*, Vol. 1, No. 2, April 1984, pp. 19-42.

[8] Cheriton, David R. and Zwaenepoel, Willy, "Distributed Process Groups in the V Kernal," *ACM Trans. Computer Systems*, Vol. 3., No. 2, May 1985, pp. 77-107.

[9] *Computer* "Special Issue on Data Flow Systems," Vol. 15, No. 2, Feb 1982.

[10] Gottlieb, A., et al., "The NYU Ultracomputer-Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. on Comp.*, Vol. C-32, No. 2, Feb. 1983, pp.175-189.

[11] *VRTX/68000 User's Guide*, Hunter & Ready, Inc., Software Release 3, Document Number 59131001, April 1985.

[12] Hwang, Kai and Briggs, Faye A., *Computer Architecture and Parallel Processing*, McGraw-Hill Book Company, 1984.

[13] Lamport, Leslie, "A New Solution of Dijkstra's Concurrent Programming Problem," *Comm. ACM*, Vol. 17, No. 8, Aug. 1974, pp. 453-455.

[14] Lamport, Leslie, "A New Approach to Proving the Correctness of Multiprocess Programs," *ACM Trans. Prog. Lang. and Systems*, Vol. 1, No. 1, July 1979, pp. 84-97.

[15] Olson, Robert, "Parallel Processing in a Message-Based Operating System," *IEEE Software*, Vol. 2, No. 4., July 1985, pp. 39-49.

[16] Ousterhout, John K., et al., "Medusa: An Experiment In Distributed Operating System Structure," *Comm. ACM*, Vol. 23, No. 2, Feb. 1980, pp. 92-105.

[17] Popek, G., et al., "LOCUS: A Network Transparent, High-Reliability Distributed System," *Proc. Eight Symp. Operating Systems Principles*, ACM, Dec. 1981, pp. 169-177.

[18] Rashid, R. and Robertson, G., "Accent: A Communication-Oriented Network Operating System Kernel," *Proc. Eight Symp. Operating Systems Principles*, ACM, Dec. 1981, pp. 64-75.

[19] *Balance 8000 Guide to Parallel Programming*, Sequent Computer Systems, Inc., 1003-41030, Rev. A, Nov. 26, 1985.

[20] *Balance 8000 System Technical Summary*, Sequent Computer Systems, Inc., 1003-41040, Rev. A, Dec. 12, 1985.

APPENDIX A
NAMING CONVENTION

All operating system programs and procedures for the LGDF Machine will obey the following naming convention. All names consist of three parts separated by "_":

(processor)_(general function)_(specific function)

(processor) is the processor or combination of processors upon which the program/procedure is to be run. Values are:

- pc - the computation processor
- pio - the communication processor
- scb - the system control board
- sio - the scb and the pio

(general function) has the following valid values:

- lm - local memory management
- sm - shared memory management
- mp - the message passing system
- ld - the task loader
- bt - hardware and VRTX configuration routines run at system boot.

(specific function) is at the programmer's discretion but should be a meaningful name, such as initialize or allocate.